Perl 3: Advanced Perl

Prerequisites, Review, and Slices

Welcome to the O'Reilly School of Technology's Perl 3 Advanced Perl course!

Course Objectives

When you complete this course, you will be able to:

- demonstrate knowledge of Perl slices.
- obtain useful info on files and on the operating system.
- manage and manipulate data using grep() and map().
- perform sleight-of-hand with references, hash references, and hashes of hashes.
- structure and optimize data.
- develop full-fledged Perl programs that employ exception-handling, multi-dimensional arrays, and regular expressions.
- implement Perl one-liners using command-line options.
- solve the Eight Queens Problem.

Introduction

If you completed OST's Perl 1 or Perl 2 courses, then we've already met. But in case you haven't, my name is Peter Scott, and I have been using and teaching Perl for over a dozen years. I'm the author of the books *Perl Debugged* and *Perl Medic*, as well as the DVD/video *Perl Fundamentals*. I've taught Perl in person to hundreds of people. I love Perl, and I sincerely want you to enjoy learning more about Perl from this course. We are always working to improve our courses to give students the best experience possible. And we rely on our students' help to reach that goal. I encourage you to let your instructor know if you see spots where the course could be improved, and also parts that you found particularly effective (so we know not to mess with those bits). We always appreciate your feedback!

In this course, written by well-known Perl trainer and author Peter Scott, you'll enter a new level of Perl expertise as you build upon skills learned in Perl 2.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your

questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.

- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

```
White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove ~~will look like this~~.

We may also include instructive comments that you don't need to type.
```

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

```
The plain black text that we present in these INTERACTIVE boxes is
provided by the system (not for you to type). The commands we want you to type look lik
e this.
```

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

```
Gray "Observe" boxes like this contain information (usually code specifics) for you to
observe.
```

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

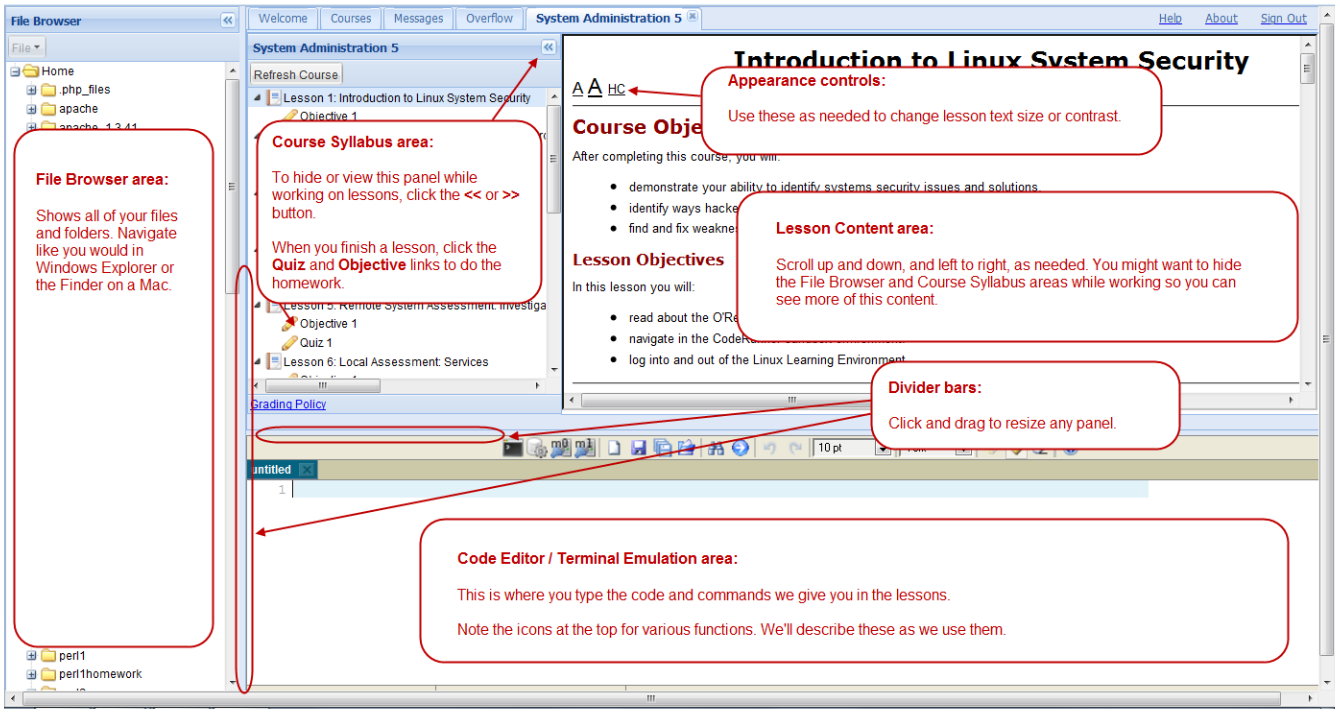
Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:



File Browser area:

Shows all of your files and folders. Navigate like you would in Windows Explorer or the Finder on a Mac.

Course Syllabus area:

To hide or view this panel while working on lessons, click the **<<** or **>>** button.

When you finish a lesson, click the **Quiz** and **Objective** links to do the homework.

Appearance controls:

Use these as needed to change lesson text size or contrast.

Lesson Content area:

Scroll up and down, and left to right, as needed. You might want to hide the File Browser and Course Syllabus areas while working so you can see more of this content.

Divider bars:

Click and drag to resize any panel.

Code Editor / Terminal Emulation area:

This is where you type the code and commands we give you in the lessons.

Note the icons at the top for various functions. We'll describe these as we use them.

These videos explain how to use CodeRunner:

File Management Demo

Code Editor Demo

Coursework Demo

Prerequisites

To thrive in this course, you should have either taken our Perl 2 course or have acquired the knowledge imparted by that course some other way. That course has as prerequisite our Perl 1 course. We'll spend some time in this lesson reviewing those prerequisites. (Of course, we can't review them completely; it took 16 lessons in Perl 1 and 15 lessons in Perl 2 to cover them. But we'll give it our best shot.)

In the next sections, we provide a detailed review of the CodeRunner environment you'll be using, and the topics covered in the Perl 1 and Perl 2 courses; we recommend going through both as a refresher, but if you've taken Perl 2 recently or otherwise feel you don't need the review, you can skip to the next section, Slices.

If you find as you review those prerequisites and the rest of this lesson that you're not sure you're ready for this course, don't worry—it happens. Don't give up, but don't try to struggle through the course if you're not sufficiently prepared. We're happy to transfer your registration to the Perl 2 or Perl 1 course so you can get well-grounded in Perl. Just contact us at info@oreillyschool.com.

Review Topics

Below is a list of topics you need have a grasp of now in order to go on and complete this course successfully. (You just need to know the basics of each topic—we'll develop these topics in more depth during this course.) There's a linked list provided in case you need to come back to these later:

- Regular Expressions: Character Classes
- Scalars, Arrays, and Hashes
- Subroutines
- Context
- Syntactic Sugar

Regex Character Classes

You must know how to craft basic regular expressions, but this does not extend to understanding lookbehind/lookahead assertions or embedded code in regexes:

OBSERVE:

```
if ( $line =~ /[aeiou]/ )
{
  print "Found a vowel";
}
```

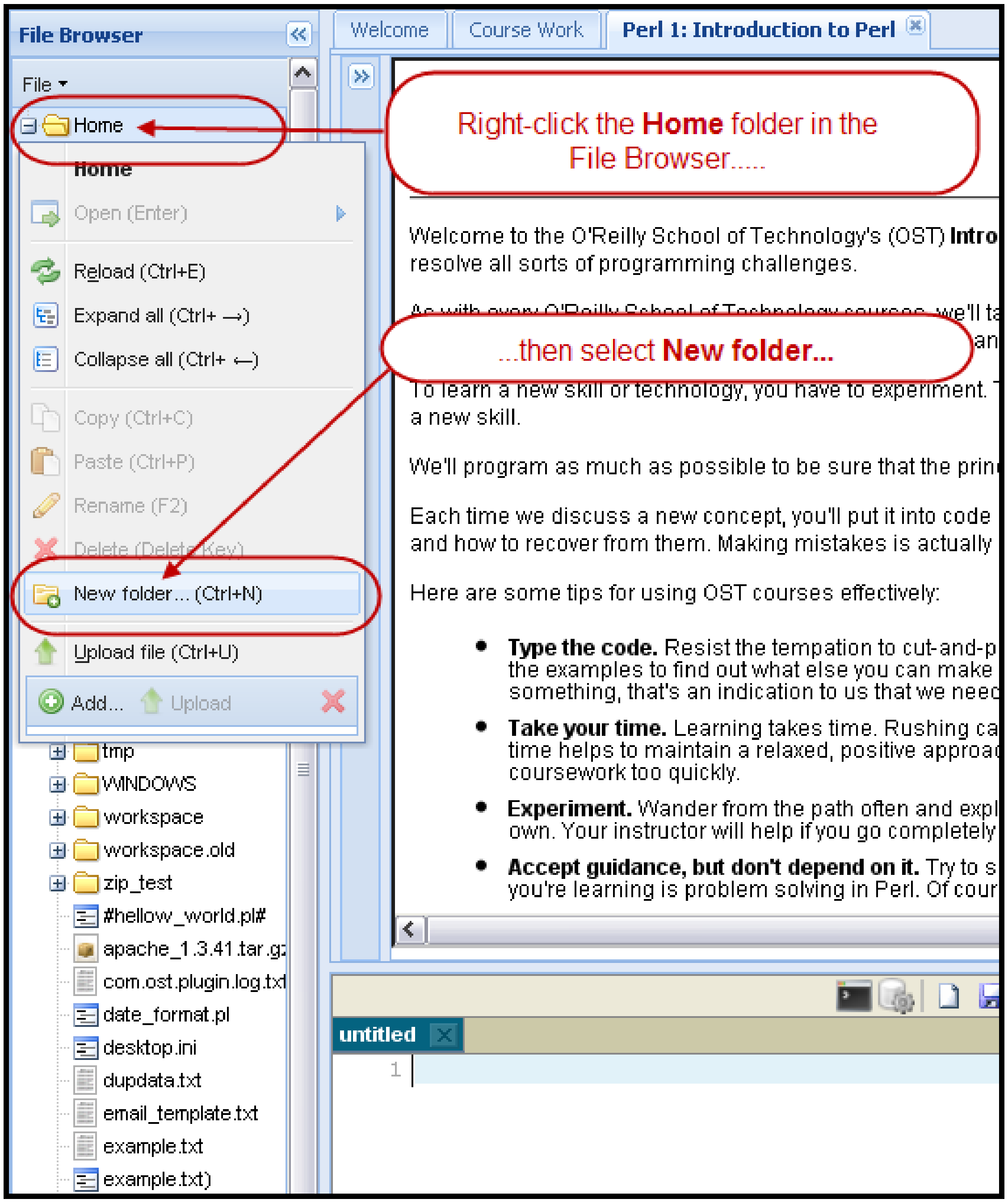
If you know what this code means, you know enough about regular expression character classes for this course.

Scalars, Arrays, and Hashes

You know how to initialize, assign, and modify variables of these types. You are able to declare your variables with **my**, you **use strict** and **use warnings** in all programs, and you understand about scoping levels created by blocks delimited by curly braces ({}), whether they are blocks that are part of looping or conditional constructs, or just plain "naked" blocks. You can iterate through the contents of an array and a hash, interpolate variables in double-quoted strings, and use **printf** for formatted printing. Let's verify those skills by doing!

For this first example, we'll demonstrate in detail, how to create and run a program in CodeRunner. After this, we'll just show you the code to type and say "Save and run it" when we want you to do this.

First, create a **/perl3** folder for your files for this course. In the upper-left corner of the CodeRunner window, select the **Home** folder, right-click it, and then select **New folder...**:



Welcome to the O'Reilly School of Technology's (OST) **Intro**
resolve all sorts of programming challenges.

As with every O'Reilly School of Technology courses, we'll ta
an

To learn a new skill or technology, you have to experiment. T
a new skill.

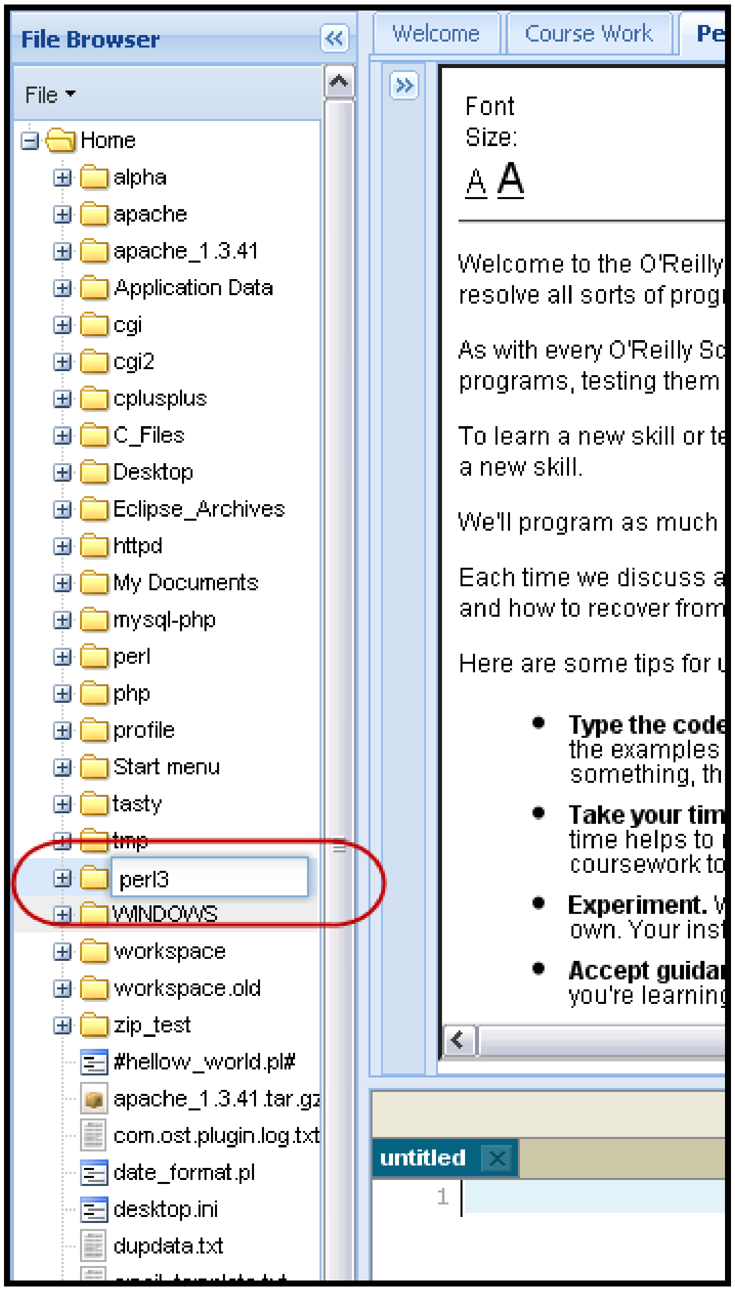
We'll program as much as possible to be sure that the prin

Each time we discuss a new concept, you'll put it into code
and how to recover from them. Making mistakes is actually

Here are some tips for using OST courses effectively:

- **Type the code.** Resist the tempation to cut-and-p
the examples to find out what else you can make
something, that's an indication to us that we need
- **Take your time.** Learning takes time. Rushing ca
time helps to maintain a relaxed, positive approa
coursework too quickly.
- **Experiment.** Wander from the path often and expl
own. Your instructor will help if you go completely
- **Accept guidance, but don't depend on it.** Try to s
you're learning is problem solving in Perl. Of cour

Replace the default folder name (untitled) with **perl3** and press **Enter**:



Welcome to the O'Reilly
resolve all sorts of progr

As with every O'Reilly Sc
programs, testing them

To learn a new skill or te
a new skill.

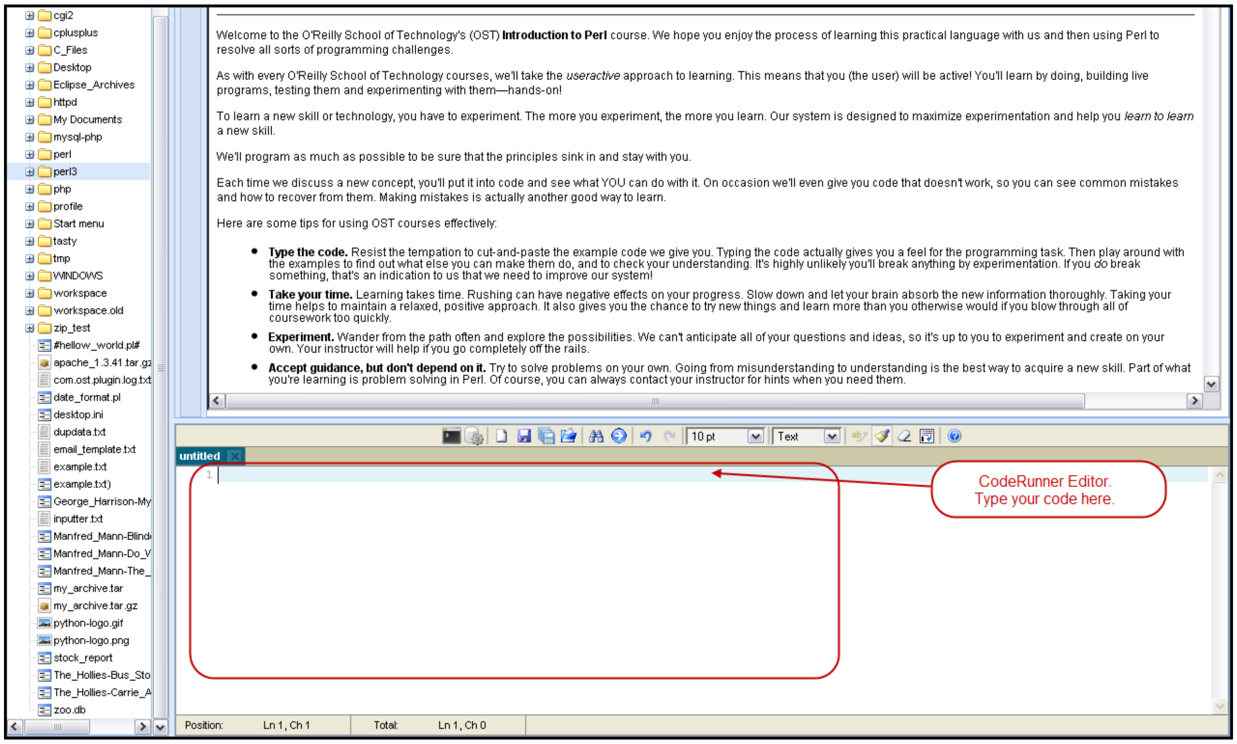
We'll program as much

Each time we discuss a
and how to recover from

Here are some tips for u

- **Type the code** the examples something, th
- **Take your tim** time helps to r coursework to
- **Experiment.** W own. Your inst
- **Accept guidar** you're learning

Take note of the editor window at the bottom of the CodeRunner window:



Welcome to the O'Reilly School of Technology's (OST) **Introduction to Perl** course. We hope you enjoy the process of learning this practical language with us and then using Perl to resolve all sorts of programming challenges.

As with every O'Reilly School of Technology courses, we'll take the *useractive* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

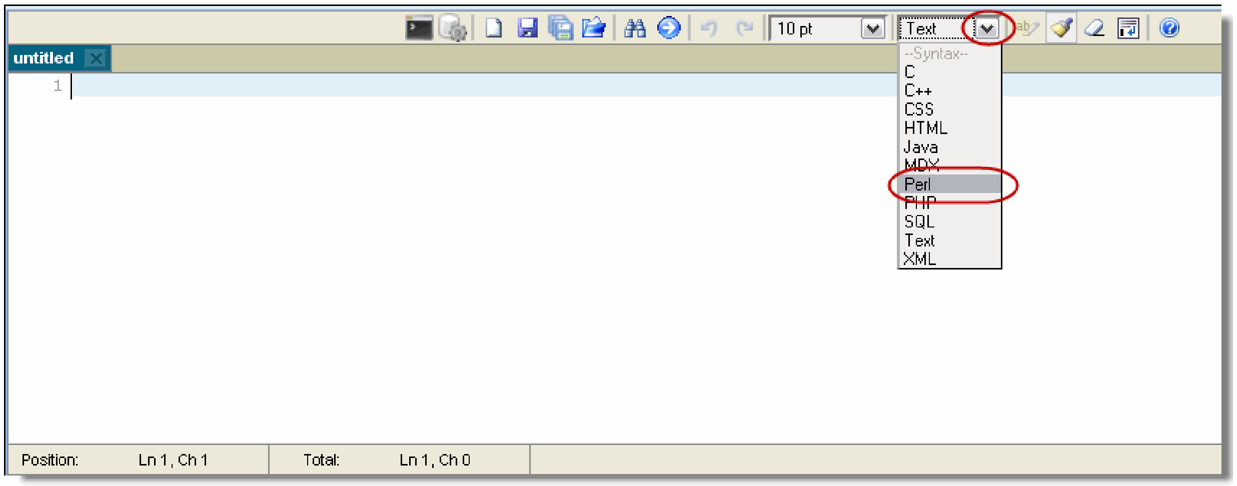
We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Here are some tips for using OST courses effectively:

- **Type the code.** Resist the temptation to cut-and-paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blow through all of coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving in Perl. Of course, you can always contact your instructor for hints when you need them.

Now, select **Perl** in the **Syntax** drop-down menu:



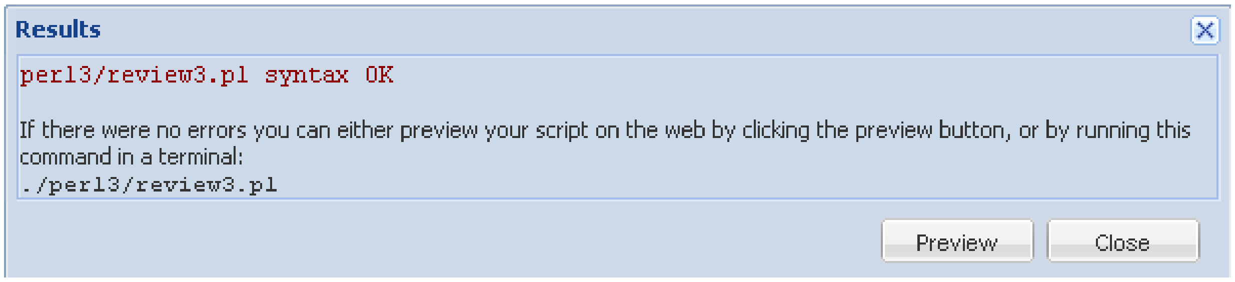
You'll see the Check Syntax icon on the left side of the toolbar.

In **review3.pl** type (don't just cut and paste!) all the code you see below so it looks like this:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $CD_LENGTH = 80;
my $song_count = 3;
my $index = 1;
my %song_length = ("Fixing a Hole" => 2 + 2/60, "Lovely Rita" => 2 + 4/60, "A Da
y in the Life" => 5 + 33/60);
print "I have $song_count MP3 files.\nTheir lengths in minutes are:\n";
my $total_length;
foreach my $title ( keys %song_length )
{
  printf "#%2d: %-30s %5.2f\n", $index++, $title, $song_length{$title};
  $total_length += $song_length{$title};
}
if ( $total_length > $CD_LENGTH )
{
  print "I can't fit all of these on a CD. Something needs to be removed.\n";
}
else
{
  print "I can fit all of these on one CD!\n";
}
```

Click the Check Syntax icon. A **Results** window appears, showing that there are no compile-time errors in the program:



Results

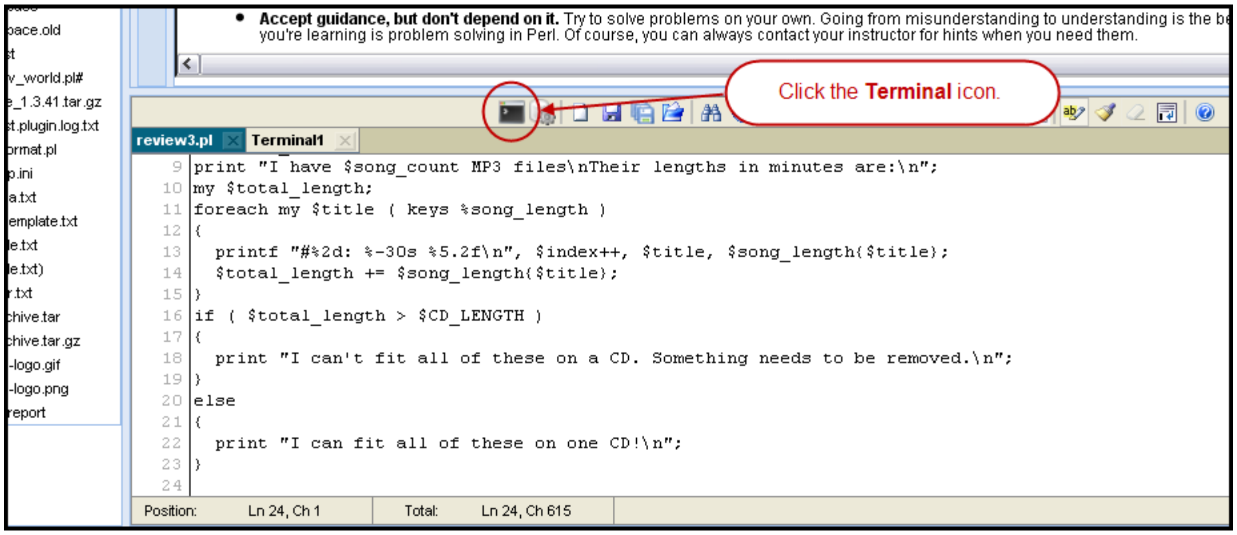
```
perl3/review3.pl syntax OK
```

If there were no errors you can either preview your script on the web by clicking the preview button, or by running this command in a terminal:

```
./perl3/review3.pl
```

Click **Close** (**Preview** won't work in this version of CodeRunner).

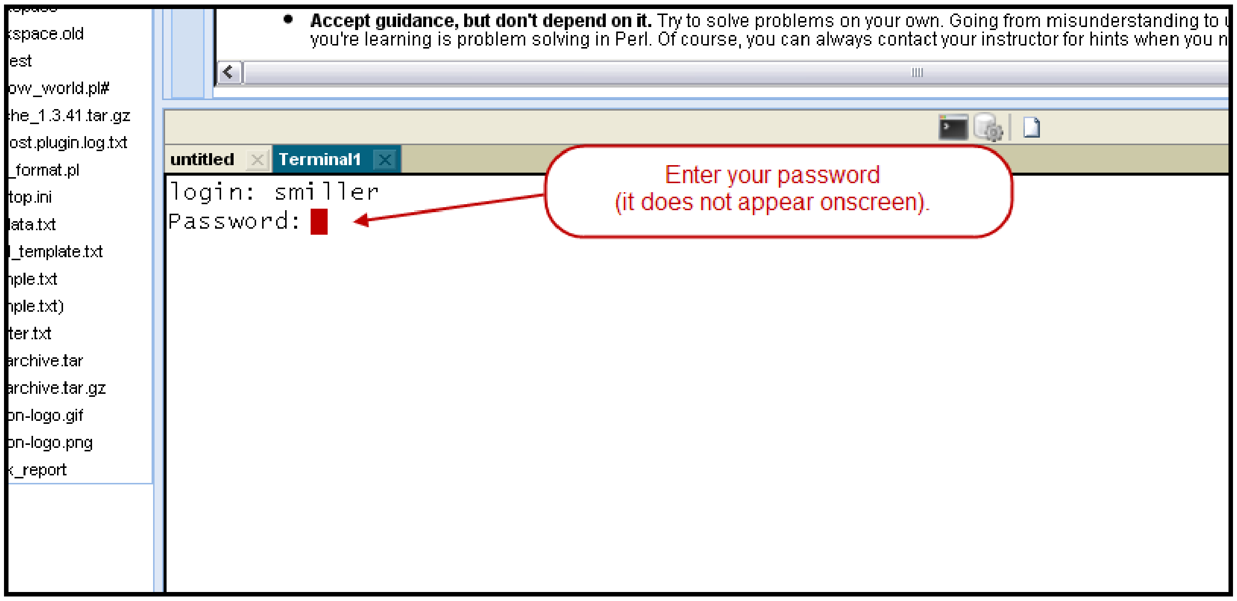
To run the program, open a terminal session by clicking the **Terminal** button:



- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the be you're learning is problem solving in Perl. Of course, you can always contact your instructor for hints when you need them.

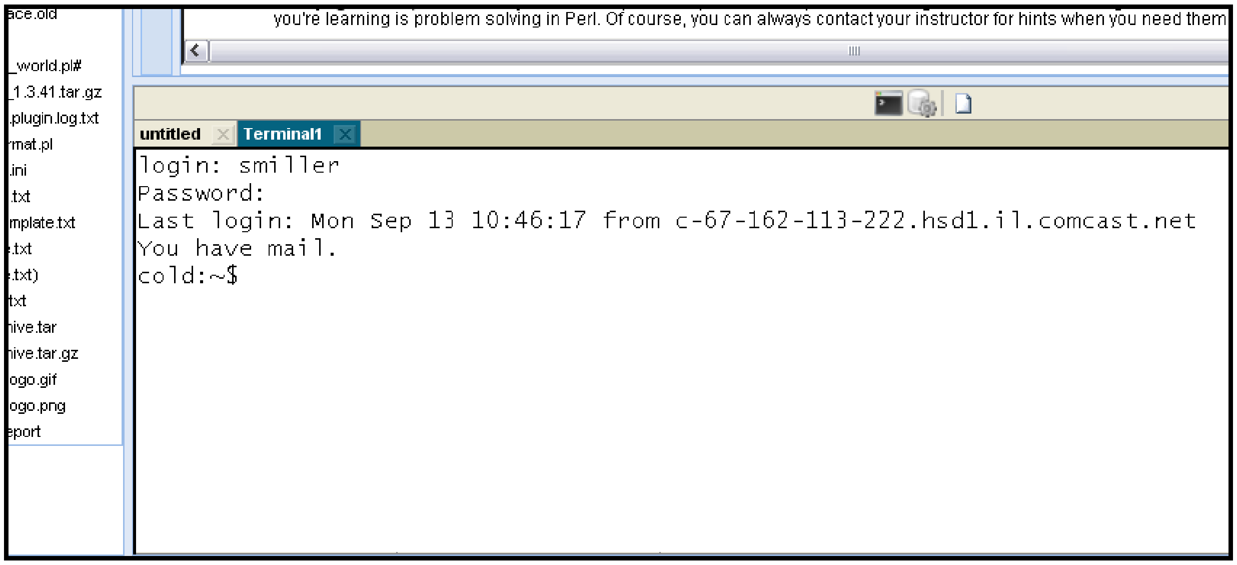
```
print "I have $song_count MP3 files\nTheir lengths in minutes are:\n";
my $total_length;
foreach my $title ( keys %song_length )
{
  printf "#%2d: %-30s %5.2f\n", $index++, $title, $song_length{$title};
  $total_length += $song_length{$title};
}
if ( $total_length > $CD_LENGTH )
{
  print "I can't fit all of these on a CD. Something needs to be removed.\n";
}
else
{
  print "I can fit all of these on one CD!\n";
}
```

You're prompted for a login name and password:


Accept guidance, but don't depend on it. Try to solve problems on your own. Going from misunderstanding to
you're learning is problem solving in Perl. Of course, you can always contact your instructor for hints when you n
untitled
Terminal1
Enter your password
(it does not appear onscreen).

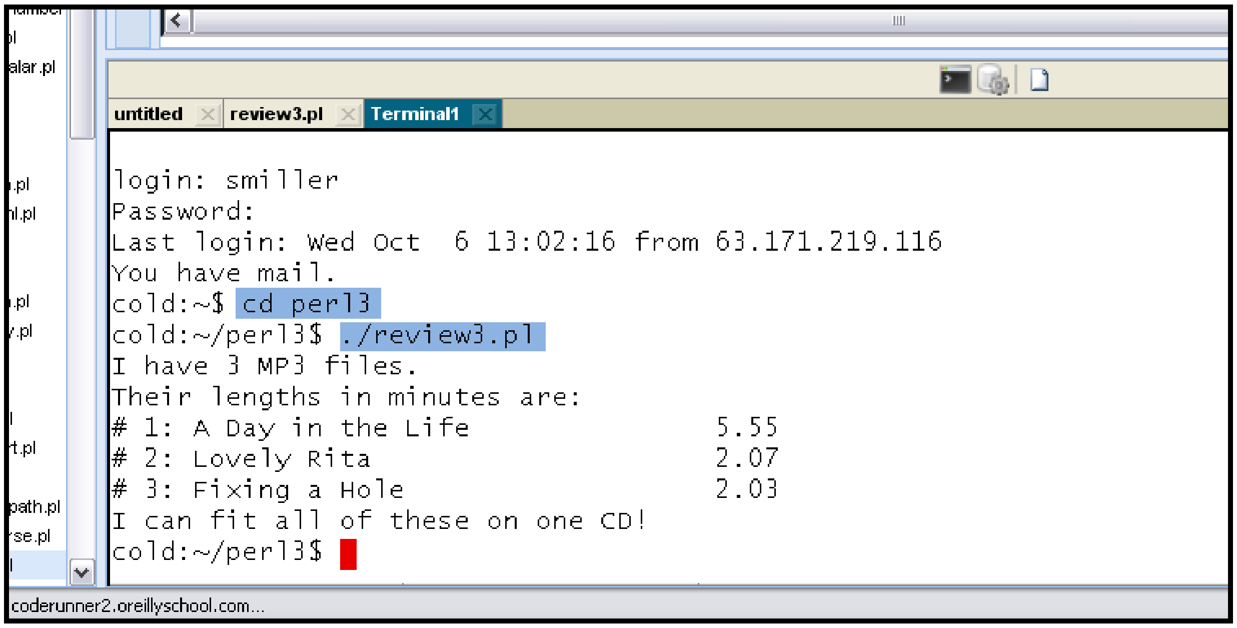
```
login: smiller
Password: 
```

Enter your O'Reilly School of Technology username and password. When you enter the correct information, you see a successful login and a "cold" prompt:


you're learning is problem solving in Perl. Of course, you can always contact your instructor for hints when you need them
untitled
Terminal1

```
login: smiller
Password:
Last login: Mon Sep 13 10:46:17 from c-67-162-113-222.hsd1.il.comcast.net
You have mail.
cold:~$
```

Now, we'll run the program and see the results; use the Unix command **cd perl3** to change to the **/perl3** folder and then type the program name preceded by **./**, as shown:



```
login: smiller
Password:
Last login: Wed Oct  6 13:02:16 from 63.171.219.116
You have mail.
cold:~$ cd perl3
cold:~/perl3$ ./review3.pl
I have 3 MP3 files.
Their lengths in minutes are:
# 1: A Day in the Life              5.55
# 2: Lovely Rita                    2.07
# 3: Fixing a Hole                  2.03
I can fit all of these on one CD!
cold:~/perl3$
```

From now on, when we want you to save and run a program, instead of repeating all of this information, we'll instruct you to "Check Syntax and run it."

Subroutines

You know how to declare and call subroutines, how to pass arbitrary arguments and return lists of values. Modify **review3.pl** by adding the blue code and deleting the ~~red~~ code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $CD_LENGTH = 80;
my $song_count = 3;
my $index = 1;

my %song_length = ("Fixing a Hole" => 2 + 2/60, "Lovely Rita" => 2 + 4/60, "A Da
y in the Life" => 5 + 33/60);
print "I have $song_count MP3 files.\nTheir lengths in minutes are:\n";
~~my $total_length;~~
foreach my $title ( keys %song_length )
{
  printf "#%2d: %-30s %5.2f\n", $index++, $title, $song_length{$title};
  ~~$total_length += $song_length{$title};~~
}
~~if ( $total_length > $CD_LENGTH )~~
if ( total_length( %song_length ) > $CD_LENGTH )
{
  print "I can't fit all of these on a CD. Something needs to be removed.\n";
}
else
{
  print "I can fit all of these on one CD!\n";
}

sub total_length
{
  my %media = @_;

  my $total = 0;
  foreach my $length ( values %media )
  {
    $total += $length;
  }
  return $total;
}
```

Check Syntax and run it. If you're not already in your /perl3 folder, use **cd perl3** to change to it:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./review3.pl
I have 3 MP3 files.
Their lengths in minutes are:
# 1: A Day in the Life 5.55
# 2: Lovely Rita 2.07
# 3: Fixing a Hole 2.03
I can fit all of these on one CD!
cold:~/perl3$~$
```

You'll see the same results.

Context

You understand the difference between scalar and list context (bonus points if you also know about boolean and void contexts!). You know that context propagates through subroutine calls all the way down. And you know what an array evaluates to in scalar and list contexts:

Modify **review3.pl** by adding the blue code and deleting the ~~red~~ code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $CD_LENGTH = 80;
~~my $song_count = 3;~~
my $index = 1;

my %song_length = ("Fixing a Hole" => 2 + 2/60, "Lovely Rita" => 2 + 4/60, "A Da
y in the Life" => 5 + 33/60);
my @songs = keys %song_length;
~~print "I have $song_count MP3 files.\nTheir lengths in minutes are:\n";~~
print "I have " . @songs . " MP3 files.\nTheir lengths in minutes are:\n";
foreach my $title ( keys %song_length )
{
  printf "#%2d: %-30s %5.2f\n", $index++, $title, $song_length{$title};
}
if ( total_length( %song_length ) > $CD_LENGTH )
{
  print "I can't fit all of these on a CD. Something needs to be removed.\n";
}
else
{
  print "I can fit all of these on one CD!\n";
}

sub total_length
{
  my %media = @_;

  my $total = 0;
  foreach my $length ( values %media )
  {
    $total += $length;
  }
  return $total;
}
```

Check Syntax and run it:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./review3.pl
I have 3 MP3 files.
Their lengths in minutes are:
# 1: A Day in the Life 5.55
# 2: Lovely Rita 2.07
# 3: Fixing a Hole 2.03
I can fit all of these on one CD!
cold:~/perl3$
```

You'll see the same results as before.

Syntactic Sugar

You know about the postfixed conditional forms and the low-precedence logical operators:

OBSERVE:

```
push @errors, $message if length $message;
@errors and die "Errors encountered: @errors\n";
```

Here, **if** the **length** of **$message** is not zero, **$message** is pushed onto the array @errors.

Modify **review3.pl** so that it takes the length of a CD as a program argument. Add the blue code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $CD_LENGTH = ( shift or die "Usage: $0 <length in minutes>\n" );
my $index = 1;

my %song_length = ("Fixing a Hole" => 2 + 2/60, "Lovely Rita" => 2 + 4/60, "A Da
y in the Life" => 5 + 33/60);
my @songs = keys %song_length;
print "I have " . @songs . " MP3 files.\nTheir lengths in minutes are:\n";
foreach my $title ( keys %song_length )
{
  printf "#%2d: %-30s %5.2f\n", $index++, $title, $song_length{$title};
}
if ( total_length( %song_length ) > $CD_LENGTH )
{
  print "I can't fit all of these on a CD. Something needs to be removed.\n";
}
else
{
  print "I can fit all of these on one CD!\n";
}

sub total_length
{
  my %media = @_;

  my $total = 0;
  foreach my $length ( values %media )
  {
    $total += $length;
  }
  return $total;
}
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./review3.pl
Usage: ./review3.pl <length in minutes>
cold:~/perl3$ ./review3.pl 80
I have 3 MP3 files.
Their lengths in minutes are:
# 1: A Day in the Life             5.55
# 2: Lovely Rita                   2.07
# 3: Fixing a Hole                 2.03
I can fit all of these on one CD!
cold:~/perl3$ ./review3.pl 5
I can't fit all of these on a CD. Something needs to be removed.
cold:~/perl3$
```

Now it returns an error if you don't provide an argument. (Also note that if the total minutes of the songs is greater than the number in the passed argment, we DO hit the other side of the if/else code.)

Input/Output

You know how to open a filehandle for reading or writing, how to read from an input filehandle with the <> ("readline") operator, and how print to an output file handle. You also know about the filehandle-less version of <> that opens files named on the command line or standard input, and the **DATA** filehandle that reads from the program file itself:

OBSERVE:

```
open my $fh, '<', $filename or die "Can't open $filename: $!\n";
while ( <$fh> ) )
{
  chomp;
  if ( s/\A<// )
  {
    push @old_file_lines, $_;
  }
  elsif ( s/\A>// )
  {
    push @new_file_lines, $_;
  }
}
```

Sorting

You can sort strings:

OBSERVE:

```
print "$_: $count{$_}\n" for sort keys %count;
```

Adding the keyword **sort** causes the output to be sorted in order of the keys in %count.

Modify **review3.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $CD_LENGTH = ( shift or die "Usage: $0 <length in minutes>\n" );
my $index = 1;

~~my %song_length = ("Fixing a Hole" => 2 + 2/60, "Lovely Rita" => 2 + 4/60, "A Da
y in the Life" => 5 + 33/60);~~

my %song_length;
while ( defined( my $line = <DATA> ) )
{
  my $title   = substr $line, 0, 17;
  my $minutes = substr $line, 18, 2;
  my $seconds = substr $line, 20, 2;
  $song_length{$title} = $minutes + $seconds / 60;
}
my @songs = keys %song_length;
print "I have " . @songs . " MP3 files.\nTheir lengths in minutes are:\n";
foreach my $title ( sort keys %song_length )
{
  printf "#%2d: %-30s %5.2f\n", $index++, $title, $song_length{$title};
}
if ( total_length( %song_length ) > $CD_LENGTH )
{
  print "I can't fit all of these on a CD. Something needs to be removed.\n";
}
else
{
  print "I can fit all of these on one CD!\n";
}

sub total_length
{
  my %media = @_;

  my $total = 0;
  foreach my $length ( values %media )
  {
    $total += $length;
  }
  return $total;
}
__END__
Fixing a Hole     2 2
Lovely Rita       2 4
A Day in the Life 5 33
```

Check Syntax and run it:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./review3.pl 80
I have 3 MP3 files.
Their lengths in minutes are:
# 1: A Day in the Life               5.55
# 2: Fixing a Hole                   2.03
# 3: Lovely Rita                     2.07
I can fit all of these on one CD!
cold:~/perl3$
```

Now our program gets its data from a text block at the end of the program, and the output is sorted.

$_

You know about Perl's "default" variable, **$_**, and many of the operators and functions that use it without mentioning it:

OBSERVE: Generic usage of $_

```
foreach ( @items )
{
  if ( defined && length )
  {
    print;
  }
}
```

Regular Expressions

You know how to determine whether text matches a regular expression, how to save parts of a match in the variables **$1**, **$2**, and so on, and you know how to use the substitution operator to change whatever matches, into something else. You know about character classes, quantifiers, and anchors:

OBSERVE: Generic regex usage

```
if ( $line =~ m/\b(\d{5})\b/ )
{
  print "Found a zip code: $1\n";
}
else
{
  $line =~ s/\d+/NOTAZIP/ and print "Replaced digits\n";
}
```

\b matches a word boundary before and after the parenthetical group **(\d{5})**. **\d{5}** matches against any 5 digits. Anything that matches the regex within the parentheses **(\d{5})** is stored in the **$1** variable, and so is printed in the "Found a zip code" statement.

If the regex does not find the specified match, the substitution after the "else" occurs, where **\d+** matches one or more digits and replaces them with **NOTAZIP**.

Modify **review3.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $CD_LENGTH = ( shift or die "Usage: $0 <length in minutes>\n" );
my $index = 1;

my %song_length;
~~while ( defined( my $line = <DATA> ) )~~
while ( <DATA> )
{
  ~~my $title   = substr $line, 0, 17;~~
  /(.*)\s+(\d+)'\s*(\d+)"/ or next;
  ~~my $minutes = substr $line, 18, 2;~~
  my ($title, $minutes, $seconds) = ($1, $2, $3);
  ~~my $seconds = substr $line, 20, 2;~~
  $song_length{$title} = $minutes + $seconds / 60;
}
my @songs = keys %song_length;
print "I have " . @songs . " MP3 files.\nTheir lengths in minutes are:\n";
foreach my $title ( sort keys %song_length )
{
  printf "#%2d: %-30s %5.2f\n", $index++, $title, $song_length{$title};
}
if ( total_length( %song_length ) > $CD_LENGTH )
{
  print "I can't fit all of these on a CD. Something needs to be removed.\n";
}
else
{
  print "I can fit all of these on one CD!\n";
}

sub total_length
{
  my %media = @_;

  my $total = 0;
  foreach my $length ( values %media )
  {
    $total += $length;
  }
  return $total;
}
__END__
~~Fixing a Hole      2 2~~
Fixing a Hole      2' 2"
~~Lovely Rita        2 4~~
Lovely Rita        2' 4"
~~A Day in the Life      5 33~~
A Day in the Life      5' 33"
```

Note how it uses the ' (single quotation mark) and " (double quotation mark) characters as markers to help interpret the data in the regular expression. Check Syntax and run it with the parameter "80" as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./review3.pl 80
I have 3 MP3 files.
Their lengths in minutes are:
# 1: A Day in the Life          5.55
# 2: Fixing a Hole              2.03
# 3: Lovely Rita                2.07
I can fit all of these on one CD!
cold:~/perl3$
```

One-Liners

You know how to run Perl code from the command line with the **-e** flag. You also know about the **-l**, **-n**, **-p**, and **-i** flags, and **BEGIN** and **END** blocks:

OBSERVE: Generic one-liners

```
perl -pi.bak -e 's[(\d{2})/(\d{2})/(\d{2})][20$3-$1-$2]'
perl -nle 'length > $max and $max = length; END{ print "Longest line is $max" }'
 *.pl
```

At the command line, type the command below as shown and see what output you get:

INTERACTIVE SESSION:

```
cold:~$ perl -le 'print "Today is " . localtime()'
cold:~$ perl -nle '$count++; END{ print "File is $count lines long" }' review3.p
l
```

Directory-Reading Functions

You know how to use **opendir()** and **readdir()** to read a directory; you also know how to use **glob()** to expand a path specification using wildcards to do much the same thing:

OBSERVE: Reading directory information with Perl functions and glob

```
opendir my $dh, "/etc" or die "Can't opendir /etc: $!\n";
print "$_\n" while readdir $dh;

perl -le 'print for glob "/etc/*.*"'
```

Wrapping Up the Review

Now let's look at some of those features at work in an example program that parses a web server log file and provides an interactive menu for choosing various reports. Create a new file called **log.pl** in your **/perl3** folder and type in the code below as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

@ARGV or die "Usage: $0 log [...]\n";

my (%remote, %method, %path, %status, %length);
my $count;
while ( <> )
{
  next unless /\A(\S+)[^"]+"([A-Z]+)\s+(\S+)[^"]+"\s+(\d+)\s+(\d+)/;
  $remote{$1}++;
  $method{$2}++;
  $path{$3}++;
  $status{$4}++;
  $length{$5}++;
  $count++;
}

print "Parsed $count records out of $. total\n";
for ( my ($what, $count) = prompt(); $what ne 'q'; ($what, $count) = prompt() )
{
  my @top = top( $count, $what eq 'r' ? %remote
                       : $what eq 'm' ? %method
                       : $what eq 'p' ? %path
                       : $what eq 's' ? %status
                       :                %length
               );
  print "Top $count:\n";
  print "$top[0]: $top[1]\n" and splice @top, 0, 2 while @top;
}

sub prompt
{
  {
    print "(q)uit or (r)emote/(m)ethod/(p)ath/(s)tatus/(l)ength <count>: ";
    chomp( $_ = <STDIN> );
    /\A(?:(q)|([rmpsl])\s+(\d+))\z/i or redo;
    return defined $1 ? lc $1 : (lc $2, $3);
  }
}

sub top
{
  my ($count, %data) = @_;

  my @keys = sort { $data{$b} <=> $data{$a} } keys %data;

  my @top;
  while ( my $key = shift @keys)
  {
    push @top, $key, $data{$key};
    last if @top == 2 * $count;
  }
  return @top;
  }
```

The file **currentlog**, located in your course materials, contains actual web server log entries from one of my sites. Use it as a sample input to this program. Run it in the terminal window as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./log.pl /software/Perl3/currentlog
Parsed 1861 records out of 2012 total
(q)uit or (r)emote/(m)ethod/(p)ath/(s)tatus/(l)ength <count>: r 5
Top 5:
91-65-217-18-dynip.superkabel.de: 55
b3091216.crawl.yahoo.net: 51
220.181.94.221: 48
abts-north-static-218.125.160.122.airtelbroadband.in: 47
sticker01.yandex.ru: 43
(q)uit or (r)emote/(m)ethod/(p)ath/(s)tatus/(l)ength <count>: m 2
Top 2:
GET: 1857
HEAD: 4
(q)uit or (r)emote/(m)ethod/(p)ath/(s)tatus/(l)ength <count>: p 8
Top 8:
/robots.txt: 210
/: 183
/gradient.php.jpg: 92
/global.css: 61
/layout.css: 61
/graphics/toplogo.jpg: 56
/graphics/company-name.jpg: 53
/graphics/nav_images/nav_off_06.gif: 53
(q)uit or (r)emote/(m)ethod/(p)ath/(s)tatus/(l)ength <count>: q
cold:~/perl3$
```

This program provides a menu of choices. You can type **q** to quit, or type letter **r**, **m**, **p**, **s**, or **l**, followed by a number. That number determines how many of the values that occur most frequently and their corresponding attributes will displayed. (For each hit, the log file contains the remote host, the type of HTTP method—usually GET—the path component of the URL, the status code of the response, and the length of content returned.)

Our program demonstrates all of these Perl features:

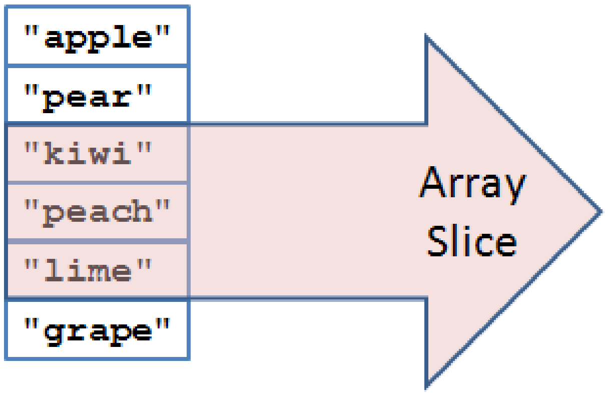
- Pragmas for ensuring safer development and execution
- Accessing command line arguments
- Low-precedence logical operators used to control flow
- Postfixed conditional statements
- Reading from files listed on the command line
- The postincrement operator
- Interpolation within double-quoted strings
- The special variable containing the number of lines last read from input
- The trinary/ternary/hook operator
- Array used in a scalar context
- Regular expression beginning and end of line anchors
- Regular expression match capturing
- Regular expression alternation
- Regular expression character classes: digits, white space
- Subroutines
- Sorting numerically and in descending order
- Array modification operators
- Changing the flow of control within loops
- Naked blocks

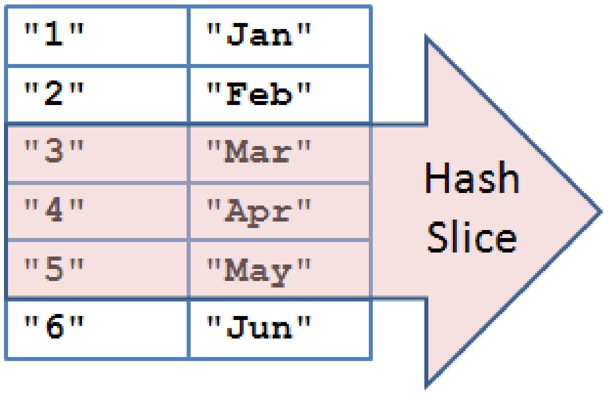
...and more! Can you locate and understand each feature within the code?

Slices

If you kept up with that *extensive* review, I think you're ready to tackle our first new topic! We'll keep it short so we don't burn you out in lesson one. We've covered a lot!

Our new topic: **slices**. A slice is just what it sounds like; part of some kind of aggregate (like an array or hash) or list. **Slicing** is a way to select a part of one of those collections:





The list slice doesn't obey quite the same rules as other lists do, but we'll get to that in a minute. First let's go over the syntax for a slice. Because a slice is a collection of multiple things, the syntax starts with an **@**:

Slice	Form	Example
Array Slice	@*array*[*list of indices*]	@fruit[1,3..5]
Hash Slice	@*hash*{*list of keys*}	@month{'Jan', 'Mar', 'Apr'}

The best way to get a feel for how this works is with an example. Create a new file called **slice.pl** in your **/perl3** folder and type in the code below as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @chars = 'a' .. 'z';
my @indices;
push @indices, rand @chars for 1 .. shift || 10;
print $chars[$_], ', ' for @indices;
print "\n";

my @beasts = qw(cat hound frog cuckoo);
my @noises = qw(miao bay ribbit cuckoo);

my %sound;
{
  my @noises_copy = @noises;
  for my $beast ( @beasts )
  {
    $sound{$beast} = shift @noises_copy;
  }
}
print "A $_ $sound{$_}s\n" for keys %sound;

my @parts = localtime;
printf "The time is %02d:%02d:%02d\n", $parts[2], $parts[1], $parts[0];
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./slice.pl
e, u, d, c, b, b, c, e, z, l,
A cuckoo cuckoos
A cat miaos
A frog ribbits
A hound bays
The time is 13:32:46
cold:~/perl3$
```

It takes an optional argument, the number of random letters to print (the default is 10).

(Of course, the actual letters printed will be different each time you run it.) This program prints random letters from the **@chars** array. The **@indices** array consists of numbers that point to elements in the **@chars** array. It also populates a hash called **%sound**, from the contents of the **@beasts** and **@sounds** arrays. (In this program we defined those arrays statically, so we could have defined the hash instead, but for now just go with it as it is.)

Now we'll modify the program so it uses slices. Add the blue code and delete the ~~red~~ code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @chars = 'a' .. 'z';
my @indices;
push @indices, rand @chars for 1 .. shift || 10;
~~print $chars[$_], ', ' for @indices;~~
~~print "\n";~~
print join( ', ' => @chars[ @indices ] ), "\n";

my @beasts = qw(cat hound frog cuckoo);
my @noises = qw(miao bay ribbit cuckoo);

my %sound;
~~{~~
~~  my @noises_copy = @noises;~~
~~  for my $beast ( @beasts )~~
~~  {~~
~~    $sound{$beast} = shift @noises_copy;~~
~~  }~~
~~}~~
@sound{ @beasts } = @noises;
print "A $_ $sound{$_}s\n" for keys %sound;

my @parts = localtime;
~~printf "The time is %02d:%02d:%02d\n", $parts[2], $parts[1], $parts[0];~~
printf "The time is %02d:%02d:%02d\n", @parts[2,1,0];
```

Check Syntax and run it. The program *works* the same way as before, but we've changed it to use a **@chars[@indices]** *array slice* first, then a **@sound{ @beasts }** *hash slice*:, and finally, a **@parts[2,1,0]** array slice.

When you use an array slice, it is as if you had typed the list of all of the corresponding elements of the array at that point. In other words, typing **@parts[2,1,0]** is exactly like typing **($parts[2], $parts[1], $parts[0])**.

In the case of a non-literal list of indices—for example, **@chars[@indices]**—the principle is the same, but you can't identify the equivalent list until the program is run.

When you use a hash slice, it is as if you had typed the list of all of the corresponding elements of the hash at that point. In other words **@sound{ @beasts }** is exactly like typing **($sound{ $beasts[0] }, $sound{ $beasts[1] }, ... $sound{ $beasts[$#beasts] })**.

This equivalence means that you can use **slices** in **lvalue** context just like we did with **@sound{ @beasts }** in the example above.

Now let's try out the list slice! Modify your code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @chars = 'a' .. 'z';
my @indices;
push @indices, rand @chars for 1 .. shift || 10;
print join( ', ' => @chars[ @indices ] ), "\n";


my @beasts = qw(cat hound frog cuckoo);
my @noises = qw(miao bay ribbit cuckoo);

my %sound;
@sound{ @beasts } = @noises;
print "A $_ $sound{$_}s\n" for keys %sound;

~~my @parts = localtime;~~
~~printf "The time is %02d:%02d:%02d\n", @parts[2,1,0];~~
my ($hr, $min, $sec) = (localtime)[2,1,0];
printf "The time is %02d:%02d:%02d\n", $hr, $min, $sec;
```

Check Syntax and run it. Now it's clear that we are extracting the hour, minute, and second components from **localtime**. (If you're unfamiliar with any of this, take a look at **perldoc -f localtime**.) And we've done it with a list slice: the syntax doesn't use an **@** symbol, it just has a bracketed list of indices following a list (in parentheses). So here, **localtime** is being called in list context.

Pop quiz! What's the difference between **$array[3]** and **@array[3]**? Ponder for a moment...good. The first one, **$array[3]**, is a *scalar*. The second one, **@array[3]**, is a *list*. It just happens to be a list of one element, but it's still a list, because it's a slice, which is equivalent to a list. A lot of Perl programmers get sloppy with their array notation and write a list when their intention is a scalar. If you did that and enabled warnings (as you always do, right?) you'd get a warning. Usually this won't make a difference—but once in a while it will. Let's create a new program called **bad_slice.pl** and see. Type the code below as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
# use warnings;

my @array;
$array[0] = localtime;
print "\$array[0] = $array[0]\n";
@array[0] = localtime;
print "\$array[0] = $array[0]\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```

cold:~/perl3$ ./bad_slice.pl
$array[0] = Thu Sep 16 14:05:06 2010
$array[0] = 6
cold:~/perl3$
```

What happened here? The first line displays the value of **localtime** in a scalar context; the second line displays the value of **localtime** in a *list* context; so the line **@array[0] = localtime;** is equivalent to the code **($array[0]) = (*seconds, minutes, hours, ...*)**.

Now you can see that **$array[0]** will be set to the first element of that list—the number of seconds—and every other element of the list returned by **localtime** will be ignored!

Okay, now uncomment the **use warnings** line. Rerun the program and you'll get output like this:

OBSERVE:

```
Scalar value @array[0] better written as $array[0] at ./perl3/bad_slice.pl line 8.
$array[0] = Thu Sep 16 14:07:52 2010
$array[0] = 52
```

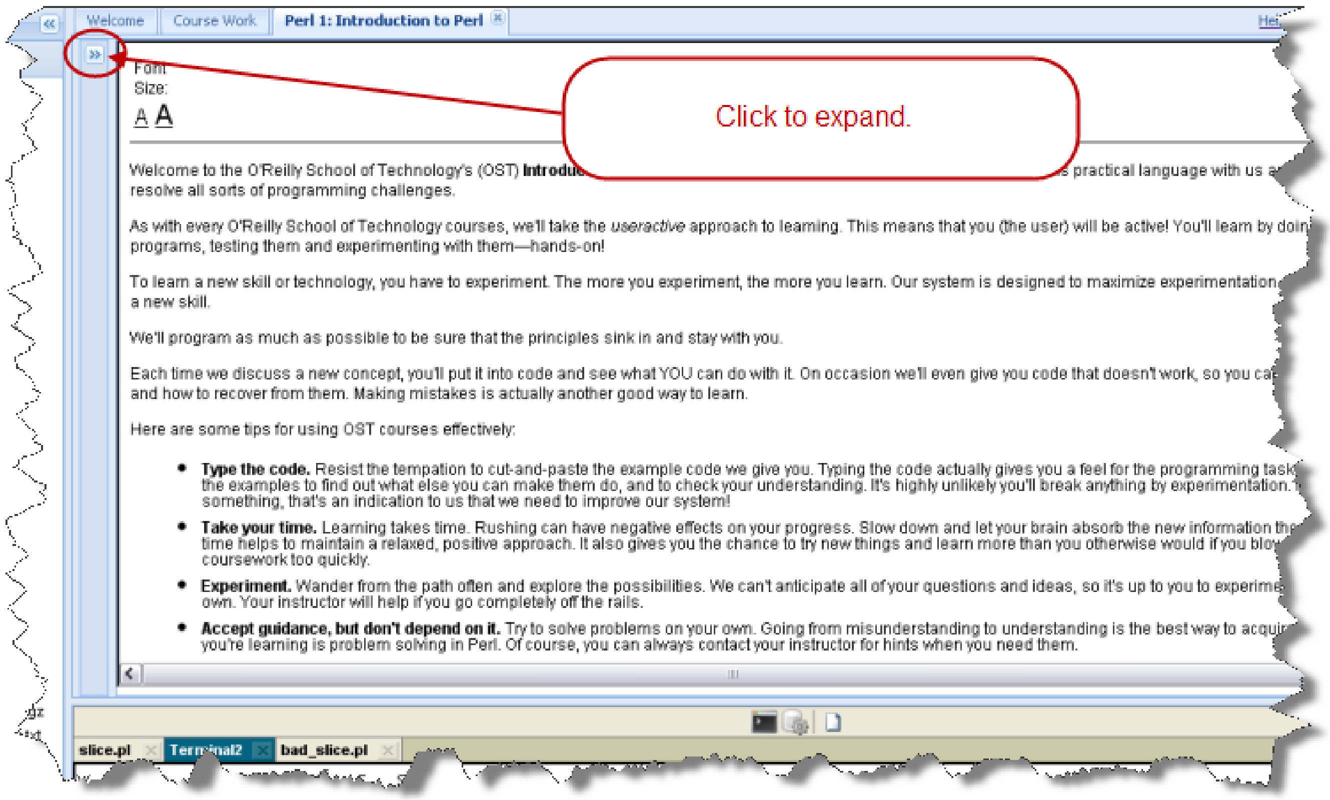
That's just one reason we always put **use warnings** in our code.

Most Perl programmers don't know about the hash slice. They're missing out! But not you.

And one more thing—don't confuse a *slice* with the built-in function *splice*. They may sound similar, but that's about all they have in common.

Alright then, you're off and running! See you in the next lesson!

Make sure you complete the homework from your syllabus (do the quiz *before* the project):



Welcome to the O'Reilly School of Technology's (OST) **Introdu** s practical language with us a resolve all sorts of programming challenges.

As with every O'Reilly School of Technology courses, we'll take the *useractive* approach to learning. This means that you (the user) will be active! You'll learn by doin programs, testing them and experimenting with them—hands-on!

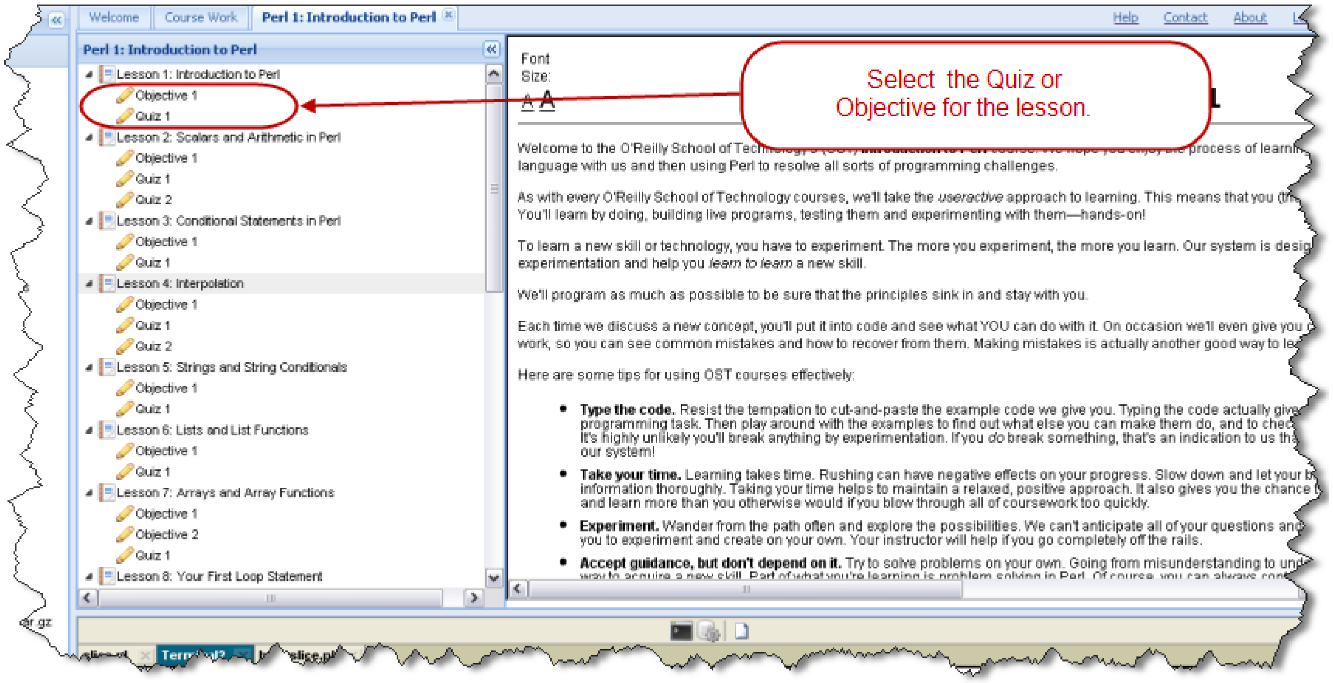
To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you ca and how to recover from them. Making mistakes is actually another good way to learn.

Here are some tips for using OST courses effectively:

- **Type the code.** Resist the tempation to cut-and-paste the example code we give you. Typing the code actually gives you a feel for the programming task the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information th time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blo coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experime own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acqui you're learning is problem solving in Perl. Of course, you can always contact your instructor for hints when you need them.


Welcome
Course Work
Perl 1: Introduction to Perl
Help
Contact
About
Perl 1: Introduction to Perl
Lesson 1: Introduction to Perl
Objective 1
Quiz 1
Lesson 2: Scalars and Arithmetic in Perl
Objective 1
Quiz 1
Quiz 2
Lesson 3: Conditional Statements in Perl
Objective 1
Quiz 1
Lesson 4: Interpolation
Objective 1
Quiz 1
Quiz 2
Lesson 5: Strings and String Conditionals
Objective 1
Quiz 1
Lesson 6: Lists and List Functions
Objective 1
Quiz 1
Lesson 7: Arrays and Array Functions
Objective 1
Objective 2
Quiz 1
Lesson 8: Your First Loop Statement
Font Size:
Select the Quiz or Objective for the lesson.

Welcome to the O'Reilly School of Tech language with us and then using Perl to resolve all sorts of programming challenges.

As with every O'Reilly School of Technology courses, we'll take the *useractive* approach to learning. This means that you (th You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is desig experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you c work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to le

Here are some tips for using OST courses effectively:

- **Type the code.** Resist the temptation to cut-and-paste the example code we give you. Typing the code actually give programming task. Then play around with the examples to find out what else you can make them do, and to chec It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us th our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your b information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance t and learn more than you otherwise would if you blow through all of coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to und

Discovering Properties of Files

When you complete this lesson, you will be able to:

Lesson Objectives

- use File Test Operators.
- use the stat Function.

Welcome back to your *Advanced Perl* course! In this lesson we'll learn about two special Perl operators that test properties of files.

File Test Operators

Each file has specific properties that you may need to know, like how old it is, or whether it's a directory (folder). Perl has a few different operators that can provide that information. Let's take a look at those. In your **/perl3** folder, create a program named **filetest.pl**, then type in the code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $file = shift || $0;
print "$file exists\n"              if -e $file;
print "$file is a regular file\n"   if -f $file;
print "$file is a directory\n"      if -d $file;
print "$file is a symbolic link\n"  if -l $file;
print "$file is empty\n"            if -z $file;
my $size = -s $file;
print "$file has size $size\n"      if $size;
my $old = -M $file;
print "$file is $old days old\n"    if defined $old;
print "$file is readable\n"         if -r $file;
print "$file is executable\n"       if -x $file;
print "$file is special\n"          if -b $file || -c $file;
```

You can run this program with any filename as its argument; if you don't give it an argument, it will test itself. Type the following commands in the Terminal window:

INTERACTIVE SESSION:

```
cold:~$ cd perl3
cold:~/perl3$ ./filetest.pl
./filetest.pl exists
./filetest.pl is a regular file
./filetest.pl has size 591
./filetest.pl is 0.00285879629629963 days old
./filetest.pl is readable
./filetest.pl is executable
```

Here's how this program works. Each file test operator appears to be a minus sign (**-**) followed by a letter. That may seem weird, like there's some kind of subtraction going on—there isn't.

Note This syntax was chosen to match the syntax used in the Bourne and C shells. The creators of Perl wanted users of earlier languages to be able to learn Perl quickly, so they incorporated many of their features.

There are several additional file test operators in Perl than are shown in that program; you can get a complete list of them by typing **perldoc -f -X**. Here are the test operators that we used in **filetest.pl**:

Operator	Meaning
-e	Does file exist?
-f	Is file a regular file?
-d	Is file a directory?
-l	Is file a symbolic link?
-z	Is file zero length (empty)?
-s	How big is file (in bytes)?
-M	How old is file (in days)?
-r	Is file readable (by current process)?
-x	Is file executable (by current process)?
-b	Is file a block special file?
-c	Is file a character special file?

All of these operators are boolean (return a true or false result), except for **-s** and **-M**, which return integer and floating point numbers, respectively. Of course, you can also use the **-s** operator in a boolean context to mean, "does the file have any contents?"

Type the commands below to try out a couple more invocations of the program:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./filetest.pl .
cold:~/perl3$ ./filetest.pl /dev/tty
```

What other files can you find on the system that will produce different results? What happens if you run the program on a file that doesn't exist?

The operators take a filename as argument; if you don't supply one, then they'll use **$_** by default. As of Perl version 5.10, it's possible to chain or stack multiple operators together as a shorthand way to "AND" them together logically. Modify **filetest.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $file = shift || $0;
print "$file exists\n"              if -e $file;
print "$file is a regular file\n"   if -f $file;
print "$file is a directory\n"      if -d $file;
print "$file is a symbolic link\n"  if -l $file;
print "$file is empty\n"            if -z $file;
my $size = -s $file;
print "$file has size $size\n"      if $size;
my $old = -M $file;
print "$file is $old days old\n"    if defined $old;
print "$file is readable\n"         if -r $file;
print "$file is executable\n"       if -x $file;
print "$file is special\n"          if -b $file || -c $file;
print "Perl version is $]\n";
if ( $] >= 5.010 )
{
  print "$file is text, writable (chaining is allowed)\n" if -w -T $file;
}
else
{
  print "$file is text, writable (chaining is not allowed)\n" if -w $file && -T $file;
}
```

Here (for Perl version 5.010 or later) we've stacked the two tests **-w** and **-T**:

Operator	Meaning
-w	Is file writeable (by current process)?
-T	Is file text (as opposed to binary)?

This form of stacking is more efficient than using **-w $file && -T $file**, as you would have had to do in versions of Perl prior to 5.10. We used the special variable **$]**, which contains Perl's version number. The code varies according to which version of Perl is running.

Check Syntax and run it to see which version of Perl we're using.

The stat Function

The file test operators don't tell you everything you want to know about a file though; for example, you may need to know who owns it. That information (and more) is provided by the built-in function **stat()**:

OBSERVE: stat()

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,$atime,$mtime,$ctime,$blksize,$blocks) =
stat($filename)
```

This table contains definitions for the elements in that list:

Index in list above	Variable in list above	Meaning
0	$dev	device number of filesystem
1	$ino	inode number
2	$mode	file mode (type and permissions)
3	$nlink	number of (hard) links to the file
4	$uid	numeric user ID of file's owner
5	$gid	numeric group ID of file's owner

6	$rdev	the device identifier (special files only)
7	$size	total size of file, in bytes
8	$atime	last access time in seconds since the epoch
9	$mtime	last modify time in seconds since the epoch
10	$ctime	inode change time in seconds since the epoch (*)
11	$blksize	preferred block size for file system I/O
12	$blocks	actual number of blocks allocated

Most of this information is esoteric and rarely useful. **stat()** returns all of the information that the system call of the same name provides. (Keep in mind that on some operating systems, some of the information you see in the table is not available.) The attributes that are most often useful are the file mode, its owner, and the time a file was last modified. (The size is also useful, but you can get that information more directly by using the **-s** operator.) Let's see those more useful elements in action. Create a new file named **stat.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $file = shift || $0;
my ($mode, $uid, $mtime) = (stat $file)[2,4,9];
printf "Mode of %s is %o\n", $file, $mode;
print "UID owning $file is $uid\n";
print "Modification time of $file is $mtime\n";
```

Check Syntax and run it as shown below:

INTERACTIVE SESSION:

```
cold:~/perl3$  ./stat.pl
Mode of ./stat.pl is 100755
UID owning ./stat.pl is 16948
Modification time of ./stat.pl is 1285169770
```

Do you see how we used the list slice **[2,4,9]** to retrieve only the elements we want? (Referring to the indices in the table above is helpful here. You can get the same table using **perldoc -f stat**.)

The times returned by **stat()** are in seconds since midnight, January 1, 1970 (the "epoch"). This is the same time reference that is returned by the **time()** function and accepted by the **localtime()** function. Now let's make the output more user-friendly. Modify your code as shown by adding the blue code and removing the ~~red~~ code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $file = shift || $0;
my ($mode, $uid, $mtime) = (stat $file)[2,4,9];
printf "Mode of %s is %o\n", $file, $mode;
print "UID owning $file is $uid\n";
~~print "Modification time of $file is $mtime\n";~~
print "Modification time of $file is " . localtime( $mtime ) . "\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./stat.pl
Mode of ./stat.pl is 100755
UID owning ./stat.pl is 16948
Modification time of ./stat.pl is Wed Sep 22 10:58:09 2010
```

You get these results because we used **localtime()** in a scalar context.

The UID (user ID) is numeric. But having the name of that UID would probably be more useful. We can get that with the **getpwuid()** built-in function. Modify your code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $file = shift || $0;
my ($mode, $uid, $mtime) = (stat $file)[2,4,9];
printf "Mode of %s is %o\n", $file, $mode;
~~print "UID owning $file is $uid\n";~~
my $owner = getpwuid $uid;
print "Owner of $file is $owner\n";
print "Modification time of $file is " . localtime( $mtime ) . "\n";
```

Check Syntax and run it as shown here:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./stat.pl
Mode of ./stat.pl is 100755
Owner of ./stat.pl is smiller
Modification time of ./stat.pl is Wed Sep 22 11:34:07 2010
cold:~/perl3$
```

So why does the mode of this file display as 100755? Hmm. Run **ls -l** on the file as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ls -l ./stat.pl
-rwxr-xr-x 1 smiller webusers 290 Sep 22 11:34 ./stat.pl
```

We get a result of **-rwxr-xr-x**, which is 755. The answer to our earlier question lies in that almost invisible minus sign at the beginning of the **ls** output. It indicates which *type* of file we have. In this case, it's a regular file; other possibilities include **d** for a directory, **l** for a symbolic link, and so on. The information regarding type is included in **$mode**. The 1000000 bit corresponds to a regular file. Just try running **stat.pl** on a directory!

Let's try a file example for fun. Create a new file named **oldest.pl** in the **/perl3** folder:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $dir = shift || '.';
opendir my $dh, $dir or die "opendir $dir: $!\n";
my %age;
while ( my $file = readdir $dh )
{
  next if $file =~ /\A\.\.?\z/;
  $age{$file} = -M "$dir/$file";
}

print "The oldest 50% of files are:\n";
my $index = 0;
for my $file ( sort { $age{$b} <=> $age{$a} } keys %age )
{
  print "$file ($age{$file} days)\n";
  $index++;
  last if $index >= (keys %age) / 2;
}
```

This program prints the names of the older half of the files in the given directory (or the current directory, by default). Now let's run it on **/etc**:

INTERACTIVE SESSION:

```
cold:~/perl3$  ./oldest.pl /etc
The oldest 50% of files are:
urlview.conf (3716.21444444444 days)
rpc (3294.24068287037 days)
quotatab (3222.75721064815 days)
minicom.users (2716.80438657407 days)
nsswitch.conf (2174.5394212963 days)
[...etc...]
```

Read the program carefully and make sure you understand how it works, then experiment with it!

Once you feel confident with these properties of files, you're ready to move on to the next lesson where we'll tackle interacting with the operating system!

When you finish this lesson, complete the homework from your syllabus.

Interacting With the System

Lesson Objectives

When you complete this lesson, you will be able to:

- access the process environment.
- use the system Command.
- use exec.
- use backticks to capture whatever that program sends to standard output and put it in a scalar or array.
- use Piped Opens to open a pipe to or from a program, and read from or write to that pipe from Perl.

Welcome back! In this lesson we'll learn ways that your Perl program interacts with the system, by accessing the process environment and by running external programs.

The Environment

Every process that runs has an available *environment*. The environment contains a number of *variables* that are places to store information. You can access the list of variables contained in an environment on Unix with the **env** command, and on Windows with the **set** command. Try the **env** command on our system. At your **cold:~$** prompt, type the command below as shown:

INTERACTIVE SESSION:

```
cold:~$ env
```

You'll see output that looks like this:

OBSERVE:

```
PATH=/usr/local/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/X11R6/bin
SECURITYSESSIONID=174015b0
HOME=/Users/peter
SHELL=/bin/tcsh
USER=peter
__CF_USER_TEXT_ENCODING=0x1F5:0:0
DISPLAY=:0.0
MANPATH=/usr/share/man:/usr/local/share/man:/usr/X11R6/man
HOSTTYPE=powermac
VENDOR=apple
OSTYPE=darwin
MACHTYPE=powerpc
SHLVL=2
PWD=/Users/peter
LOGNAME=peter
GROUP=peter
HOST=marvin.local
TERM=vt100
WINDOWID=8388624
```

The output you get could be a lot different from the output you see in the box above, because I ran my command on a Mac (**env** is useful for discovering which kind of system you're running).

The output is a series of lines in this format: *variable=value*. The *variable* is the name of a variable, the *value* is the content of that variable. You can use any one of these pairs of *variable=value* pairs at the command line, much like a Perl variable:

INTERACTIVE SESSION:

```
cold:~$ echo $TERM
xterm
```

And you can change or create environment variables yourself, like this:

INTERACTIVE SESSION:

```
cold:~$ export FOOD=chocolate
cold:~$ echo $FOOD
chocolate
```

Most of the time, environment variables are used to pass information to *child processes*. Initially, we set an environment variable at the command line (like **FOOD** above). The child processes are all of the programs that you invoke from the same shell (terminal window)—in our classes, those are usually Perl programs. The variables that are there already were set by various scripts that are configured to run every time you start a shell or log in to the system.

Environment variables are *inherited* by child processes. If, after entering the command above, you were to start a child shell, and then run **env**, you'd see that it already had the variable **FOOD** set to **chocolate**.

But enough about shell commands—we're here to learn how to work in *Perl!* Perl has a special hash that is mapped to the environment. It's called **%ENV**. Each key is a variable name. If you change an entry in the hash, Perl changes the environment; this is one of the "magic" variables in Perl.

Let's try it out. Type the one-liner below as shown:

INTERACTIVE SESSION:

```
cold:~$ perl -le 'print "$_ = $ENV{$_}" for sort keys %ENV'
```

You get more or less the same environment variables you got from **env** in the shell. (Any differences are the results of the shell setting or changing variables as it started a new child process to run Perl.)

A common mistake programmers make is attempting to change the environment of a *parent* process. It's impossible to do this from Perl or *any other* programming language. Try it:

INTERACTIVE SESSION:

```
cold:~$ perl -e '$ENV{FOOD} = "Vanilla"'
cold:~$ env | sort
```

Notice that **FOOD** is still set to **chocolate** from our previous entry in the output (which we piped through **sort** to make it easier to search). That's how the environment works. When you use it in a Perl program, most of the time you'll be reading a variable that was set by a parent process. (Once in a while you'll set a variable that will be read by a child process—more on that later in the lesson).

An environment variable of special note is **PATH**. This variable contains a ordered list of paths to search for programs that are not found in the current directory and are run without specifying an absolute or relative path. So, when you type **ls**, for example, at the command line, your shell looks through each of the directories named in **PATH** until it finds one that contains the executable file **ls**, at which point it runs that file. We can use Perl to tell us where it located that program! Run this one-liner:

INTERACTIVE SESSION:

```
cold:~$ perl -le 'my $prog = shift or die; print $_, -x "$_/$prog" ? " <-- HERE" : "" f
or split /:/, $ENV{PATH}' ls
```

Your output will look something like this:

OBSERVE:

```
/usr/kerberos/bin
/usr/local/bin
/bin <-- HERE
/usr/bin
/usr/X11R6/bin
/usr/local/java/bin
/users/pscott/bin
```

Congratulations! You've just invented a streamlined version of the system utility **whereis**. Now try running that one-liner a few more times, each time replacing **ls** with one of the programs here:

- jdb
- php
- perl
- sclient

You could of course write that one-liner out as a program in a file, particularly if you wanted to call it again (although in its present form it could easily be assigned to a shell alias). If you find it more concise than you would like, I encourage you to rewrite it as a program. It is important that you understand all the code we create, whether it's in the form of a program or a one-liner. Sometimes you need to read a one-liner very carefully to understand it, but that's an important skill too, because one-liners are a common use of Perl and it's vital that you are comfortable both with reading and writing them.

You can see that some programs exist in more than one directory—that's handy to know; maybe the programs are different. When you just type the name of the program at the command prompt, you'll only ever execute the *first* one in your path.

Note

The term "path" has several different meanings in the Unix world; this could cause confusion. The context helps you to figure out which meaning is intended. Sometimes path refers to the address of a file, such as **/etc/passwd**, **public_html/images/squirrel472.jpg**, **../sibling/cousin.txt**, or **todo.list**. The first path example, **/etc/passwd**, is an *absolute* path, because it refers to the same file regardless of your current directory. The next two examples, **public_html/images/squirrel472.jpg** and **../sibling/cousin.txt**, are *relative* paths, which are found by starting from the current directory. The last address in the list, **todo.list**, refers to a file in the current directory; it is the same as the relative path **./todo.list**. Generally, we refer to the last component (like **cousin.txt**) as a *filename*, but it is also a (degenerate) case of a "path."

We've introduced the environment variable PATH, or in Perl, **$ENV{PATH}**. In this context, path refers to an ordered set of directories to search whenever the name of a program is typed at the command line without a directory as part of the name. In this case, the bare file name refers to a file in the current directory only if the '**.**' (period) is present in PATH and the file does not occur in a directory named earlier in PATH.

The system Command

Now let's see how we can run a separate program from a Perl program. Some programmers do this far too much. For instance, they run the Unix **date** program from Perl and parse the result to get the date, instead of doing it more efficiently from directly inside Perl with the **localtime()** function. In general, whenever you think you need to run an external program to execute a task, consider whether there is a way to do it from Perl itself. There are modules available to perform just about any task conceivable.

Our first examples of the **system()** function can be done in pure Perl, but we'll set that aside for this lesson. Suppose that your program used a lot of time to create an entire directory tree of temporary files under **/tmp/myprog**, and you want to remove all of them. Rather than going through all of the subdirectories that might have been created (and rather than use the **File::Path module**, which provides the **remove_tree()** function for this purpose), we'll call on the recursive flag of the system's **rm** command to do it for us. Check it out:

OBSERVE: system

```
system "rm -r /tmp/myprog";
```

You could use parentheses around the arguments, but most programmers don't. When you run **system()** (*command*), the information that *command* sends to standard output and standard error will appear on your terminal (unless you've already redirected those streams). Similarly, if the command gets data from standard input, the command retrieves that data from the same source that standard input is currently using.

Let's try to use the **system()** function in a program. Create **edit.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $temp_file = "$ENV{'HOME'}/.edit.pl.$$";
unlink $temp_file; # In case I left one lying around

my $date = localtime;
{
  open my $fh, '>', $temp_file or die "open $temp_file: $!\n";
  print {$fh} <<"END_OF_TEXT";
This file was written on $date by $0.
Here's your chance to customize it!
END_OF_TEXT
}

print "Stand by to edit!\n";
system "vi $temp_file";

print "Contents of $temp_file are now:\n";
{
  open my $fh, '<', $temp_file or die "open $temp_file: $!\n";
  print while <$fh>;
}
unlink $temp_file;
```

Check Syntax and run it. This program brings you to the **vi** editor so you can edit the temporary file that was created. Here are a few tips for accessing and editing in the **vi** editor:

- To get into insert mode, type **i**.
- To get out of insert mode, press the **Esc** key.
- To save and close the file, press **:** and then type **x**.

You can use the **system()** function in the same way to allow your users to edit and customize output files in the middle of a program, for example, to customize a report before it is emailed.

Programmers usually set the environment variable **EDITOR** globally to contain the program name of their favorite editor on the system.

Okay. Let's go ahead and edit the file. Run the command below as shown:

INTERACTIVE SESSION:

```
cold:~$ cd perl3
cold:~/perl3$ export EDITOR=emacs
```

Note When you close and restart the terminal session, the variable will change back to its default.

Now change your program to recognize the EDITOR variable. Add the blue code and delete the ~~red~~ code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $temp_file = "$ENV{HOME}/.edit.pl.$$";
unlink $temp_file; # In case I left one lying around

my $date = localtime;
{
  open my $fh, '>', $temp_file or die "open $temp_file: $!\n";
  print {$fh} <<"END_OF_TEXT";
This file was written on $date by $0.
Here's your chance to customize it!
END_OF_TEXT
}

print "Stand by to edit!\n";
~~system "vi $temp_file";~~
my $editor = $ENV{EDITOR} || 'vi';
system "$editor $temp_file";

print "Contents of $temp_file are now:\n";
{
  open my $fh, '<', $temp_file or die "open $temp_file: $!\n";
  print while <$fh>;
}
unlink $temp_file;
```

Check Syntax and run it. The file opens in emacs. Add some text, and save the change by pressing **Ctrl+x** and then **Ctrl+s**. Exit emacs by pressing **Ctrl+x** and then **Ctrl+c**.

The **system()** function has rules about whether it passes its argument to the shell for interpretation. If there is only one argument and it contains shell metacharacters—characters that the shell interprets in a particular way—then it will be run through a shell. A common use of this feature is to run a command and ignore its output. Create **grep.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $string = shift or die "Usage: $0 string [file]\n";
my $file   = shift || $0;
system "grep $string $file";
print "$string ", $? == 0 ? "was" : "wasn't", " found in $file\n";
```

Check Syntax and run it as shown (changing to your **/perl3** folder if necessary):

INTERACTIVE SESSION:

```
cold:~/perl3$ ./grep.pl
Usage: ./grep.pl string [file]
cold:~/perl3$ ./grep.pl perl
#!/usr/bin/perl
perl was found in ./perl3/grep.pl
cold:~/perl3$ ./grep.pl qqq
qqq wasn't found in ./grep.pl
```

Our program uses another feature of Perl to find out the *exit status* of the **grep** program: the special variable **$?**, which contains the exit status of the last program to be run via **system()**. By convention, programs exit with a code of zero to indicate success. **grep** exits with a non-zero code when it doesn't find the search expression in the text it's searching. (In this program, the default file being searched is the program itself.) We're being a little bit lazy here in using **grep** to tell us whether a certain string (or regular expression) is contained in a file, rather than writing the equivalent Perl code for that purpose. It's also less than elegant; **grep** prints out the lines that match, and they get in the way. You can provide **grep** with an option that allows it to suppress those lines, but let's pretend we don't know about it for now. Instead, we'll redirect that output to **/dev/null** so we don't see it. Modify your code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $string = shift or die "Usage: $0 string [file]\n";
my $file   = shift || $0;
system "grep $string $file >/dev/null 2>&1";
print "$string ", $? == 0 ? "was" : "wasn't", " found in $file\n";
```

Check Syntax and run it again with the same inputs:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./grep.pl perl
perl was found in ./grep.pl
cold:~/perl3$ ./grep.pl qqq
qqq wasn't found in ./grep.pl
```

Here we redirect both standard output and standard error to **/dev/null**. This is standard Bourne shell syntax. Because the metacharacters **&** and **>** are in the string passed to **system()**, Perl passes the command through the shell so those characters get interpreted according to your instructions.

exec

The **exec()** function is closely related to **system()**. The two functions operate identically except that **exec()** causes the current process to be *replaced* by the one being invoked. Think of **exec() @args** as a shorthand for this:

OBSERVE:

```
system @args;
exit;
```

...except **exec @args** is more efficient with memory. We can use **exec @args** to run the **perldoc** program to format and output help text embedded in *our own program*, for when the user wants help. Let's give that a try. Create a new file and call it **help.pl**, then type in the code below:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

if ( @ARGV && shift eq '-h' )
{
  exec "perldoc -t $0";
}

print "I guess you don't need any help...\n";

__END__

=head1 NAME

help.pl - Demonstration of perldoc

=head1 SYNOPSIS

./help.pl

=head1 DESCRIPTION

This program runs C<exec> to invoke perldoc on I<ourselves>.

=cut
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./help.pl
I guess you don't need any help...
cold:~/perl3$ ./help.pl -h
NAME
    help.pl - Demonstration of perldoc

SYNOPSIS
    ./help.pl

DESCRIPTION
    This program runs "exec" to invoke perldoc on *ourselves*.
/tmp/LAWyNHwdun (END)
```

Press **Q** to end the program and return to the command prompt. **perldoc** reads the documentation that is embedded in the program after the **__END__** line (which prevents parsing by *Perl*). It is in a format called **POD** (Plain Old Documentation, a highly technical term), which **perldoc** understands. An in-depth discussion of POD syntax is outside the scope of this course (you can read about it in http://perldoc.perl.org/perlpod.html), but I'm using POD here anyway, because it's a pretty cool use of **exec()**.

Backticks

system() and **exec()** have worked well for us so far, but suppose we want to get at the *output* of a program. We need something that will capture whatever that program sends to standard output and put it in a scalar or array. Perl's **backticks** are one way we can do that. Take a look:

OBSERVE:

```
$output = `program`;   # Scalar context
@output = `program`;   # List context
```

In scalar context, the entire output to standard output is returned in a single string. In list context, the entire output is returned as a list of strings, one for each line. Let's do an example that uses backticks. Create **ps.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %pid_count;
for ( `ps -elf` )
{
  my $ppid = (split)[4];
  next if $ppid eq 'PPID';  # Header line
  $pid_count{$ppid}++;
}

print "Most popular parent process:\n";
for ( sort { $pid_count{$b} <=> $pid_count{$a} } keys %pid_count )
{
  print "\t$_ ($pid_count{$_} instances)\n";
  last;
}
```

Check Syntax and run it as shown here:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./ps.pl
Most popular parent process:
        2 (42 instances)
```

Our code reports on the most popular parent process id listed in the output from the **ps** program (to see what that process id looks like, run **ps -elf** at the command line). It finds that entry by sorting the whole list in descending order and then exiting the loop after printing the first entry. (We used a list slice on the return from **split** to extract only the fifth element of each line.)

Piped Opens

You can use the **open** function to open a pipe to or from a program, and read from or write to that pipe from Perl. Let's start with an example that reads the lines that are output by the program **last**, a standard Unix program that reports the time of a user's last login. It generates output that looks like this:

OBSERVE:

```
peter    pts/0        192.168.1.3      Mon Sep 16 20:17   still logged in
peter    pts/1        pepe             Mon Sep  6 14:41 - 14:54  (00:12)
peter    pts/0        192.168.1.3      Mon Sep  6 12:44 - 14:56  (02:12)
steve    pts/1        pepe             Mon Sep  6 09:13 - 09:38  (00:24)
peter    pts/0        192.168.1.3      Mon Sep  6 06:47 - 09:27  (02:40)
steve    pts/0        192.168.1.3      Mon Sep  6 05:57 - 06:41  (00:44)
reboot   system boot  2.6.27.24-170.2. Tue May  4 08:05         (42+10:05)
[... more similar lines...]
```

The first word on each line is the username, eventually followed by the date. Now suppose we want a program that prints the first line listed by **last** for each distinct username; in other words, shows us the last time each user known to the **last** program logged on. Create **last.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

open my $fh, '-|', "last" or die "Can't open pipe: $!\n";
my %seen;
while ( <$fh> )
{
  next unless /\A(\S+).*\s((?:[A-Z][a-z]{2} ){2}[ \d]\d \d\d:\d\d)/;
  print "$1 - $2\n" unless $seen{$1}++;
}
```

Check Syntax and run it. The output will look something like this:

OBSERVE:

```
peter - Mon Sep  6 20:17
tim - Sat Sep  4 18:11
cam - Mon Aug 16 15:48
svn - Wed Jul  7 19:27
steve - Thu May 20 14:36
reboot - Tue May  4 08:05
```

The form of the *piped open* for input from a program is:

OBSERVE:

```
open filehandle, '-|', program
```

Take a look at the mode **'-|'** here where we have previously had **open** modes of <, or >, or >>. Using the **-|** mode, **program** will be run and its standard output will be connected to **filehandle**.

Using the **-|** mode you can start reading output as soon as it becomes available. With backticks, you have to wait for the program to finish running before you can get *any* of the output.

The regular expression here is worth studying too. We can't just **split** the line on white space, because sometimes the second column contains a space ("system boot"). We could unpack the fixed-width columns, but instead we craft a regular expression to match the date and time. See if you can figure out how it works!

You can also open a pipe *to* a program, so that whatever you write on that filehandle goes to the standard input of the program. The form for that looks like this:

OBSERVE:

```
open filehandle, '|-', program
```

Let's try an example! Create **od.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

open my $fh, '|-', 'od -c' or die "open: $!\n";
print {$fh} "Hello world! \e \a \b \r \n";
close $fh;
```

Check Syntax and run it as shown here:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./od.pl
0000000   H   e   l   l   o       w   o   r   l   d   !     033       \a
0000020      \b      \r      \n
0000026
```

This is a basic example of a piped open for output. We sent a fixed string to the **od** program, which is designed to display data that may contain unprintable characters in a form that shows their octal code or other representation. Here, the printable characters have been rendered explicitly; the **od** program knows symbolic forms for all of the digraphs I used except for the **\e** for the ESCAPE character.

Note You cannot open a pipe to a program for *both* reading and writing. (In other words, there is no **-|-** mode.)

The special variable **$?** is also set to the exit status of a program that is run via backticks or a piped open.

Phew! That was a pretty long lesson, but worth the trip. Now that you have a handle on ways to access the process environment and by run external programs, you're ready to take on regular expressions! We'll do that in the next lesson! See you in there...

Don't forget to go back to the syllabus to complete the homework.

Regular Expressions: Global Matches and More

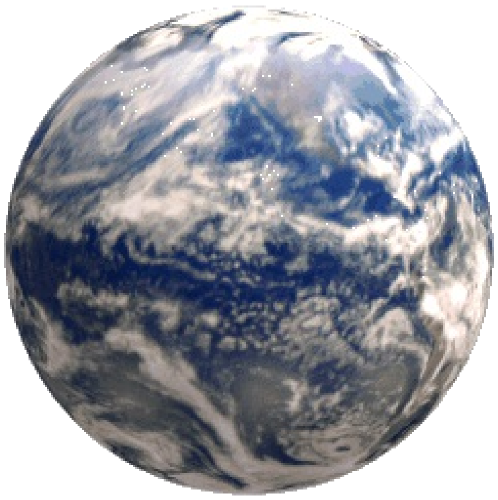
Lesson Objectives

When you complete this lesson, you will be able to:

- make Global Matches.
- Match and Substitute in Scalar Context.
- Match and Substitute in List Context.
- utilize Global Matching in Scalar and List Contexts.
- utilize Global Substitution in Scalar and List Contexts.
- perform Nongreedy Matching.

Global Matches

"It has been said that arguing against globalization is like arguing against the laws of gravity."
-Kofi Annan



Let's go with the flow, then, and look at what globalization means in Perl. So today, we will think globally (and act locally, right there on your computer). We're going to introduce a new *modifier* on regular expressions: the **/g** or *global* modifier.

/g is both simple and powerful. Your code reads it as, "Don't stop with the first match; keep going!" Let's start with a basic example to get a feel for it. Create **g_simple.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = "The cad saw far rat map bay\n";

print;

s/a/o/;

print;

s/a/o/g;

print;
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
code:~$ cd perl3
cold:~/perl3$ ./g_simple.pl
The cad saw far rat map bay
The cod saw far rat map bay
The cod sow for rot mop boy
```

(Don't worry, it's not supposed to make sense.) The first line is the string (in **$_**) in its initial state. The second line is the string after a substitution with no modifiers; the first letter, **a**, was changed to an **o**, but the rest remain unchanged. The third line is the string after a *global* substitution using the **/g** modifier; *all* of the remaining occurrences of the letter **a** are changed to **o**.

If you want to specify an additional modifier (for instance **/i** for case insensitivity), combine the letters in any order: **/.../gi** or **/.../ig** will both work the same way.

Now let's take a look at how the match and substitution operators evaluate in different contexts, and then how they evaluate with the **/g** modifier.

Match and Substitution in Scalar Context

In a scalar context, a match returns true or false depending on whether it succeeded. We'll demonstrate that with an example. Create **pig_sing.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $line = "Never try to teach a pig to sing.";

my $result = ( $line =~ /\bpig\b/ );
print "  Successful match result: $result\n";
   $result = ( $line =~ /jackass/ );
print "Unsuccessful match result: ", (defined $result ? "'$result'" : 'undef'), "\n";

$_ = "It wastes your time and annoys the pig.";
$result = /\bpig\b/;
print "  Successful match result: $result\n";

$result = /jackass/;
print "Unsuccessful match result: ", (defined $result ? "'$result'" : 'undef'), "\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./pig_sing.pl
  Successful match result: 1
Unsuccessful match result: ''
  Successful match result: 1
Unsuccessful match result: ''
```

A successful match in scalar context yields a true value (Perl uses the number **1**), and an unsuccessful match in scalar context yields a false value (Perl uses the empty string). You've been using matches in the conditions of **if** and **while** statements for a while now, but we included it here so you could see the match operation when it's bound to an explicit variable (**$line**) and when implicitly bound to **$_**.

Now let's see what a substitution does in scalar context. Modify **pig_sing.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $line = "Never try to teach a pig to sing.";

my $result = ( $line =~ s/\bpig\b/squirrel/ );
print "  Successful substitution result: $result\n";
   $result = ( $line =~ s/jackass/zebra/ );
print "Unsuccessful substitution result: ", (defined $result ? "'$result'" : 'undef'),
"\n";

$_ = "It wastes your time and annoys the pig.";
$result = s/\bpig\b/squirrel/;
print "  Successful substitution result: $result\n";

$result = s/jackass/zebra/;
print "Unsuccessful substitution result: ", (defined $result ? "'$result'" : 'undef'),
"\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./pig_sing.pl
  Successful substitution result: 1
Unsuccessful substitution result: ''
  Successful substitution result: 1
Unsuccessful substitution result: ''
```

A successful substitution in scalar context yields a true value. An unsuccessful substitution in scalar context yields a false value (again, Perl uses the empty string). Here you can see what the substitution operation looks like when bound to an explicit variable (**$line**) and when implicitly bound to **$_**.

Match and Substitution in List Context

In a list context, an unsuccessful match returns the empty list; a successful match is a bit more interesting. Modify **pig_sing.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $line = "Never try to teach a pig to sing.";

my @results = ( $line =~ ~~s~~/\bpig\b/~~squirrel/~~ );
print "  Successful match result: @results\n";
   @results = ( $line =~ ~~s~~/jackass/~~zebra/~~ );
print "Unsuccessful match result: ", (~~defined ~~@results ? "'@results'" : '~~undef~~<none>'),
 "\n";

$_ = "It wastes your time and annoys the pig.";
@results = ~~s~~/\b(pig)\b/~~squirrel/~~;
print "  Successful match result: @results\n";

@results = ~~s~~/(jackass)/~~zebra/~~;
print "Unsuccessful match result: ", (~~defined~~ @results ? "'@results'" : '~~undef~~<none>'),
 "\n";

@results = /\s(time).*(p.g)/;
print "  Successful match result: @results\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./pig_sing.pl
  Successful match result: 1
Unsuccessful match result: <none>
  Successful match result: pig
Unsuccessful match result: <none>
  Successful match result: time pig
```

In the first successful match, the result is a list containing the single element **1**. In the second successful match, the regex contains a capturing group, and the result is a list of the portions of the input string that matched the capture: the list **($1)**. In the third successful match, the regex contains *two* capturing groups, and the result is the list of that which matched both captures: **($1, $2)**.

The general rule is that a successful match in list context returns a list of that which matched each set of capturing parentheses, and, if there are no captures, it returns the list **(1)**.

Now let's take a brief detour to consider list *assignment* in *scalar* context. We'll examine that using this one-liner:

INTERACTIVE SESSION:

```
cold:~/perl3$ perl -le '$x = (@y = 6..10); print $x'
5
cold:~/perl3$
```

The list assigned to **@y** contains five items. We place that list assignment in scalar context and assign the result to **$x**. When we print **$x**, the result is **5**, which demonstrates that the result of a list assignment in scalar context is equal to the number of items in the list. Try another example:

INTERACTIVE SESSION:

```
cold:~/perl3$ perl -le '$x = ( ($a,$b,$c) = 6..10 ); print $x'
5
cold:~/perl3$
```

When we combine list assignment in scalar context with the list context result of a match, the result is a really useful Perl idiom. Modify **pig_sing.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $line = "Never try to teach a pig to sing.";

if ( my ($res) = ( $line =~ /\b(pig)\b/ ) )
~~my @results = ( $line =~ /\bpig\b/ );~~
~~print "  Successful match result: @results\n";~~
~~   @results = ( $line =~ /jackass/ );~~
~~print "Unsuccessful match result: ", (@results ? "'@results'" : '<none>'), "\n";~~

{
  print "Successful match result: $res\n";
}

$_ = "It wastes your time and annoys the pig.";

~~@results = /\b(pig)\b/;~~
~~print "  Successful match result: @results\n";~~

~~@results = /(jackass)/;~~
~~print "Unsuccessful match result: ", (@results ? "'@results'" : '<none>'), "\n";~~

~~@results = /\s(time).*(p.g)/;~~
~~print "  Successful match result: @results\n";~~

if ( my ($nomatch) = /(aardvark)/ )
{
  print "You won't see this\n";
}
elsif ( my ($first, $second) = /\s(time).*(pig)/ )
{
  print "Successful match result: $first, $second\n";
}
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./pig_sing.pl
Successful match result: pig
Successful match result: time, pig
```

What's going on here? If the match is successful, there will be a list assignment containing either one thing (in the first two matches) or two things (in the last match). In a scalar context, that list assignment will evaluate as **1** or **2** respectively, both of which are true. But if the match is unsuccessful, there will be nothing in the list, so the number of things assigned will be zero, which is false. Take a look at the result of assigning an empty list in scalar context. Type the command below as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ perl -le '$x = ( ($a,$b,$c) = () ); print $x'
0
cold:~/perl3$
```

The idiom is of the form: **if (my ($var1, $var2,...) = /...(...)...(...).../)**.

This lets us assign the matches to variables with meaningful names at the same time that we test whether the match was successful. This is more readable than the alternative:

OBSERVE:

```
if ( /...(...)...(...).../ )
{
  my ($var1, $var2, ...) = ($1, $2, ...);
```

Now let's see a substitution in list context. Modify **pig_sing.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $line = "Never try to teach a pig to sing.";

~~if ( my ($res) = ( $line =~ /\b(pig)\b/ ) )~~
~~{~~
~~  print "Successful match result: $res\n";~~
~~}~~

my ($result) = ( $line =~ s/\bpig\b/squirrel/ );
print "  Successful substitution result: $result\n";
   ($result) = ( $line =~ s/jackass/zebra/ );
print "Unsuccessful substitution result: ", (defined $result ? "'$result'" : 'undef'),
"\n";

$_ = "It wastes your time and annoys the pig.";

~~if ( my ($nomatch) = /(aardvark)/ )~~
~~{~~
~~  print "You won't see this\n";~~
~~}~~
~~elsif ( my ($first, $second) = /\s(time).*(pig)/ )~~
~~{~~
~~  print "Successful match result: $first, $second\n";~~
~~}~~
 ($result)  = s/\b(pig)\b/squirrel/;
print "  Successful substitution result: $result\n";

 ($result)  = s/(jackass)/zebra/;
print "Unsuccessful substitution result: ", (defined $result ? "'$result'" : 'undef'),
"\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./pig_sing.pl
  Successful substitution result: 1
Unsuccessful substitution result: ''
  Successful substitution result: 1
Unsuccessful substitution result: ''
```

We get the same result as we did in scalar context.

Global Matching: Scalar and List Contexts

Now let's see the effect of **/g** in a scalar context. Create **g_match_number.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

while ( <DATA> )
{
  print "Starting to match line: ";
  while ( /\d+/g )
  {
    print "Match! ";
  }
  print "\n";
}

__END__
Sing a song of 6 pence, a pocket full of rye; 4 and 20 blackbirds, baked in a pie.
1969: Apollo 11 returns from the Moon after travelling 828743 nautical miles.
The first five values of Ackermann's function for n=1 are 2, 3, 5, 13, and 65533.
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./g_match_number.pl
Starting to match line: Match! Match! Match!
Starting to match line: Match! Match! Match!
Starting to match line: Match! Match! Match! Match! Match! Match!
```

We get a match for each number found in each of the lines. The **match** operator matches each item possible, in turn, returning true each time it is called, until it can't match any more in the input, at which point it returns false. Now let's do something more interesting with each number. Modify **g_match_number.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

while ( <DATA> )
{
  print "Starting to match line: ";
  while ( /(\d+)/g )
  {
    print "~~Match! ~~- $1";
  }
  print "\n";
}

__END__
Sing a song of 6 pence, a pocket full of rye; 4 and 20 blackbirds, baked in a pie.
1969: Apollo 11 returns from the Moon after travelling 828743 nautical miles.
The first five values of Ackermann's function for n=1 are 2, 3, 5, 13, and 65533.
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./g_match_number.pl
Starting to match line: - 6 - 4 - 20
Starting to match line: - 1969 - 11 - 828743
Starting to match line: - 1 - 2 - 3 - 5 - 13 - 65533
```

What happened here? Now we have a capturing group (inside the parentheses) in the regex, so just like before, we go through the **while** loop every time there is a match, but this time we save whatever matched **\d+** into **$1**.

Now we're going to make a mistake on purpose. After our earlier discussion about saving captures into variables as we do the match, you might be tempted to make this change (make the changes as shown, but don't run it yet):

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

while ( <DATA> )
{
  print "Starting to match line: ";
  while ( my ($number) = /(\d+)/g )
  {
    ~~print "- $1";~~
    print "- $number";
  }
  print "\n";
}

__END__
Sing a song of 6 pence, a pocket full of rye; 4 and 20 blackbirds, baked in a pie.
1969: Apollo 11 returns from the Moon after travelling 828743 nautical miles.
The first five values of Ackermann's function for n=1 are 2, 3, 5, 13, and 65533.
```

Check Syntax and save it, but when you run it, be prepared to press **Ctrl+C** as quickly as possible. Okay, ready? Run it, and press **Ctrl+C**! (It may take a while to stop running.) See how the program went into an infinite loop? Why do you suppose that happened?

It's because we are no longer using the **/g** modifier in a *scalar* context. By assigning the match to the list containing the single element **$number**, we have put **$number** in *list* context. And I haven't told you what **/g** does for a match in list context yet!

Let's try that now. Modify **g_match_number.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

while ( <DATA> )
{
  print "Starting to match line: ";
  ~~while ( my ($number) = /(\d+)/g )~~
  for my $number ( /(\d+)/g )
  {
    print " - $number";
  }
  print "\n";
}

__END__
Sing a song of 6 pence, a pocket full of rye; 4 and 20 blackbirds, baked in a pie.
1969: Apollo 11 returns from the Moon after travelling 828743 nautical miles.
The first five values of Ackermann's function for n=1 are 2, 3, 5, 13, and 65533.
```

Check Syntax and run it. It's similar to the last version that worked. keep in mind that **while** imposes scalar context and **for(each)** imposes list context. You should see this:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./g_match_number.pl
Starting to match line: - 6 - 4 - 20
Starting to match line: - 1969 - 11 - 828743
Starting to match line: - 1 - 2 - 3 - 5 - 13 - 65533
```

In a list context, a global match returns a list of everything that was matched by the capturing parentheses. Let's explore this functionality some more. Create **g_match_number2.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $line = "01/30/10 17:30:21 trish.oreilly.com ENABLE /perl1 & 02/01/10 09:10:00 tim.o
reilly.com ENROLL /perl1";

my ($date1, $date2) = ( $line =~ m!(\d+/\d+/\d+)!g );
print "Date: $date1; date: $date2\n";
```

Check Syntax and run it as shown:

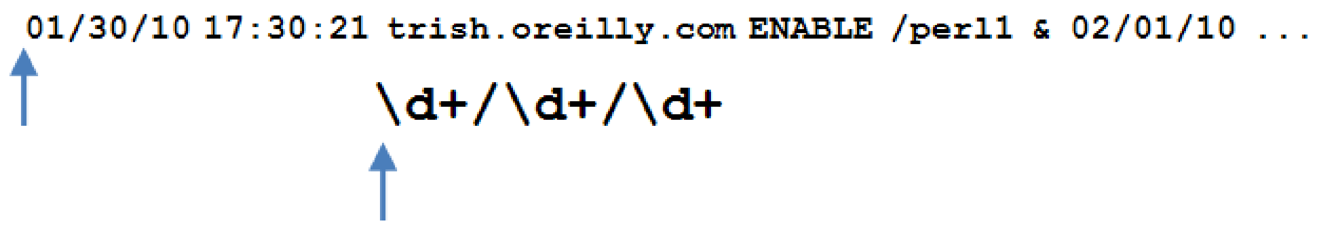
INTERACTIVE SESSION:

```
cold:~/perl3$ ./g_match_number2.pl
Date: 01/30/10; date: 02/01/10
```

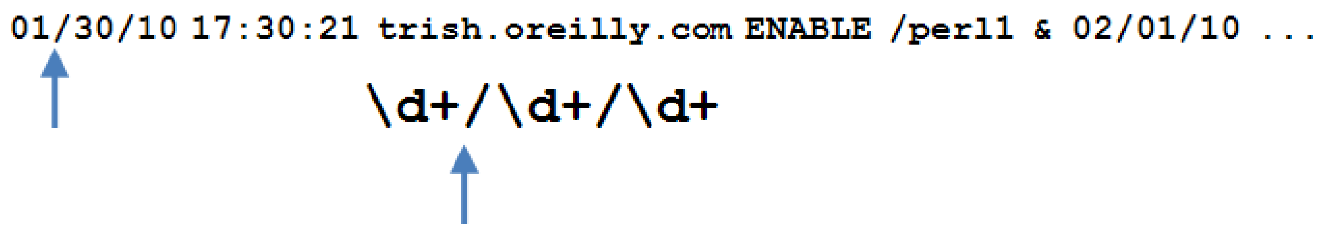
Here we have a sample log format containing two entries separated by an ampersand (**&**). Each entry consists of several fields. We parse out the date by looking for three sets of one or more digits separated by slashes. In English,

/g tells our program that, "once we've matched everything in the regular expression, see if we can match it again. If the regex doesn't match at the new location, advance the point at which we're testing for a match one character at a time until we get a match or we run out of input."

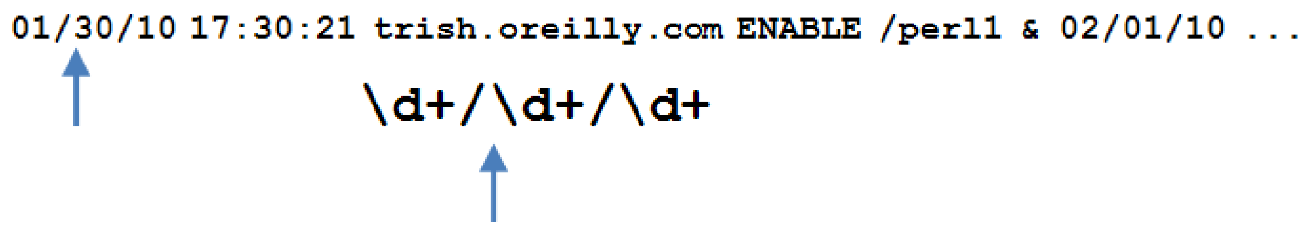
A visual aid may help here. Think of two pointers being advanced as a regex is matched against an input. Before any matching has taken place, the pointers are in this state:


01/30/10 17:30:21 trish.oreilly.com ENABLE /perl1 & 02/01/10 ...
\d+/\d+/\d+

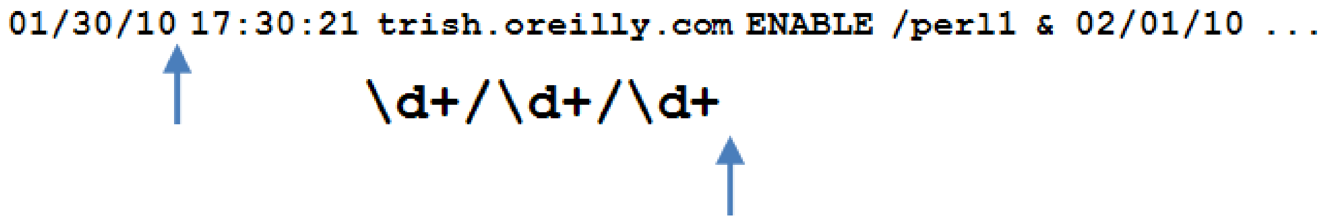
After matching the first **\d+**, the pointers are in this state:


01/30/10 17:30:21 trish.oreilly.com ENABLE /perl1 & 02/01/10 ...
\d+/\d+/\d+

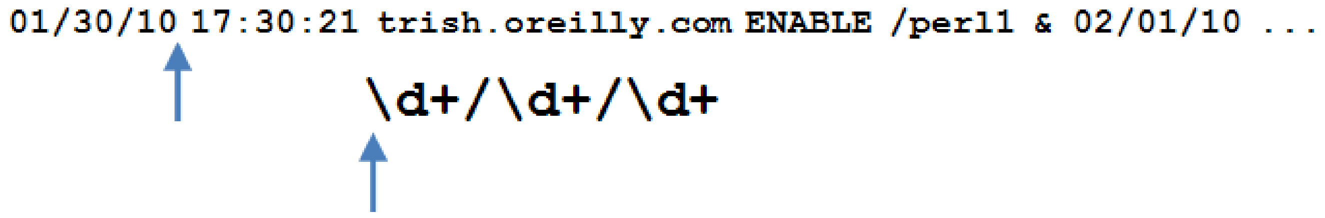
Now the input no longer satisfies the **\d+** part of the regex, so the regex engine looks at what comes after the regex pointer, sees a **/** (forward slash) required, looks at what comes into the input pointer, sees there is a **/** there, and advances both pointers:


01/30/10 17:30:21 trish.oreilly.com ENABLE /perl1 & 02/01/10 ...
\d+/\d+/\d+

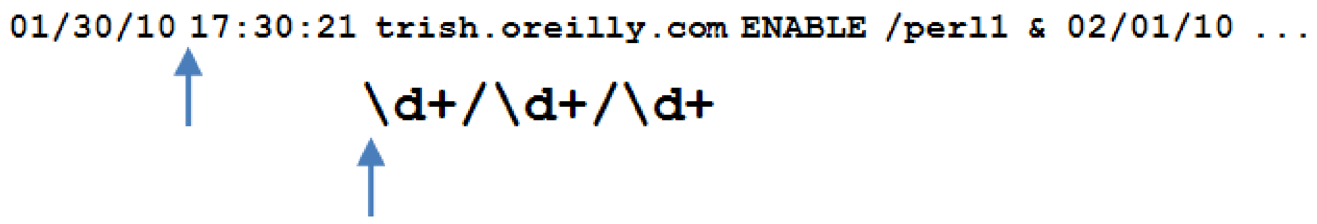
This keeps going until we reach the end of the regex:


01/30/10 17:30:21 trish.oreilly.com ENABLE /perl1 & 02/01/10 ...
\d+/\d+/\d+

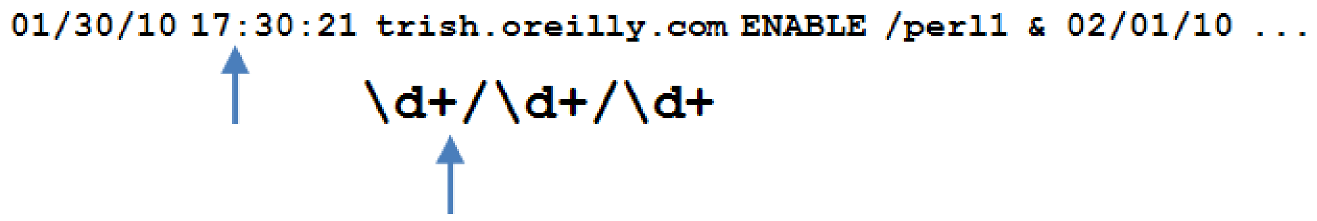
Except for the **/g** modifier, we would be finished. But **/g** instructs us to bring the regex pointer back to the beginning of the regex and to keep going:


01/30/10 17:30:21 trish.oreilly.com ENABLE /perl1 & 02/01/10 ...
\d+/\d+/\d+

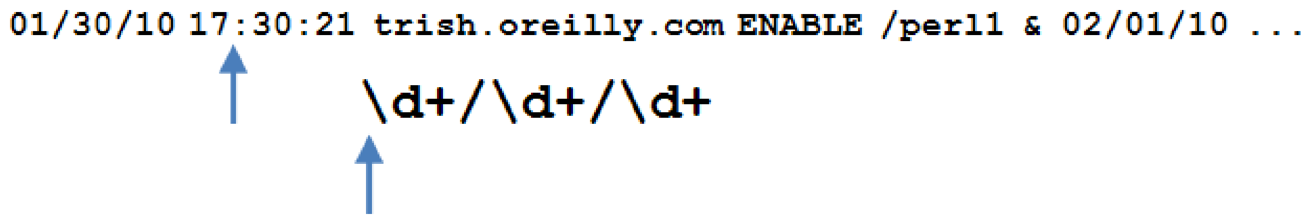
The input pointer doesn't precede a digit—it's pointing at a space—so we advance it until we find something that satisfies that part of the regex:


01/30/10 17:30:21 trish.oreilly.com ENABLE /perl1 & 02/01/10 ...
\d+/\d+/\d+

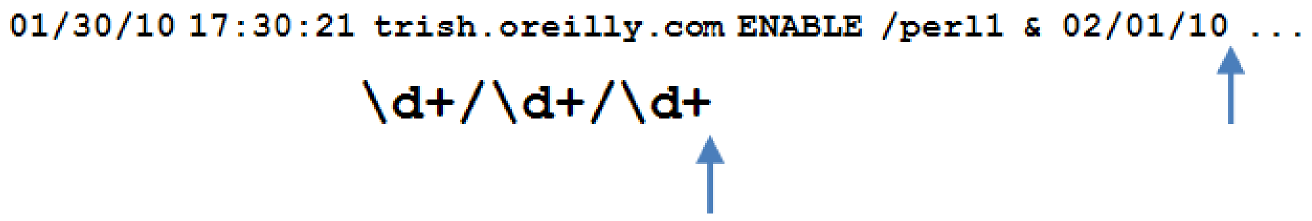
But after reading in those digits and getting to this point:


01/30/10 17:30:21 trish.oreilly.com ENABLE /perl1 & 02/01/10 ...
\d+/\d+/\d+

...the regex is still not satisfied; it requires a **/**, but the input has a **:** (colon), so the regex engine *backtracks*:

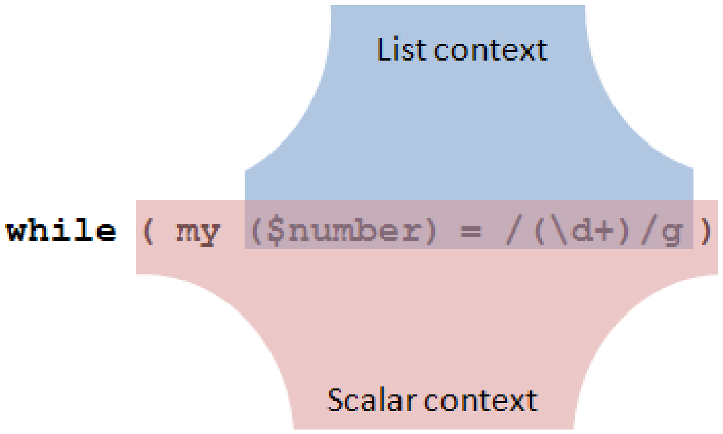

01/30/10 17:30:21 trish.oreilly.com ENABLE /perl1 & 02/01/10 ...
\d+/\d+/\d+

...and tries again. Eventually it will wind up here:


01/30/10 17:30:21 trish.oreilly.com ENABLE /perl1 & 02/01/10 ...
\d+/\d+/\d+

...having achieved another successful match of the entire regex. The rest of the input has been omitted from the image for clarity, but in short, the regex engine obeys the **/g** modifier, resets its pointer and keeps looking for a match, will match part of the regex, but not the whole thing, and so there are no more matches. But it has successfully matched *twice*, so in a scalar context the regex will return **1**, and in a list context the match will return either the list **(1)** (if there were no capturing parentheses), or the list of all the **$1**, **$2**, **$3**, and so on, that it captured on each iteration of the regex.

Now you can see why the earlier revision of **g_match_number.pl** went into an infinite loop! Although the condition of the **while** statement imposes scalar context, the assignment to the single-element list **($number)** imposes *list* context, and because that is the innermost part of the expression, that's the context in which the match is evaluated:



In a list context, the match returns the list of all the captures. We only save one of them; that capture goes into **$number**. Then the list assignment is evaluated in the scalar context imposed by the **while** statement. There's one element in the list, so the assignment is true, and the loop is executed. Now we come back around to test the condition again, but the global match is not being called in scalar context, so the match is not present in the middle of any iteration through the input—the match finished with that in one round when the match was called in list context. So the match is ready to run again from the beginning of the input, and that's what the match does.

This excursion through the fine details of regular expression operation may feel a bit tedious, but stick with it—you need to understand the operation of matches in list and scalar contexts with and without the **/g** modifier thoroughly in order to write your own regexes. For many programmers regular expressions are a total mystery that they never understand well and only get to work by accident by copying and pasting old pieces of code and hacking at them until they seem to work. We accept no mysteries in this course! We're better than that! Let's keep going. Modify **g_match_number2.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $line = "01/30/10 17:30:21 trish.oreilly.com ENABLE /perl1 & 02/01/10 09:10:00 tim.o
reilly.com ENROLL /perl1";

~~my ($date1, $date2) = ( $line =~ m!(\d+/\d+/\d+)!g );~~
~~print "Date: $date1; date: $date2\n";~~
my ($date1, $time1, $course1, $date2, $time2, $course2) = ( $line =~ m!([\d/]+)\s+([\d:
]+)[^/]*/(\S*)!g );
print "Date: $date1; time: $time1; course: $course1; Date: $date2; time: $time2; course
: $course2\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./g_match_number2.pl
Date: 01/30/10; time: 17:30:21; course: perl1; Date: 02/01/10; time: 09:10:00; course:
perl1
```

Here you can see the match operator return two sets of three captures; that is, **($1, $2, $3)**, twice. This example is somewhat contrived because we'll rarely have exactly two records in a single input to process this way; it's much more common to want to iterate through whatever number of records there might be. We'll try that more common usage out now. Modify **g_match_number2.pl** in your **/perl3** folder as follows:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = "09/26/10 18:23:17 trish.oreilly.com ENABLE /perl2 & 09/27/10 09:16:23 tim.oreilly
.com ENROLL /perl2";

while ( m!([\d/]+)\s+([\d:]+)[^/]*/(\S*)!g )
{
  my ($date, $time, $course) = ($1, $2, $3);
  print "Date; $date; time: $time; course: $course\n";
}
~~my ($date1, $time1, $course1, $date2, $time2, $course2) = ( $line =~ m!([\d/]+)\s+([\d:
]+)[^/]*/(\S*)!g );~~
~~print "Date: $date1; time: $time1; course: $course1; Date: $date2; time: $time2; course
: $course2\n";~~
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./g_match_number2.pl
Date; 09/26/10; time: 18:23:17; course: perl2
Date; 09/27/10; time: 09:16:23; course: perl2
```

We are matching on **$_**, which keeps the code less cluttered. Usually you will set **$_** implicitly through a readline operator like **<$fh>** or **<>**.

Here's a cool use of matching in a list context where we don't know how many tokens we're going to match, we just want to match as many as there are. Create **g_match_list.pl**:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = join ' ', split /\n/, <<'END_OF_TEXT';
This text is embedded in the program in multiple
lines, but our program splits the heredoc into
a list of lines (which don't contain the newline
characters themselves, because those were what was
split on), and then joins them with single spaces
to form a single long string.  All in one expression, too!
END_OF_TEXT

my @words = /(\w+)/g;
print " - $_ - \n" for @words;
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$./g_match_list.pl
- This -
- text -
- is -
- embedded -
- in -
- the -
- program -
- in -
. . .
(etc.)
```

See if you can improve the regular expression so it does a better job of matching what you think are "words" in the input.

Global Substitution: Scalar and List Contexts

Now that we've covered the behavior of global matching, let's make sure you understand global substitution. Create **g_subs.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

while ( <DATA> )
{
  print "Before: $_";
  while ( my $result = s/(['"])(\w+)\1/{$2}/g )
  {
    print "  Made $result change(s)\n";
  }
  print "After: $_";
}

__END__
In this text, "some" words are "quoted", like this: 'quoted'.
That means they're surrounded by either 'single' quotes
or "double" quotes.  But an "apostrophe" in a word like "don't"
doesn't "count".
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./g_subs.pl
Before: In this text, "some" words are "quoted", like this: 'quoted'.
  Made 3 change(s)
After: In this text, {some} words are {quoted}, like this: {quoted}.
Before: That means they're surrounded by either 'single' quotes
  Made 1 change(s)
After: That means they're surrounded by either {single} quotes
Before: or "double" quotes.  But an "apostrophe" in a word like "don't"
  Made 2 change(s)
After: or {double} quotes.  But an {apostrophe} in a word like "don't"
Before: doesn't "count".
  Made 1 change(s)
After: doesn't {count}.
```

Unlike the match operator, the substitution operator with **/g** in a scalar context does *not* act as an iterator; it does all of the substitutions and then returns the number of substitutions made.

And in a list context, it's the same. Modify **g_subs.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

while ( <DATA> )
{
  print "Before: $_";
  ~~while ( my $result = s/(['"])(\w+)\1/{$2}/g )~~
~~  {~~
~~    print "  Made $result change(s)\n";~~
~~  }~~
  my @results = s/(['"])(\w+)\1/{$2}/g;
  print "  Made @results change(s)\n";
  print "After: $_";
}

__END__
In this text, "some" words are "quoted," like this: 'quoted.'
That means they're surrounded by either 'single' quotation marks
or "double" quotation marks.  But an "apostrophe" in a word like "don't"
doesn't "count." Get it?
```

Check Syntax and run it. You'll get exactly the same output as before. In a list context, we see that a substitution with **/g** returns a list containing one element, which is the number of changes made. But evaluating a substitution in list context isn't particularly useful.

Nongreedy Matching

"Greed is good"
--Gordon Gecko, *Wall Street*

Every quantifier that we've met so far in our exploration of regular expressions (*, +, ?, and {*m*,*n*} is *greedy:* it'll match as much of the input as it can. That's useful, but sometimes we want the opposite: to match as *little* as possible. (We need to be careful when we do this though. To "match as little as possible" can result in "match nothing" and that may not be the outcome we want.)

The syntax for nongreedy quantifiers is simple and logical; just add a question mark to the corresponding greedy quantifier. Let's use, for example, **\d** as an atom to quantify:

Greedy Regex	**Meaning**	**Nongreedy Regex**	**Meaning**
\d*	Match zero or more digits, preferably as many as possible	\d*?	Match zero or more digits, zero if possible
\d+	Match one or more digits, preferably as many as possible	\d+?	Match one or more digits, one if possible
\d?	Match zero or one digits, preferably one	\d??	Match zero or one digits, preferably zero
\d{3,7}	Match 3 to 7 digits, preferably 7	\d{3,7}?	Match 3 to 7 digits, preferably 3

The difference between greedy and nongreedy matching can be illustrated using this one-liner:

INTERACTIVE SESSION:

```
cold:~/perl3$ perl -le '$_ = "First pig second pig third pig last pig"; s/(.*)pig/$1cow
/; print'
First pig second pig third pig last cow
cold:~/perl3$ perl -le '$_ = "First pig second pig third pig last pig"; s/(.*?)pig/$1co
w/; print'
First cow second pig third pig last pig
```

See how **.*** matched as *many* characters as possible whereas **.*?** matched as *few* characters as possible? Now let's see that at work in an example. Create **nongreedy.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = "abc123def 456ghi7";

print "Greedy:    ";
print for /(c.*[def])/;
print "\nNongreedy: ";
print for /(c.*?[def])/;
print "\n";
```

Check Syntax and run it:

INTERACTIVE SESSION:

```
 cold:~/perl3$ ./nongreedy.pl
Greedy:    c123def
Nongreedy: c123d
```

We use the result of a match containing a capture in a list context to print the match in a really succinct idiom. So why did that happen? In the first expression, the **.*** will match as many characters as possible, which initially means that it will match all the way to the end of the input (because it's being *greedy*); but then the regex engine will look at what comes next in the regex and see that it needs a **d**, **e**, or **f**. There isn't one though, because the regex has arrived at the end of the string, so it'll backtrack, undoing one character at a time from the greedy match of **.*** until it finds its input pointer in front of a **d**, **e**, or **f**.

In the second expression, the **.*?** will match as *few* characters as possible, which initially means that it matches *zero* characters. But then the regex engine sees that it needs a **d**, **e**, or **f**, and it looks at the next character in the input and it's a **1**, which won't do, so it allows the **.*?** to match one character, and tries again; but now the next character is a **2**, which still won't do, so our regex allows the **.*?** to match two characters; but now the next character is a **3**, which still won't do, so our regex allows the **.*?** to match three characters, and now the next character is a **d**, which *will* do, and so it has a match.

That little description of how the regex engine behaves for a greedy quantifier versus a nongreedy quantifier contains all you need to know about how the two types of quantifier work. Just remember that a greedy quantifier means that the regex engine will match as many things as possible and then backtrack as necessary to match the rest of the regex, while a nongreedy quantifier means that the regex engine will match as few things as possible and then match more things as necessary to match the rest of the regex.

A nongreedy quantifier can often save you from having to create a more complicated regex. For example, suppose you were parsing some simple HTML. Create **match_html.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = join '', <DATA>;
s/\n//g;
length and print " - $_ -\n" for split /<(.+?)>/;

__END__
<HTML>
<HEAD><TITLE>Lesson 4</TITLE></HEAD>
<BODY>
<H1>Lesson 4: Global Matches</H1>
<P>Here we will learn about the <B>/g</B> modifier.
</P>
</BODY>
</HTML>
```

Check Syntax and run it:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./match_html.pl
- HTML -
- HEAD -
- TITLE -
- Lesson 4 -
- /TITLE -
- /HEAD -
- BODY -
- H1 -
- Lesson 4: Global Matches -
- /H1 -
- P -
- Here we will learn about the -
- B -
- /g -
- /B -
- modifier. -
- /P -
- /BODY -
- /HTML -
cold:~/perl3$
```

That's a useful start to parsing HTML, but suppose we had used greedy matching instead. Modify the program as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = join '', <DATA>;
s/\n//g;
length and print " - $_ -\n" for ~~split /<(.+?)>/;~~ split /<(.+)>/;

__END__
<HTML>
<HEAD><TITLE>Lesson 4</TITLE></HEAD>
<BODY>
<H1>Lesson 4: Global Matches</H1>
<P>Here we will learn about the <B>/g</B> modifier.
</P>
</BODY>
</HTML>
```

Check Syntax and run it. What is going on with the output? Think about why it looks that way.

You can fix that problem there without resorting to nongreedy quantifiers if you want. This next version of the program works the same way as the one with the nongreedy quantifier. Modify your code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = join '', <DATA>;
s/\n//g;
length and print " - $_ -\n" for ~~split /<(.+)>/;~~ split /<([^>]+)>/;

__END__
<HTML>
<HEAD><TITLE>Lesson 4</TITLE></HEAD>
<BODY>
<H1>Lesson 4: Global Matches</H1>
<P>Here we will learn about the <B>/g</B> modifier.
</P>
</BODY>
</HTML>
```

But this approach may become difficult to implement when the token that comes after the quantified atom is more than one character long. So nongreedy quantifiers are definitely worth learning!

Here's another example you can try as a one-liner, taking output from the **curl** program that fetches a web page:

INTERACTIVE SESSION:

```
cold:~/perl3$ curl -s http://www.oreillyschool.com/certificate-programs | perl -nle '/<
li>.*?>(.+?)</ and print $1'
```

Try it! It lists the certificate programs available at the O'Reilly School of Technology (At least, until they change the format of that page... this is one of the problems of web page scraping. At that point, we'll modify the one-liner!):

OBSERVE:

```
C#.NET Programming Certificate
Client-Side Programming Certificate
Database Administration Certificate
Java Programming Certificate
Linux Systems Administration Certificate
Open-Source Programming Certificate
Perl Programming Certificate
PHP/SQL Programming Certificate
Python Programming Certificate
Web Programming Certificate
```

Notice that a few items that aren't courses were output. (This is also one of the problems of parsing HTML: it's a language describing presentation, not semantics.) See if you can modify the regular expression so that it shows only the courses.

WARNING In general, this is *not* the way to parse HTML, because HTML may contain all kinds of constructions that defeat a basic regex, for example, angle brackets inside comments. The proper way to parse any kind of HTML is with a module like **HTML::Parser**. But if you control the HTML that will be your program's input and can guarantee that it will always be formatted in a way that your regexes can parse, you can ignore this restriction, because your input is technically not "HTML." It's "a custom format that is equivalent to a subset of HTML".

Wow. That was one long lesson, but an really important one. Now your regular expression expertise has officially reached an advanced level! Good work! Keep it up and see you in the next lesson...

Once you finish the lesson, go back to the syllabus to complete the homework.

grep() and map()

Lesson Objectives

When you complete this lesson, you will be able to:

- use the grep() function to select the elements from a list that match some criteria.
- use the map() fucntion.

"The human animal differs from the lesser primates in his passion for lists."
-H. Allen Smith

Welcome to a new lesson, where you'll use the Perl skills you've learned so far, and incorporate some new ones to manipulate lists at a whole new level! In this lesson we'll add two important functions to your tool chest: **grep()** and **map()**. Both of these functions take a list as input, and provide a list as output. These tools will allow you to transform lists using simple expressions rather than whole blocks of code. Sounds great, right? Let's get going!

grep()

If you've used the Unix program called **grep**—and we built a simplified version of it in Perl 2—you may be tempted to think that the **grep()** function in Perl does the same thing. Well, it's kind of related, except that Perl's **grep()** really has nothing to do with regular expressions.

The **grep()** function is used to select the elements of a list that match some criteria. And it lets you write less code than you would building a loop and a temporary array to serve that same purpose. Let's try executing that task without using **grep()** first. Create **tracks.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $MINUTE = 60;   # Seconds therein

my %length;

while ( <DATA> )
{
  next unless /\A\d+\.\s+"(.*?)".*\s(\d+):(\d+)/;
  my ($title, $min, $sec) = ($1, $2, $3);
  $length{$title} = $min * $MINUTE + $sec;
}

my $limit = shift || 3;
my @long_tracks;
for my $title ( sort keys %length)
{
  push @long_tracks, $title if $length{$title} > $limit * $MINUTE;
}
report( $limit, @long_tracks );

sub report
{
  my $limit = shift;

  print "Tracks over $limit minutes long:\n";
  print "$_\n" for @_;
}

__END__
Sgt. Pepper's Lonely Hearts Club Band (Track listing from Wikipedia)
Side one
No. Title Length
1. "Sgt. Pepper's Lonely Hearts Club Band"   2:00
2. "With a Little Help from My Friends"   2:43
3. "Lucy in the Sky with Diamonds"   3:26
4. "Getting Better"   2:47
5. "Fixing a Hole"   2:35
6. "She's Leaving Home"   3:33
7. "Being for the Benefit of Mr. Kite!"   2:35
Side two
No. Title Length
1. "Within You Without You" (George Harrison) 5:05
2. "When I'm Sixty-Four"   2:37
3. "Lovely Rita"   2:41
4. "Good Morning Good Morning"   2:42
5. "Sgt. Pepper's Lonely Hearts Club Band (Reprise)"   1:19
6. "A Day in the Life"   5:04
```

Our program **tracks.pl** takes an optional argument to report the minimum length of a track in minutes. Check Syntax and run it:

INTERACTIVE SESSION:

```
code:~$ cd perl3
cold:~/perl3$ ./tracks.pl
Tracks over 3 minutes long:
A Day in the Life
Lucy in the Sky with Diamonds
She's Leaving Home
Within You Without You
cold:~/perl3$ ./tracks.pl 4
Tracks over 4 minutes long:
A Day in the Life
Within You Without You
```

The goal in this program was to construct the array **@long_tracks** and have it contain the titles we wanted. We could have reported them one at a time in the loop, but imagine that we need that array for something else later in the program. Now let's change the program and try something else. Modify your code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $MINUTE = 60;   # Seconds therein

my %length;

while ( <DATA> )
{
  next unless /\A\d+\.\s+"(.*?)".*\s(\d+):(\d+)/;
  my ($title, $min, $sec) = ($1, $2, $3);
  $length{$title} = $min * $MINUTE + $sec;
}

my $limit = shift || 3;
my @long_tracks = grep { $length{$_} > $limit * $MINUTE } sort keys %length;
~~for my $title ( sort keys %length)~~
~~{~~
~~  push @long_tracks, $title if $length{$title} > $limit * $MINUTE;~~
~~}~~
report( $limit, @long_tracks );

sub report
{
  my $limit = shift;

  print "Tracks over $limit minutes long:\n";
  print "$_\n" for @_;
}

__END__
Sgt. Pepper's Lonely Hearts Club Band (Track listing from Wikipedia)
Side one
No. Title Length
1. "Sgt. Pepper's Lonely Hearts Club Band"   2:00
2. "With a Little Help from My Friends"   2:43
3. "Lucy in the Sky with Diamonds"   3:26
4. "Getting Better"   2:47
5. "Fixing a Hole"   2:35
6. "She's Leaving Home"   3:33
7. "Being for the Benefit of Mr. Kite!"   2:35
Side two
No. Title Length
1. "Within You Without You" (George Harrison) 5:05
2. "When I'm Sixty-Four"   2:37
3. "Lovely Rita"   2:41
4. "Good Morning Good Morning"   2:42
5. "Sgt. Pepper's Lonely Hearts Club Band (Reprise)"   1:19
6. "A Day in the Life"   5:04
```

Check Syntax and run it. You'll see exactly the same output as before.

So how does it work? The **grep()** function's general form looks like this:

OBSERVE:

```
RESULT_LIST = grep { EXPRESSION } LIST
```

Or, in our example:

OBSERVE:

```
my @long_tracks = grep { $length{$_} > $limit * $MINUTE } sort keys %length
```

Your chosen **EXPRESSION** goes inside the curly braces; Perl evaluates it in Boolean context once for each element of the input list **LIST**. **$_** is set equal to each element in turn (this is much like the postfixed foreach statement). If the result of **EXPRESSION** is true, the element being evaluated is added to **RESULT_LIST**; if not, it is ignored.

So **grep()** works as a filter; any elements of the input that pass the test in **EXPRESSION** go through to the result.

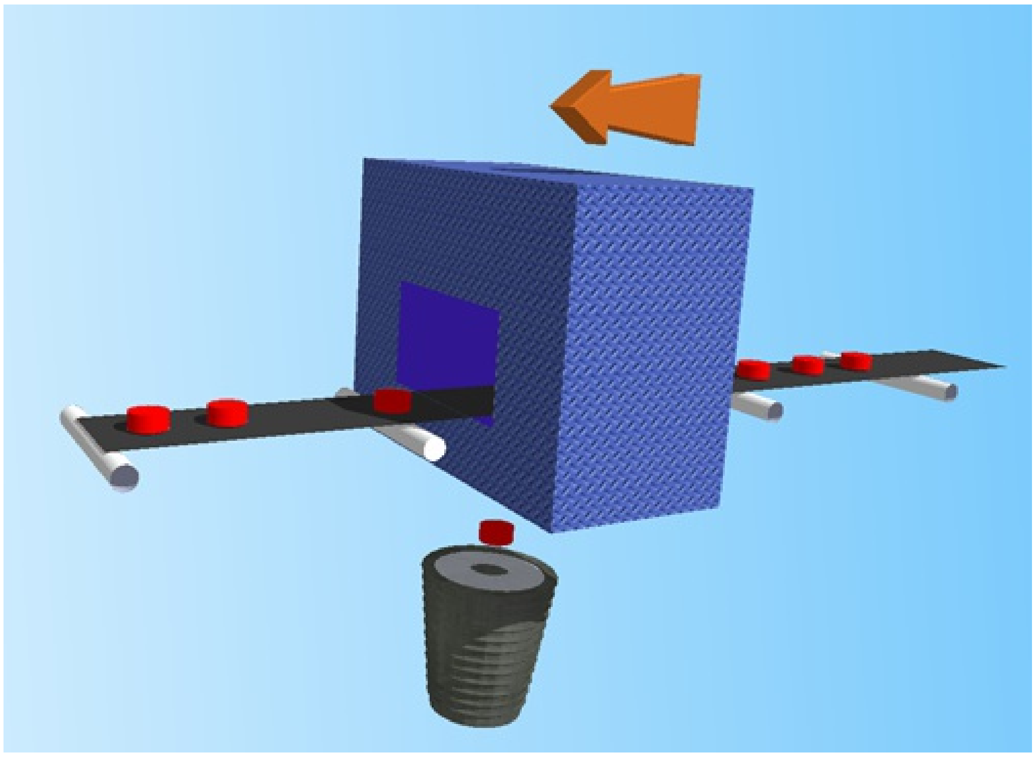
The braces delimit a block of code; you can have multiple statements within them (separated by semicolons). The value of the last statement or expression in the block of code will be used for the test. So this description of **grep()** is equally valid:

OBSERVE:

```
RESULT_LIST = grep { CODE } LIST
```

You could, for example, replace the code **{ $length{$_} > $limit * $MINUTE }** in the program above with **{ my $len = $length{$_}; $len / $MINUTE > $limit }** and it would perform exactly the same function.

I like to think of a conveyor belt metaphor for **grep()**:



As each element of the input list passes (from right to left) through the expression/code block, it either makes it out the other end into the result, or is rejected.

Here's an example of a common use of **grep()**. The **readdir()** function for returning a list of filenames in a directory corresponding to a directory handle opened by **opendir()** always returns the two directories "." and ".." as part of the list, because those entries are always in every directory. Since they correspond to the current directory and parent directory respectively, it is almost never useful to process those elements. Let's take a look. Create **directory.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

opendir my $dh, '.' or die "Can't open . $!\n";
my @files = readdir $dh;
print "$_\n" for sort @files;
```

Check Syntax and run it. Near the top, the lines "." and ".." are printed. Those elements are almost certainly going to get in the way, regardless of the purpose we put **@files** to in our program.

Here's how we usually solve this problem. Modify **directory.pl** as follows:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

opendir my $dh, '.' or die "Can't open . $!\n";
my @files = grep { ! /\A\.\.?\z/ } readdir $dh;
print "$_\n" for sort @files;
```

Check Syntax and run it. The "." and ".." lines are no longer printed. Because **$_** is set in the block to the current element being considered, we can include a match expression that is implicitly bound to **$_**, as we have done here. Do you understand the regex? It means:

!	Negation
\A	Must start with
\.	Literal period
\.?	Followed by zero or one literal periods
\z	Must end here

We'll be using regexes of increasing complexity in our programs from now on. If you don't understand them, review the corresponding material from earlier lessons, in the **Intermediate Perl** course, or ask your instructor.

map()

map() is like **grep()**, only it's even cooler! It has the same general form:

OBSERVE:

```
RESULT_LIST = map { EXPRESSION or CODE } LIST
```

And just like with **grep()**, with **map()**the block is executed once for each element of the input **LIST**, with **$_** being set to each element in turn. But in this case, the result of the block is not used to decide whether to pass the element through, but is used *as* the output element!

Let's try some examples. Create **miles.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl -C2
use strict;
use warnings;

my @squares = map { $_ ** 2 } 1 .. 10;
print "@squares\n";

my %main_tank    = ( humvee => 30, voyager => 0, moped => 1 );
my %reserve_tank = (humvee => 0, voyager => 400, moped => 0.1 );
my %mpg          = (humvee => 9, voyager => 25_000/1_100, moped => 80 );
my @vehicles     = qw(humvee voyager moped);
my @miles_left = map { $main_tank{$_} > 0
                        ? $mpg{$_} * $main_tank{$_}
                        : $mpg{$_} * $reserve_tank{$_}
                      } @vehicles;
print "@miles_left\n";

my @x_axis = map { $_ * 0.3 } -10 .. 10;
my @y_axis = map { cos $_ } @x_axis;
print map { '*' x (40 + 30 * $_), "\n" } @y_axis;

my %char = ( eacute => 233, cedilla => 231, agrave => 224 );
$_ = chr for values %char;
my @words = qw(caf_eacute_ gar_cedilla_on d_eacute_j_agrave_);
print map { s/_(.*?)_/$char{$1}/g; "$_\n" } @words;
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./miles.pl
1 4 9 16 25 36 49 64 81 100
270 9090.90909090909 80
**********
************
*****************
************************
*********************************
******************************************
**************************************************
**********************************************************
****************************************************************
********************************************************************
**********************************************************************
********************************************************************
****************************************************************
**********************************************************
**************************************************
******************************************
*********************************
************************
*****************
************
**********
café
garçon
déjà
```

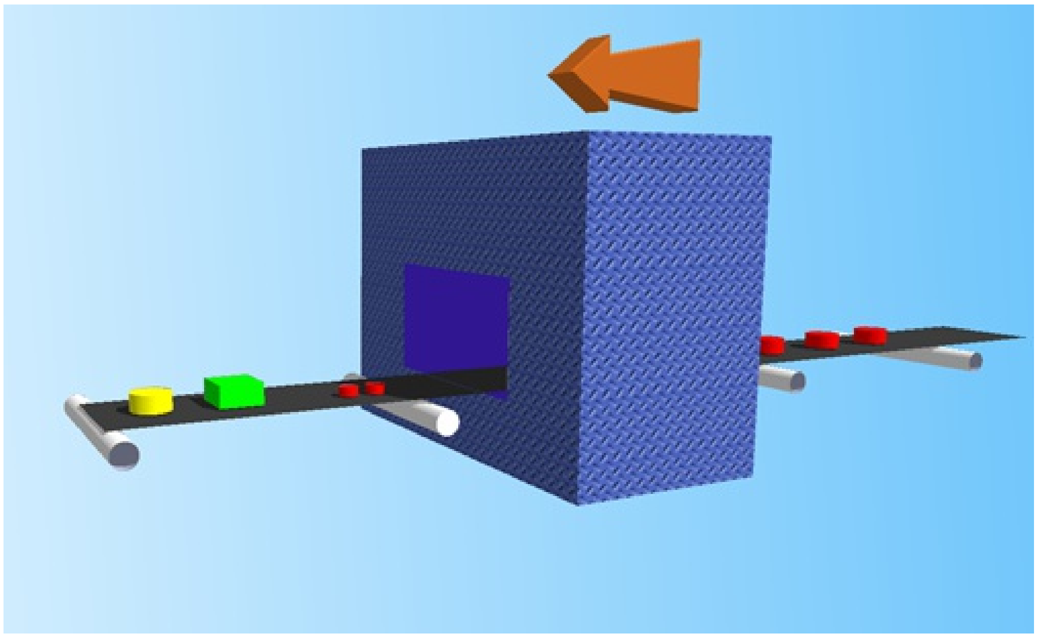
If you didn't get the accents on the last three lines, make sure you have the **-C2** flag on the shebang line (that's the first line of a Perl program, by the way—it begins with #!). This is not usually necessary and is mostly required here due to the complex environment that brings the terminal window to you through a Java applet.

If you're curious about how the cosine wave pattern of asterisks was made, check out perldoc perlop for the explanation of the **x** (repetition) operator in scalar context.

In our code there are several examples of the use of the **map()** operator. Go over each one carefully to make sure you understand them; the code in each case is brief and could easily be overlooked.

The **values()** function returns a list of *aliases* to the actual values, meaning that if I change those values, I change the actual values in the hash itself. Go to perldoc -f values to read about that feature.

In the conveyor belt metaphor, the machine that the inputs pass through gets to replace each input with whatever it wants, usually the result of transforming the input in some way:



This is important: the expression (or last statement) in the block is evaluated in *list* context, so it can return *any number* of elements. That means that the number of elements in the output doesn't have to be the same as the number of elements in the input: it can be fewer (if the block evaluates to an empty list at least some of the time) or more (if the block evaluates to a list containing two or more elements at least some of the time).

For example, we may create a hash merely to test for *set existence*, because we want to know whether some string is a key in the hash. We usually make all the values in the hash **1** in this case. If the hash is created from a literal list of keys or an array of keys, the code is longer than it has to be. Create **map_hash.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %marsupial = (koala => 1, kangaroo => 1, possum => 1, wombat => 1);

chomp( my @amphibians = <DATA> );

my %amphibian;
$amphibian{$_} = 1 for @amphibians;

print "Marsupials: ", join( ' ', sort keys %marsupial ), "\n";
print "Amphibians: ", join( ' ', sort keys %amphibian ), "\n";
__END__
frog
toad
salamander
newt
caecilian
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./map_hash.pl
Marsupials: kangaroo koala possum wombat
Amphibians: caecilian frog newt salamander toad
```

I'm always sneaking something extra into these examples! See how this program **chomp()**s every member of an array in one pass? Take a look at perldoc -f chomp for further explanation.

That program shows two ways of setting a hash: a literal list and setting from an array of keys. In each case, every value in the hash is **1**. Now modify **map_hash.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

~~my %marsupial = (koala => 1, kangaroo => 1, possum => 1, wombat => 1);~~
my %marsupial = map { ($_, 1) } qw(koala kangaroo possum wombat);
chomp( my @amphibians = <DATA> );

my %amphibian = map { ($_, 1) } @amphibians;
~~$amphibian{$_} = 1 for @amphibians;~~

print "Marsupials: ", join( ' ', sort keys %marsupial ), "\n";
print "Amphibians: ", join( ' ', sort keys %amphibian ), "\n";
__END__
frog
toad
salamander
newt
caecilian
```

Check Syntax and run it. You get exactly the same output as before. The result of each **map()** block is a two-element list (I put parentheses around them to make this clearer, but you can actually leave them off): the element from the input list (the desired key) and the literal **1** (the value).

There is always another way to get the same results you would from **grep()** and **map()**; but using them may keep your code shorter and more readable. Remember, Perl's motto is, *"There's More Than One Way To Do It"*—in fact, that's the subtitle of the book "Programming Perl" published by O'Reilly!

A word of warning: sometimes programmers get carried away with the succinctness of **map()** and use it where they should be using **foreach** statements. You can tell that this is happening is when they are ignoring the output list. For instance:

OBSERVE:

```
my $total = 0;
map { $total += $length{$_} } grep { $length{$_} > $limit * $MINUTE } keys %length;
```

We call that using **map()** in a *void context*. It may *work,* but it's not a good practice; it makes the code harder to understand by another Perl programmer, who will wonder why there is no output list. And it's not any shorter than the preferred code either:

OBSERVE:

```
my $total = 0;
$total += $length{$_} for grep { $length{$_} > $limit * $MINUTE } keys %length;
```

It's not uncommon to see **grep()** and **map()** blocks that are several lines long. Just make sure to indent them thoughtfully to help the reader if you're writing code like that. Later on in this course we'll join a **map()**, a **grep()**, and another **map()** together in a chain! There's no reason that one conveyor belt can't feed into another, after all.

We are making really good progress. Keep it going in the next lesson! See you there...

Once you finish the lesson, go back to the syllabus to complete the homework.

Regular Expressions: More Match and Substitution Modifiers

Lesson Objectives

When you complete this lesson, you will be able to:

- use the /m Modifier.
- use the /s Modifier.
- use the /x Modifier.
- use the /e Modifier.

Welcome back! In this lesson, we'll cover a few more modifiers that you can place on the match (**m//**) and substitution (**s///**) operators.

Let's dive right in!

The /m Modifier

I'd like to introduce two new anchors to you: **^** and **$**. The **/m** modifier allows you to use those anchors to represent the beginning and end of a line, respectively. Let's try a quick example. Go to your editor and in your **/perl3** folder, create **multimatch.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = join '', <DATA>;

print "Words not followed by punctuation at the end of the line:\n";
print "\t$_\n" for /(\w+)$/mg;

print "Second words of lines started by the word 'Now':\n";
print "\t$_\n" for /^Now\s+(\w+)/mg;

print "Words followed by a period at the end of the line:\n";
print "\t$_\n" for /(\w+)\.$/mg;

__END__
Now is the winter of our discontent
Made glorious summer by this sun of York;
And all the clouds that lour'd upon our house
In the deep bosom of the ocean buried.
Now are our brows bound with victorious wreaths;
Our bruised arms hung up for monuments;
Our stern alarums changed to merry meetings,
Our dreadful marches to delightful measures.
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
code:~$ cd perl3
cold:~/perl3$ ./multimatch.pl
Words not followed by punctuation at the end of the line:
        discontent
        house
Second words of lines started by the word 'Now':
        is
        are
Words followed by a period at the end of the line:
        buried
        measures
```

Let's look at these anchors in detail to see how they work. First, we'll make a small modification to the **multimatch.pl** program, and remove the **/m** modifier:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = join '', <DATA>;

print "Words not followed by punctuation at the end of the line:\n";
print "\t$_\n" for /(\w+)$/~~m~~g;

print "Second words of lines started by the word 'Now':\n";
print "\t$_\n" for /^Now\s+(\w+)/~~m~~g;

print "Words followed by a period at the end of the line:\n";
print "\t$_\n" for /(\w+)\.$/~~m~~g;

__END__
Now is the winter of our discontent
Made glorious summer by this sun of York;
And all the clouds that lour'd upon our house
In the deep bosom of the ocean buried.
Now are our brows bound with victorious wreaths;
Our bruised arms hung up for monuments;
Our stern alarums changed to merry meetings,
Our dreadful marches to delightful measures.
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./multimatch.pl
Words not followed by punctuation at the end of the line:
Second words of lines started by the word 'Now':
        is
Words followed by a period at the end of the line:
        measures
```

So that's how **^** and **$** behave *without* the **/m** modifier (I showed you right away because I *knew* that would be one of your first questions!): they match at the beginning and end of the *string*, not each line. To help remember what **/m** does, think of it as standing for "multiline" mode.

I can guess what you're thinking: "Now hold on a minute, Perl already has anchors for the beginning and end of the string: **\A** and **\z**, respectively. I know that Perl's motto is, 'There's more than one way to do it,' but isn't this going too far?"

Well, you're partly right. There's a lot of subtle information we're about to cover here, just go with it. If you read much Perl code written by others, you'll run into frequent use of the **^** and **$** anchors to indicate the beginning and end of strings. When Larry Wall designed Perl, he wanted to create a regex language that felt familiar to programmers. The **\A** and **\z** anchors didn't exist yet, so he used the **^** and **$** anchors, which were already established in existing regular expression tools.

But **^** and **$** are not the best choices for this syntax. The **^** symbol already has an inordinate number of meanings in a regular expression; and the **$** symbol looks like it belongs at the beginning of a scalar. Also, the **$** anchor doesn't *quite* mean "match at end of string." In fact, it means, "match at end of string *or* before an end-of-line that is followed by end of string." Try this one-liner and you'll see the difference:

INTERACTIVE SESSION:

```
cold:~/perl3$ perl -le '$_="kumquat\n"; print "\$ matches" if /t$/; print "\\z does not
 match" unless /t\z/'
$ matches
\z does not match
```

$ is equivalent to the regex subexpression **(?:\n\z|\z)**. And just for symmetry, the **\Z** anchor was introduced to mean the same thing. Extend your previous one-liner as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ perl -le '$_="kumquat\n"; print "\$ matches" if /t$/; print "\\z does not
 match" unless /t\z/; print "But \\Z *does* match" if /t\Z/'
$ matches
\z does not match
But \Z *does* match
```

At first glance, it may seem confusing to have an anchor that matches either end of string or before a newline followed by end of string. But in fact, it's actually really useful. Most programmers forget that the anchor may match before a newline, because they will often read a line into a scalar and not **chomp()** it, so the newline on the end isn't important to them. Having **$** match before that newline means **chomp()** is optional.

We are left with these best practices:

1. For matching beginning and end of string, use **\A** and **\z** respectively.
2. For matching the end of string *or* before a newline followed by end of string, use **\Z**.
3. Much code that you encounter will use **^** and **$** instead of **\A** and **\Z** respectively; that's okay.
4. If you want to match the beginning of a *line,* use the **^** anchor with the **/m** modifier.
5. If you want to match the end of a *line,* use the **$** anchor with the **/m** modifier.

Go ahead and read over the **multimatch.pl** example we started and make sure you can follow it. Note in particular how we used the **/g** flag to get a list of everything matching in the capturing parentheses.

Now let's try a Perl one-liner. Type the command below as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ perl -0ne 'print "$_\n" for /^(\w+)$/mg' *
(output will vary ...)
cold:~/perl3$ cd ..
cold:~/perl3$
```

That will print out all the lines in all the files in the current directory that consist *solely* of word characters. You'll see the line **__END__** from **multimatch.pl**. How does it work?

The **-0** command-line flag to Perl (that's a zero, not a letter "Oh," by the way) works similarly to the **join '',...** in **multimatch.pl** to read the whole data section into a single string. It tells Perl that its *input record separator* is no longer **\n**, but instead the *null character* (the character with ASCII value 0). That character doesn't occur at all in any of the files in the current directory, so the **-n** flag results in each entire file being read in to **$_** in a single string.

When we print a match, we need to enter **\n** explicitly, because we didn't use the **-l** flag to add a newline to **print()** statements. Why not? Because **-l** actually adds the *input record separator* to the end of **print()** statements. Ordinarily that would be a newline, but not this time.

The **-0** flag can do even more than I've shown here; see perldoc perlrun for details.

Note

You may have gathered by now that Perl one-liners can be really powerful. They can also be really cryptic. I am introducing them frequently now to inspire you to create them yourself to solve real problems. It may take you a while to craft a one-liner to solve a problem initially, and you may wonder if it's worth all the effort. It is. When you become familiar with command-line flags such as **-0**, regular expressions, and the match and substitution operators, you'll be able to write the one-liners you need just about as fast as you can type. This will really increase your productivity!

The /s Modifier

The **/m** modifier is similar to the **/s** modifier. You can think of **/s** as "single line" in that it changes the meaning of the character class shortcut **.** (the period symbol) so that it matches *any* character. Normally **.** is synonymous with **[^\n]**; that is, "match any character *except* the newline." When parsing text containing several lines, it is very common to want to contain a match to within a single line. Let's try an example. Create **report_parse.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = join '', <DATA>;

print "Date: Total collected\n";
while ( /^([\d-]+).* ([\d.]+)/mg )
{
  print "$1: $2\n";
}

__END__
Date Principal donors Attendance Total collected
2010-09-23 Fred Flintstone, Daffy Duck 20 1700.00
2010-09-24 Bugs Bunny, Marvin the Martian, Gossamer 170 2500.00
2010-09-25 Pepe le Pew, Atom Ant, Foghorn Leghorn 410 8100.00
2010-09-26 Snagglepuss 7 12.50
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./report_parse.pl
Date: Total collected
2010-09-23: 1700.00
2010-09-24: 2500.00
2010-09-25: 8100.00
2010-09-26: 12.50
cold:~/perl3$
```

If **.** matched any character *including* newline, that regex would have a significant problem; the greedy ***** would match everything from **Fred** up to the **12.50** in the *last* line.

Before we go any further, let me clarify a couple of things. First, when solving real problems, you're not likely to have the actual data for the problem in the file itself. But in our course, we put data in files in many examples to save you from having to create or copy separate data files. You use the **<>** operator to read from files named on the command line.

Second, this program reads multiple lines into **$_** in one pass and then uses a regex to parse **$_**, one line at a time. That's not very efficient. Imagine that instead of having the line assign to **$_**, you were handed a variable containing the multiline report from some source outside of your control.

So when is the right time to use **/s**? *Not* using **/s** is useful when parsing line-oriented input because **.** can't "run off the end of the line." But what about when you're parsing something that *isn't* line oriented, like HTML (and this is again subject to the earlier caveat about parsing HTML in general)? This is where /s can be useful. A newline can occur anywhere there's white space. But you need to parse it by matching beginning and ending tags, and there may be other intervening tags you'll want to ignore. Let's try an example. Create **parse_courses.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = join '', <DATA>;

my ($base) = /BASE\s+HREF="(.*?)"/ or die "Expecting a BASE";
while ( /<A\s+HREF="(.*?)">(?:\s*<.*?>\s*)*(.*?)\s*</gs )
{
  my ($text, $url) = ($2, $1);
  $text =~ s/\n/ /g;
  print "$text: $base$url\n";
}

__END__
<HTML>
<HEAD>
<BASE HREF="http://www.oreillyschool.com/" />
</HEAD>
<BODY>
<A HREF="courses/">
<B>Course Listing</B>
</A>
<A
HREF="courses/perl1/"><I>Introduction to
Perl</I>
</A><A HREF="courses/perl2/">Intermediate Perl</A><A HREF="courses/perl3/">Advanced Per
l
</A><A HREF="courses/perl4/"><DIV ID="Tantalize">Future Perl
course</DIV></A>
</BODY>
</HTML>
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./parse_courses.pl
Course Listing: http://www.oreillyschool.com/courses/
Introduction to Perl: http://www.oreillyschool.com/courses/perl1/
Intermediate Perl: http://www.oreillyschool.com/courses/perl2/
Advanced Perl: http://www.oreillyschool.com/courses/perl3/
Future Perl course: http://www.oreillyschool.com/courses/perl4/
```

Can you see the impact **/s** had on your code? Try taking it out:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = join '', <DATA>;

my ($base) = /BASE\s+HREF="(.*?)"/ or die "Expecting a BASE";
while ( /<A\s+HREF="(.*?)">(?:\s*<.*?>\s*)*(.*?)\s*</g~~s~~ )
{
  my ($text, $url) = ($2, $1);
  $text =~ s/\n/ /g;
  print "$text: $base$url\n";
}

__END__
<HTML>
<HEAD>
<BASE HREF="http://www.oreillyschool.com/" />
</HEAD>
<BODY>
<A HREF="courses/">
<B>Course Listing</B>
</A>
<A
HREF="courses/perl1/"><I>Introduction to
Perl</I>
</A><A HREF="courses/perl2/">Intermediate Perl</A><A HREF="courses/perl3/">Advanced Per
l
</A><A HREF="courses/perl4/"><DIV ID="Tantalize">Future Perl
course</DIV></A>
</BODY>
</HTML>
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./parse_courses.pl
Course Listing: http://www.oreillyschool.com/courses/
: http://www.oreillyschool.com/courses/perl1/
Intermediate Perl: http://www.oreillyschool.com/courses/perl2/
Advanced Perl: http://www.oreillyschool.com/courses/perl3/
: http://www.oreillyschool.com/courses/perl4/
```

Can you see why that is happening now that **.** can no longer match a newline?

Note For the sake of pretty output, we take the additional step of turning all newlines in the text portion of a URL into spaces, just as a browser would.

Our example gave us some ugly HTML! Usually it is best to parse text line by line, but when the tokens you're looking for may be split across lines, you have no choice but to parse the entire document. This also gives you the freedom to make multiple passes through the data (as we did here with an initial scan) to find the value in the <BASE> tag.

Let's take a closer look at that regex. To help with that, we'll introduce another helpful modifier.

The /x Modifier

The **/x** modifier will come as a welcome relief if you have been going cross-eyed looking at dense regular expressions to mentally parse every character in them. The **/x** modifier allows you to embed comments in your regexes, and it also makes white space insignificant. This means that you will be able to format and comment regexes just like regular program code—of course regexes *are* code.

Let's use the **/x** modifier on our previous example. Modify that program as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = join '', <DATA>;

my ($base) = /BASE\s+HREF="(.*?)"/ or die "Expecting a BASE";
~~while ( /<A\s+HREF="(.*?)">(?:\s*<.*?>\s*)*(.*?)\s*</g )~~
while ( /<A\s+                # Anchor tag
         HREF="(.*?)"         # Save the target in $1 (assume quoted)
         >                    # Assume tag ends here
         (?:\s*               # Grab any white space around...
            <.*?>             # ... any tags around the text...
            \s*               # ... with optional white space after
          )*                  # Those tags are optional
          (.*?)               # And now save text in $2
          \s*                 # Ignoring any trailing white space
          <                   # Terminated by beginning of the "A" tag
         /gsx )
{
  my ($text, $url) = ($2, $1);
  $text =~ s/\n/ /g;
  print "$text: $base$url\n";
}

__END__
<HTML>
<HEAD>
<BASE HREF="http://www.oreillyschool.com/" />
</HEAD>
<BODY>
<A HREF="courses/">
<B>Course Listing</B>
</A>
<A
HREF="courses/perl1/"><I>Introduction to
Perl</I>
</A><A HREF="courses/perl2/">Intermediate Perl</A><A HREF="courses/perl3/">Advanced Per
l
</A><A HREF="courses/perl4/"><DIV ID="Tantalize">Future Perl
course</DIV></A>
</BODY>
</HTML>
```

Check Syntax and run it. You'll get the same results as before. We haven't changed the meaning of our code at all; all we've done is add comments. Yes, comments inside a regular expression. Now you can see what the regex is doing more clearly, and spot potential bugs and improvements.

/x means that newlines, spaces, and tabs lose their literal match meaning and are instead ignored, so we can use them for indentation and breaking the regex across multiple lines. If we had had any literal spaces in the regex, we would have had to replace them, typically with **\s**. If we wanted to match just a space character, we would have had to replace them with a backslashed space character. (Newlines and tabs can be matched with **\n** and **\t** respectively.)

/x also means that a **#** sign and anything after it on the same line are ignored, so if you wanted to match a literal **#** sign, you'd have to backslash it too.

Some programmers believe that **/x** is so useful that they include it in *every* regex they write. You can use it on everything you write from now on if you like; we will use it specifically where we feel it helps.

Note You can combine the modifiers **/m** and **/s**, as **/ms** (or **/sm**), and get the effects of both at the same time. Damian Conway goes so far as to recommend in his book Perl Best Practices (O'Reilly Media, 2005) that you should put **/msx** on *every* regular expression. That's advice you may want to take. This is one of those areas where equally competent people can have differing opinions, and you can take one of several equally valid approaches depending on what works best for you.

The /e Modifier

The **/e** modifier is different from all the others that we've seen so far in that it can only be used on the *substitution* operator, because it does not affect the interpretation of the regular expression, but instead affects the interpretation of the replacement string.

The **/e** stands for *evaluate*; it causes the replacement string to be compiled and executed as Perl code. This can take some getting used to, so let's start by using a one-liner without the **/e** modifier. Type the command as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ perl -wle '$_="pot"; print; s/o/"i"/; print'
pot
p"i"t
```

Now add the **/e** modifier:

INTERACTIVE SESSION:

```
cold:~/perl3$ perl -wle '$_="pot"; print; s/o/"i"/e; print'
pot
pit
```

When the code **"i"** is evaluated as Perl, it is interpreted as a double-quoted string with a value of **i**, so that gets used as the replacement string.

Of course the real power of **/e** is exercised when we use it to substitute a string that varies according to what was matched. Create **incr.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

while ( <DATA> )
{
  s/(\d+)/$1 + 1/ge;
  print;
}

__END__
Sing a song of 6 pence, a pocket full of rye; 4 and 20 blackbirds, baked in a pie.
1969: Apollo 11 returns from the Moon after travelling 828743 nautical miles.
The first five values of Ackermann's function for n=1 are 2, 3, 5, 13, and 65533.
```

Check Syntax and run it:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./incr.pl
Sing a song of 7 pence, a pocket full of rye; 5 and 21 blackbirds, baked in a pie.
1970: Apollo 12 returns from the Moon after travelling 828744 nautical miles.
The first five values of Ackermann's function for n=2 are 3, 4, 6, 14, and 65534.
```

/e gives you the power to change a part of the text; as long as it can be computed, you can change it. Since we're bold and adventurous types, let's do change the text to something complicated! Create **eval.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

while ( <DATA> )
{
  s!(http://.*?)(\s|$)!visit($1).$2!ge or next;
  print;
}

sub visit
{
  my $url = shift;

  my $content = `curl -s $url 2>&1`;
  $? and return "[$url: Couldn't fetch]";
  return "[$url: " . length($content) . " bytes]";
}

__END__
Today I visited http://www.oreillyschool.com/ and logged in to
my courses, then I hopped over to http://www.cnn.com/ for some general news,
and http://www.perlbuzz.com/ for the latest cool news about Perl.
Then I tried the address http://xyz.liklkvsj.xx/ that someone I didn't
know emailed me about, but I couldn't reach it.
```

Check Syntax and run it:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./eval.pl
Today I visited [http://www.oreillyschool.com/: 11137 bytes] and logged in to
my courses, then I hopped over to [http://www.cnn.com/: 97819 bytes] for some general n
ews,
and [http://www.perlbuzz.com/: 59278 bytes] for the latest cool news about Perl.
Then I tried the address [http://xyz.liklkvsj.xx/: Couldn't fetch] that someone I didn'
t
```

This program prints out its data section with its URLs replaced by a small block that includes the URL and its length, or an error message. Let's walk through it together and see how it works:

OBSERVE:

```
s!(http://://.*?)(\s|$)!
```

We look for URLs that begin with **http://** and, by using the parentheses **()**, save them into **$1**. URLs can include all sorts of characters, so we say that ours ends when we see either a whitespace character (**\s**), or (**|**) the end of the line (**$**). By using the second set of parentheses **()**, we save that into **$2**.

OBSERVE:

```
!visit($1).$2!ge
```

Our replacement string starts with the result of calling the subroutine **visit()**, passing the URL we saved (**$1**). We put back the whitespace character we matched from **$2**, so we don't delete any spaces. (There is a feature of regular expressions outside the scope of this course called "lookahead" that allows you to match something without consuming it; you can read about it in perldoc perlre.

OBSERVE:

```
or next;
```

Just to remind you of the result of a substitution, we won't print any lines that don't match.

OBSERVE:

```
`curl -s $url 2>&1`
```

The **-s** flag to **curl** tells it not to output statistics. The **2>&1** construct tells the shell invoked by the backticks to redirect any errors from **curl** to standard output so they will go into **$content**. It is common to add to commands executed with backticks. If you don't do that, anything the external command sends to standard error will go wherever standard error is set—most likely the terminal. The intention of backticks is to intercept whatever the program outputs, and usually, we want that to include errors as well.

OBSERVE:

```
$? and return
```

The special variable **$?** is set by Perl to the exit status of the command that was last executed by **system** or backticks. A successful program will have an exit code of zero. We will assume that any other code returned by **curl** means that it was not able to retrieve the URL.

Isn't that cool? And here's something even cooler. Create **double_eval.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = join '', <DATA>;
s/([\d+*\/-]+)\s*=/qq{"$1 = " . ($1)}/eeg;
print;

__END__
1+2/4= and 42**3 =
 and 12-3/4 =
```

Check Syntax and run it:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./double_eval.pl
1+2/4 = 1.5 and 42**3 = 74088
 and 12-3/4 = 11.25
```

Wow! This program outputs the result of arithmetic expressions in the text itself! But why is the **/e** modifier repeated? What happens if you leave it off? Try it! Modify your code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$_ = join '', <DATA>;
s/([\d+*\/-]+)\s*=/qq{"$1 = " . ($1)}/~~e~~eg;
print;

__END__
1+2/4= and 42**3 =
 and 12-3/4 =
```

Check Syntax and run it:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./double_eval.pl
"1+2/4 = " . (1+2/4) and "42**3 = " . (42**3)
 and "12-3/4 = " . (12-3/4)
```

That's not so good, is it. The problem here is that we want Perl to do the work of evaluating the expression itself. So once we have captured the expression into a scalar, using **/e** to output that scalar has the effect you see above. We want Perl to evaluate the expression *again*, and we are allowed to repeat the **/e** modifier to get that effect.

But if we're going to evaluate the replacement string again, then our replacement string needs to be *code that generates code*. Here's how it works: Take an expression like **1+2/4**. That gets saved into **$1**. The replacement string is **qq{"$1 = " . ($1)}**. The *first* **/e** modifier means that the replacement string (including the quotation marks), gets evaluated, yielding the result **"1+2/4 = " . (1+2/4)**. The *second* **/e** modifier means that the previous result gets evaluated, yielding the result **1+2/4 = 1.5**.

By working backwards from the result you want, you can write these kinds of double-**/e** substitutions yourself. They don't come up often as practical solutions, but when they do, they are incredibly useful.

You're making great progress. Feel free to direct any questions you may have to your instructor. Keep up the good work and see you in the next lesson!

Once you finish the lesson, go back to the syllabus to complete the homework.

References

Lesson Objectives

When you complete this lesson, you will be able to:

- make references to Scalars
- make references to Arrays
- make a nonymous References

"County library? Reference desk, please. Hello? Yes, I need a word definition. Well, that's the problem. I don't know how to spell it and I'm not allowed to say it. Could you just rattle off all the swear words you know and I'll stop you when...Hello?"
-Bill Watterson, *Calvin & Hobbes*

Welcome to lesson seven, an exploration of **references** in Perl.

References to Scalars

I've claimed, "this is a really important and useful feature of Perl!" so often, I must sound like a cross between an infomercial presenter and a politician. But trust me, all the great stuff I've told you about Perl is true! But wait—there's more! Here's yet another great Perl feature: **references**.

References are scalars that point, or *refer to*, data somewhere else. When you want use the values they point to, you dereference the references.

So far you've seen that scalars can contain strings, numbers, or filehandles (or directory handles, which are a lot like filehandles). But a reference is an entirely new element for a scalar to contain.

References unlock the door to all the kinds of data representation and processing that you can do in Perl. The most advanced Perl programs depend on references. We'll spend most of the rest of this course learning about references in all their glory, which will enable you to write programs for complex data manipulation. And, if you go on to our fourth Perl course, we'll use that knowledge of references to do object-oriented programming for everything from e-mail to databases. Perl references are a really big deal!

We'll start with the basics; making and using references to scalars. But first, let me tell you what a Perl reference is *not:* it is *not* the same as a C/C++ pointer. (Don't worry if you're not familiar with C or C++.) Even though I will be using diagrams containing arrows, and I may use the phrase "points to" when talking about references, a reference in Perl can be used for only one thing: getting back the thing to which it points. You *cannot* perform "pointer arithmetic" on Perl references (if you're not familiar with pointer arithmetic, that's okay too).

In order to make a reference to a named **scalar** in our code, we type a **backslash** before the scalar; the result of that expression is the **reference** to that scalar:

OBSERVE:

```
$scalar_ref = \$scalar;
```

At first it may seem odd to use the backslash as a operator for forming a reference; so far we've only seen the backslash as an escape character or part of a digraph, but that was from inside of strings or regexes. Now our backslash occurs within program code. Let's try an example. Create **scalar_ref.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $marsupial = 'wallaby';

my $creature_ref = \$marsupial;

print $$creature_ref, "\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
code:~$ cd perl3
cold:~/perl3$ ./scalar_ref.pl
wallaby
```

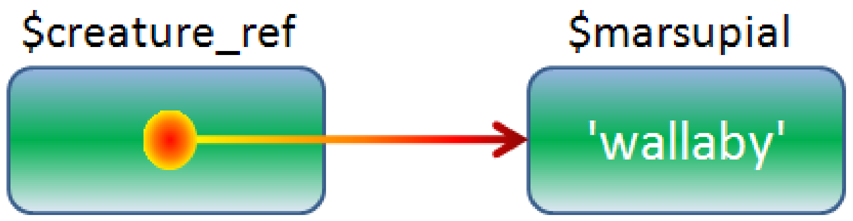
Okay, so that may not have been super exciting, but hey—we gotta start somewhere, right? In addition to *taking* a reference, we've shown you what to do with it: you can get back the scalar it refers to by putting a dollar sign (**$**) in front of the **reference**, like this:

OBSERVE:

```
$scalar = $$scalar_ref
```

At first, that construction may seem odd too because it brings to mind the special variable **$$** (which is the current process ID). Or it may just seem odd because, like the backslash, we haven't thought of a dollar sign as an operator before. And yet here both of those symbols are being pressed into service as operators. The different syntactical contexts in which they are used allow those symbols to be used for varied purposes.

Here's a visual aid for the reference to help makes this a bit more clear:



In this diagram the reference arrow points to the data that is stored in the variable with the name **$marsupial;** not at the name **$marsupial** itself. This distinction will become increasingly important as we go on through the lessons.

Now, if you're curious—and I hope you are—you probably already wondered what's inside of that orange blob in the **$scalar_ref** and what you can do with it. So let's do some experimentation with it to see what happens. (I encourage you to try this kind of thing yourself all the time—be fearless! You can't break our system. The worst that could happen is that you encounter something strange and you have to ask your instructor for help. No problem.) Modify the program as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $marsupial = 'wallaby';

my $creature_ref = \$marsupial;

print $$creature_ref, "\n";
print "\$creature_ref = $creature_ref\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./scalar_ref.pl
wallaby
$creature_ref = SCALAR(0x94a26fc)
```

That's really interesting! I bet you didn't see that coming. It's Perl's way of saying, "This is a reference to a scalar." (The hexadecimal number may be different on your screen, that's okay.)

We just printed out a scalar as a *string* (by interpolating it inside double quotation marks). I wonder if it would look different if we printed it as a *number*? Some special variables in Perl can behave differently when treated as numbers instead of strings. We can force Perl to interpret a scalar as a number by performing an arithmetic operation on the scalar that expects it to be a number, but won't change the value of that number. Adding or subtracting zero, or multiplying or dividing by one will do that. Modify **scalar_ref.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $marsupial = 'wallaby';

my $creature_ref = \$marsupial;

print $$creature_ref, "\n";
print "\$creature_ref = $creature_ref\n";
print "\$creature_ref = ", $creature_ref + 0, "\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./scalar_ref.pl
wallaby
$creature_ref = SCALAR(0x928d6fc)
$creature_ref = 155854588
```

Look at that: the reference is also a number. And you're probably thinking, "Is it the same number as the hexadecimal string in parentheses?" You can perform this test to find your answer. Modify **scalar_ref.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $marsupial = 'wallaby';

my $creature_ref = \$marsupial;

print $$creature_ref, "\n";
print "\$creature_ref = $creature_ref\n";
~~print "\$creature_ref = ", $creature_ref + 0, "\n";~~
printf "\$creature_ref = %x\n", $creature_ref;
```

(We no longer need to add zero because the **%x** format specified in **printf()** will impose a numeric context.)

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./scalar_ref.pl
wallaby
$creature_ref = SCALAR(0x928d6fc)
$creature_ref = 928d6fc
```

That's not so surprising. If Perl has a cryptic hexadecimal number in a reference's string value and a cryptic number associated with the same reference's numeric value, it makes sense that they would be the same number.

I mentioned earlier that pointer arithmetic was pointless with references. Let's add to it and see what happens. Modify **scalar_ref.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $marsupial = 'wallaby';

my $creature_ref = \$marsupial;

print $$creature_ref, "\n";
print "\$creature_ref = $creature_ref\n";
printf "\$creature_ref = %x\n", $creature_ref;
my $augmented = $creature_ref + 32;
printf "\$creature_ref + 32 = %x\n", $augmented;

print "\$augmented = $augmented\n";
print "\$$augmented = ", $$augmented, "\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./scalar_ref.pl
wallaby
$creature_ref = SCALAR(0x9bb96fc)
$creature_ref = 9bb96fc
$creature_ref + 32 = 9bb971c
$augmented = 163288860
Can't use string ("163288860") as a SCALAR ref while "strict refs" in use at ./scalar_r
ef.pl line 19.
```

So what happened? As soon as we added something to the reference, the result was a plain old number; there was no magic **SCALAR(0x...)** printed when we put it in a double-quoted string. Attempting to dereference that result produced an error.

The reference has only one really useful ability and that is to get back the thing it references, by *dereferencing* it. When our program changed, even the number that was associated with the reference changed. And you may have noticed that I have never referred to that number as an "address." It may look like an address, but you can't actually do anything with addresses in Perl. The only guarantee is that references to the same thing will stringify or numerify to the same values, and references to different things will not. Modify **scalar_ref.pl** as shown here:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $marsupial = 'wallaby';

my $creature_ref = \$marsupial;

print $$creature_ref, "\n";
print "\$creature_ref = $creature_ref\n";
~~printf "\$creature_ref = %x\n", $creature_ref;~~

~~my $augmented = $creature_ref + 32;~~
~~printf "\$creature_ref + 32 = %x\n", $augmented;~~

~~print "\$augmented = $augmented\n";~~
~~print "\$$augmented = ", $$augmented, "\n";~~

my $ref_copy = $creature_ref;
print "\$ref_copy     = $ref_copy\n";

my $australian = 'wallaby';
my $aussie_ref = \$australian;

print "\$aussie_ref   = $aussie_ref\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./scalar_ref.pl
wallaby
$creature_ref = SCALAR(0x93276fc)
$ref_copy     = SCALAR(0x93276fc)
$aussie_ref   = SCALAR(0x933b78c)
```

The numbers may be different for you, but the important thing to note is that the references **$creature_ref** and **$ref_copy** stringify identically (you can use **eq** to test whether they refer to the same thing); and **$creature_ref** and **$aussie_ref** stringify to different strings, because they point to different scalars, even though the scalars they point to *happen* to have the same value.

Okay, so that's enough about the mechanics of references for now; let's get busy working. We're going to focus more or less exclusively now on taking references and dereferencing. References to scalars are good, but references to arrays even better! Let's learn about those.

References to Arrays

For our next example, we'll use a reference to an array with the backslash operator, just like we did before:

OBSERVE:

```
$array_ref = \@array;
```

Create **array_ref.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

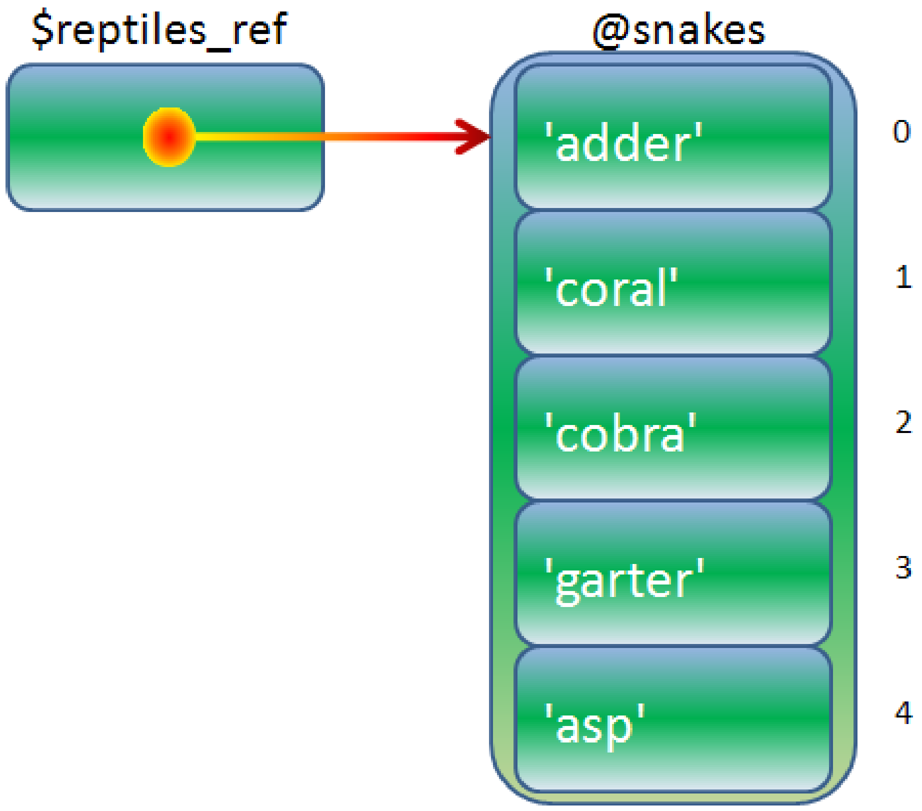
```
#!/usr/bin/perl
use strict;
use warnings;

my @snakes = qw(adder coral cobra garter asp);

my $reptiles_ref = \@snakes;

my $index;
printf "%d $_\n", $index++ for @$reptiles_ref;
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./array_ref.pl
0 adder
1 coral
2 cobra
3 garter
4 asp
```

Perl is very consistent in its reference syntax: to dereference a reference to an array, put an at sign (**@**) in front of it; this will return the contents of the array. Here's a graphic representation to help you visualize and understand array references:



When you have a reference to something, it provides you with a way to change that relative data. Create **ref_mod.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

{
  my $marsupial = 'wallaby';
  my $creature_ref = \$marsupial;

  my @snakes = qw(adder coral cobra garter asp);
  my $reptiles_ref = \@snakes;

  modify_refs( $creature_ref, $reptiles_ref );

  print "\$marsupial = $marsupial\n";
  print "\@snakes = @snakes\n";
}

sub modify_refs
{
  my ($scalar_ref, $array_ref) = @_;

  $$scalar_ref .= ' stew';
  pop @$array_ref;
}
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./ref_mod.pl
$marsupial = wallaby stew
@snakes = adder coral cobra garter
```

Even from deep inside that subroutine, your reference was able to modify **$marsupial** and **@snakes** through the references to them, even though the variables they referred to were no longer in scope during that subroutine. Remember, references point to *storage:* the data *stored* in the variable for which you've taken a reference.

Everywhere you can use a scalar variable like **$marsupial**, you can use a dereferenced reference to a scalar: **$$creature_ref**. In the subroutine above we copied the reference **$creature_ref** into **$scalar_ref** (it still points to the same thing—we established that behavior earlier) and we were able to use the dereferenced reference as an **lvalue** (an **lvalue** is an expression that you can write on the left hand side of an assignment statement; it defines a specific memory address of a variable) to modify the original data. The same goes for the array reference.

You can use an array reference anywhere you'd have the *name* portion of an array (the identifier that comes after the **@** sigil—a sigil is the bit of punctuation that tells Perl what sort of variable is being used) in code; this program gives some examples of those constructions so you can see what I mean. Create **array_ref_ex.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @countries = qw(Sudan Sweden Switzerland Surinam Singapore);
my $countries_ref = \@countries;

print "Array     in scalar context: " . @countries       . "\n";
print "Array ref in scalar context: " . @$countries_ref . "\n";

print "Array in list context:     ", @countries,      "\n";
print "Array ref in list context: ", @$countries_ref, "\n";

print "Third member of array:      $countries[2]\n";
print "Third member via array ref: $$countries_ref[2]\n";

print "Index of last element of array:      $#countries\n";
print "Index of last element via array ref: $#$countries_ref\n";
```

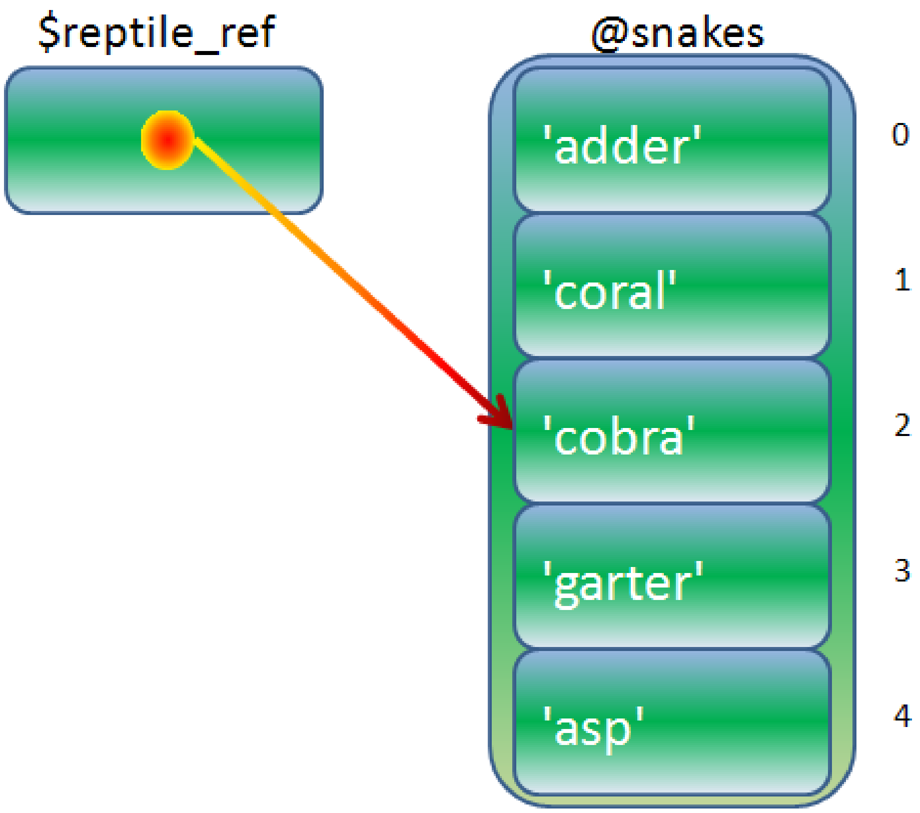
Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./array_ref_ex.pl
Array     in scalar context: 5
Array ref in scalar context: 5
Array in list context:     SudanSwedenSwitzerlandSurinamSingapore
Array ref in list context: SudanSwedenSwitzerlandSurinamSingapore
Third member of array:      Switzerland
Third member via array ref: Switzerland
Index of last element of array:      4
Index of last element via array ref: 4
```

Do you see what I mean? If you want to know how to do a particular operation with an array reference, just ask yourself how you'd do it with an array named, for instance, **@foo**, and replace the **foo** part of the construction with the reference.

Take another look at the graphical representation of the array reference. I made sure to draw a distinction between an array and the scalars that are contained within it. What do you suppose *this* diagram represents?:



That's a reference to an array *element* rather than the whole array. So what kind of reference is it? It's a reference to a *scalar!* Create **element_ref.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @snakes = qw(adder coral cobra garter asp);
my $reptile_ref = \$snakes[2];

print "\$reptile_ref looks like $reptile_ref\n";

$$reptile_ref = 'boa';

print "\@snakes = @snakes\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./element_ref.pl
$reptile_ref looks like SCALAR(0x9e2b7a4)
@snakes = adder coral boa garter asp
```

The reference **$reptile_ref** is a scalar reference, and we can use it to modify the element of the array to which it points. (We *cannot* use our reference to get at the array itself or any other element of the array.)

Anonymous References

You may have noticed that I tend to repeat myself. It isn't because I don't have confidence in your ability to catch information the first time around. I repeat important information to help you retain it—and there's a lot of it. So let me reiterate here, references point to the data that is stored in a scalar or array, not to the name of that scalar or array. In fact, given a reference, you cannot retrieve the name of the thing that it points to, only the data stored in it. And frequently we write subroutines that expect one or more references in their arguments, so we end up passing references around our code like ordinary data. So, what if we don't actually *need* a named variable at all, because we plan to use only references?

Suppose we have a subroutine and its purpose is to return an array reference for later use. Create **anon_ref.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $day_ref = make_day_ref( 'German' );

print "Der dritte tag der woche ist $$day_ref[2]\n";


sub make_day_ref
{
  my $language = shift;

  my @english_names = qw(Sunday Monday Tuesday Wednesday Thursday Friday Saturday);
  my @french_names  = qw(Dimanche Lundi Mardi Mercredi Jeudi Vendredi Samedi);
  my @german_names  = qw(Sonntag Montag Dienstag Mittwoch Donnerstag Freitag Samstag);
  if ( $language eq 'English' )
  {
    return \@english_names;
  }
  elsif ( $language eq 'French' )
  {
    return \@french_names;
  }
  elsif ( $language eq 'German' )
  {
    return \@german_names;
  }
  else
  {
    die "Unrecognized language $language";
  }
}
```

Check Syntax and run it (you know what to do):

INTERACTIVE SESSION:

```
cold:~/perl3$ ./anon_ref.pl
Der dritte tag der woche ist Dienstag
```

This code could be a good start to internationalizing a program. But look how long our routine **make_day_ref** is. Essentially it's just defining data. So, what's the point in defining the names **@english_names**, **@french_names**, and so on, when we only use them once, in order to take a reference?

The answer lies in a new piece of Perl syntax, the **anonymous array reference constructor**. That's quite a mouthful! The **anonymous array reference constructor** is an expression that returns a reference to an array that we never bothered putting into a named variable before, so we put **square brackets** around the ***list*** we want in that array instead, like this:

OBSERVE:

```
$array_ref = [ list ]
```

Let's see that at work; modify **anon_ref.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $day_ref = make_day_ref( 'German' );

print "Der dritte tag der woche ist $$day_ref[2]\n";


sub make_day_ref
{
  my $language = shift;

~~  my @english_names = qw(Sunday Monday Tuesday Wednesday Thursday Friday Saturday);~~
~~  my @french_names  = qw(Dimanche Lundi Mardi Mercredi Jeudi Vendredi Samedi);~~
~~  my @german_names  = qw(Sonntag Montag Dienstag Mittwoch Donnerstag Freitag Samstag);~~
  if ( $language eq 'English' )
  {
    return ~~\@english_names~~ [ qw(Sunday Monday Tuesday Wednesday Thursday Friday Saturda
y) ];
  }
  elsif ( $language eq 'French' )
  {
    return ~~\@french_names~~ [ qw(Dimanche Lundi Mardi Mercredi Jeudi Vendredi Samedi) ];
  }
  elsif ( $language eq 'German' )
  {
    return ~~\@german_names~~ [ qw(Sonntag Montag Dienstag Mittwoch Donnerstag Freitag Sams
tag) ];
  }
  else
  {
    die "Unrecognized language $language";
  }
}
```

Check Syntax and run it. The result should be the same as before. But we've made the code shorter! Now we don't need to think up variable names just to be able to construct references.

The anonymous array ref constructor may look odd; we're used to seeing square brackets used to delimit array indexes. Don't worry, Perl can tell when you're using them for this purpose.

Now you may be wondering whether there is an anonymous scalar ref constructor. There is, but we don't often have a need for them, so I'll just show it to you briefly:

OBSERVE:

```
$scalar_ref = \LITERAL
```

That's right, you just put a **backslash** in front of any literal value to which you want a reference. For instance:

OBSERVE:

```
$e_ref = \2.71828183;
$name_ref = \"Peter";
```

These anonymous scalar references are *immutable* (or "constant," if you like); you can't change them. That quality can be useful in certain circumstances. Try this one-liner:

INTERACTIVE SESSION:

```
cold:~/perl3$ perl -le '$s_ref = \"four"; print $$s_ref; $$s_ref = 42'
four
Modification of a read-only value attempted at -e line 1.
```

That is *not* true of anonymously-constructed array references:

INTERACTIVE SESSION:

```
cold:~/perl3$ perl -le '$a_ref = [ 10..20 ]; print "@$a_ref"; $$a_ref[5] *= 2; print "@
$a_ref"'
10 11 12 13 14 15 16 17 18 19 20
10 11 12 13 14 30 16 17 18 19 20
```

Finally, here's what happens when you stringify (in this case, by printing) an array ref:

OBSERVE:

```
cold:~/perl3$ perl -le '$a_ref = [ 10..20 ]; print $a_ref'
ARRAY(0x8cf2158)
```

There's no real use to printing out a reference, but sooner or later you'll make the mistake of using a reference where you meant to use a scalar, and you'll see a string like that instead. But this way you'll know what it means.

We've only just begun to cover references and you can see already that they are the keys to "packaging" data to pass around your program. It gets even better from here!

Once you finish the lesson, go back to the syllabus to complete the homework.

Hash References and Dereferencing Syntax

Lesson Objectives

When you complete this lesson, you will be able to:

- make references to Named Hashes.
- use the Data::Dumper module.
- make references to Anonymous Hashes.
- apply both rules for Dereferencing.

Hello and welcome back! This lesson will include lots of valuable information.

References to Named Hashes

Now that you've experienced the power of scalar and array references, you probably aren't surprised to learn that you can also take references to hashes. In fact, references to hashes are arguably the most useful references in Perl.

Perl remains consistent in its syntax. To take a reference to a hash, put a backslash \ before the hash:

OBSERVE:

```
$hash_ref = \%hash
```

Let's try a short example. Create **hash_ref.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

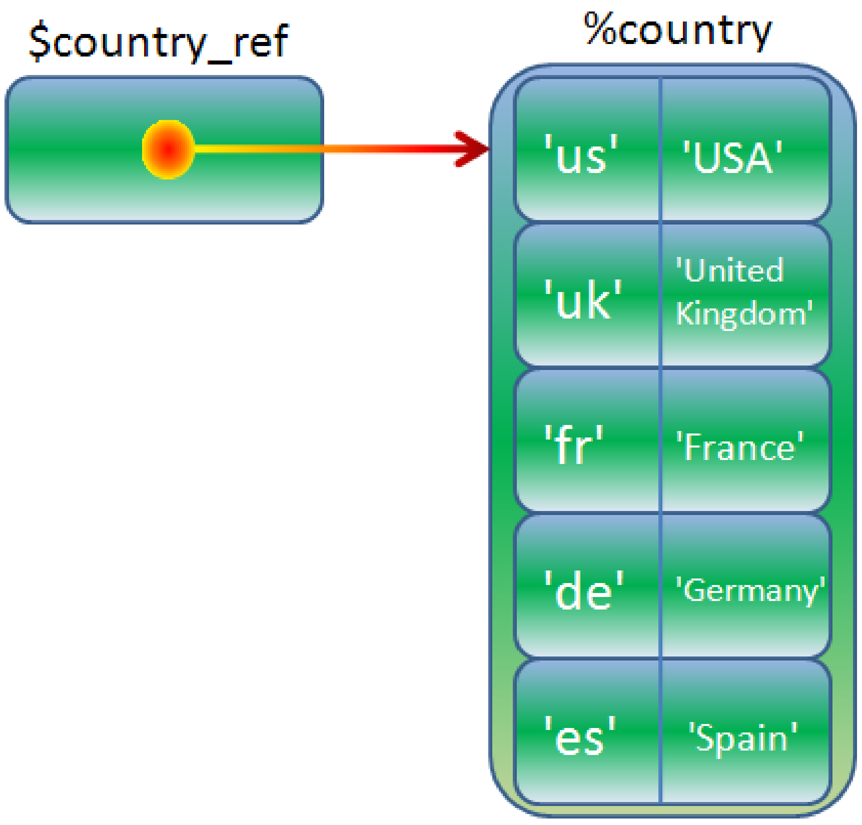
```
#!/usr/bin/perl
use strict;
use warnings;

my %country = ( us => 'USA',     uk => 'United Kingdom', fr => 'France',
                de => 'Germany', es => 'Spain',          mx => 'Mexico',
                jp => 'Japan',   in => 'India',          th => 'Thailand' );

my $country_ref = \%country;

print "\$country_ref = $country_ref\n";

print "$_ = $$country_ref{$_}\n" for keys %$country_ref;
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

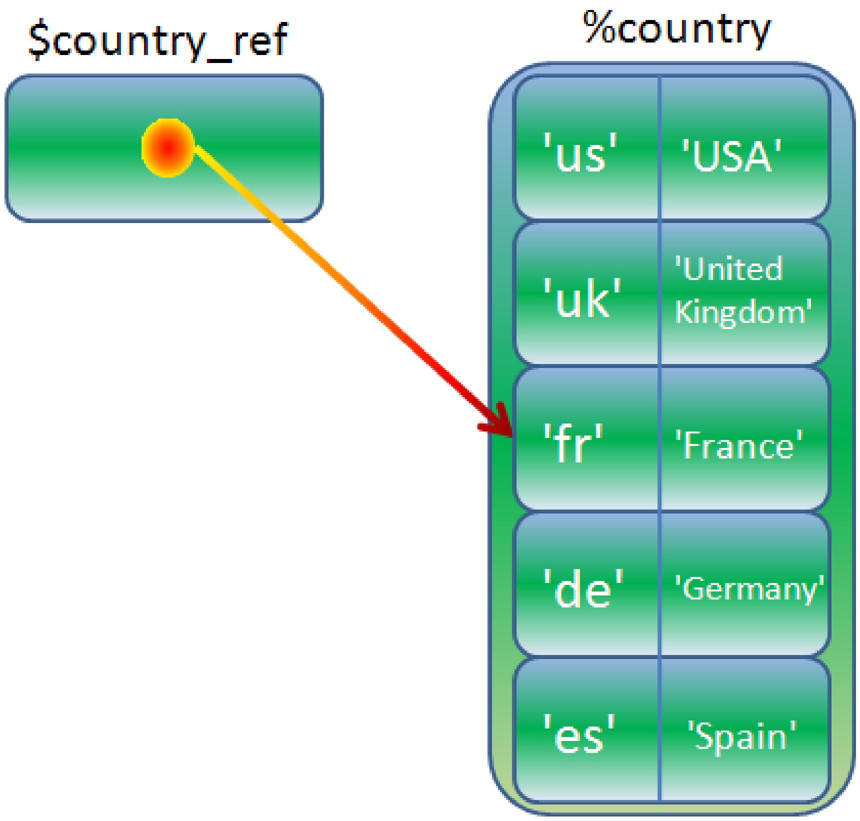
```
cold:~$ cd perl3
cold:~/perl3$ ./hash_ref.pl
$country_ref = HASH(0x8c2971c)
mx = Mexico
uk = United Kingdom
fr = France
jp = Japan
de = Germany
in = India
es = Spain
us = USA
th = Thailand
```

Here's a graphical representation of that operation:



First we printed a stringified **hash ref**. Then we show a couple of ways to dereference a hash ref. A hash ref works just like an array ref: you can use a hash ref everywhere you would ordinarily use the *name* portion of a hash (the identifier that comes after the **%** sigil) in code. So, to access the keys of the hash via the reference, if we were working on an actual hash like **%country**, we would type **%country**, but since we're working on a hash *reference*, we substitute the reference for the identifier portion (**country**) of the hash and end up with **%$country_ref**. Then we applied the same principle to get each successive value in the hash with the key **$_**. If we had an actual hash **%country**, we would access **$country{$_}**, but instead, we have a reference, so we end up with **$$country_ref{$_}**.

The diagram above illustrates that a hash is a container that exists independently of the scalars within it, and so a reference to the hash is not the same as a reference to a member of that hash. A reference to a member of the hash would look like this:



Let's see how we'd do that. Create **element_ref2.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %country = ( us => 'USA',     uk => 'United Kingdom', fr => 'France',
                de => 'Germany', es => 'Spain',          mx => 'Mexico',
                jp => 'Japan',   in => 'India',          th => 'Thailand' );

my $hispanic_ref = \$country{mx};

print "\$hispanic_ref = $hispanic_ref\n";
$$hispanic_ref = 'United Mexican States';  # http://en.wikipedia.org/wiki/Mexico

print "$_  = $country{$_}\n" for sort keys %country;
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./element_ref2.pl
$hispanic_ref = SCALAR(0x819f75c)
de = Germany
es = Spain
fr = France
in = India
jp = Japan
mx = United Mexican States
th = Thailand
uk = United Kingdom
us = USA
```

Looks good!

Data::Dumper

Now let's take a different look at hash references. Open **map_hash.pl** in your **/perl3** folder and modify it as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

use Data::Dumper;

my %marsupial = map { ($_, 1) } qw(koala kangaroo possum wombat);

chomp( my @amphibians = <DATA> );

my %amphibian = map { $_, 1 } @amphibians;

~~print "Marsupials: ", join( ' ', sort keys %marsupial ), "\n";~~
~~print "Amphibians: ", join( ' ', sort keys %amphibian ), "\n";~~
print Dumper \%amphibian, \%marsupial;

__END__
frog
toad
salamander
newt
caecilian
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./map_hash.pl
$VAR1 = {
          'toad' => 1,
          'newt' => 1,
          'salamander' => 1,
          'caecilian' => 1,
          'frog' => 1
        };
$VAR2 = {
          'wombat' => 1,
          'kangaroo' => 1,
          'possum' => 1,
          'koala' => 1
        };
```

(You might see the hash elements in a different order.)

Let's take a closer look at this program:

OBSERVE:

```
#!/usr/bin/perl
use strict;
use warnings;

use Data::Dumper;

my %marsupial = map { ($_, 1) } qw(koala kangaroo possum wombat);

chomp( my @amphibians = <DATA> );

my %amphibian = map { ($_, 1) } @amphibians;

print Dumper \%amphibian, \%marsupial;
__END__
frog
toad
salamander
newt
caecilian
```

We use the map technique we learned about earlier to initialize a hash. We use the handy **Data::Dumper** module, which provides the **Dumper()** function which displays the contents of any list of variables, including references. This feature is really useful for printing a quick dump of a data structure when you're troubleshooting a program. (You can delve more deeply into modules and even learn how to write your own in the next Perl course!)

I've passed the **Dumper()** routine two arguments: two references to hashes. **Dumper()** can recognize a hash ref (we'll see how later). As you may recall, a hash ref does not give you the means to get at the name of the variable it references, so **Dumper()** prints **$VAR1** and **$VAR2** instead.

So what's up with this **{ ($_, 1) }** syntax that **Dumper()** has chosen to denote a hash ref? That's the topic of our next section!

References to Anonymous Hashes

Perl is consistent in its choice of syntax for references to anonymous hashes. Just as anonymous array refs are delimited with square brackets (square brackets are used around array indexes), anonymous hash refs are delimited with curly braces **{}** (similar to the way they're used around hash keys). Here's a general picture of the syntax:

OBSERVE:

```
$hash_ref = { LIST }
```

And here's that syntax at work in a specific example:

OBSERVE:

```
$stock_ref = { ants => 10E6, aardvarks => 4, antelopes => 2 }
```

Fortunately, Perl can tell when you intend for braces to be used as an anonymous hash ref constructor, even when you're using them as the *implicit* result returned from a subroutine. Create **anon_hash.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $trans_ref = make_trans_ref( 'French' );

print "J'ai un rendez-vous $$trans_ref{Friday} soir\n";

sub make_trans_ref
{
  my $language = shift;

  if ( $language eq 'French' )
  {
    { Sunday    => 'dimanche', Monday   => 'lundi', Tuesday => 'mardi',
      Wednesday => 'mercredi', Thursday => 'jeudi',
      Friday    => 'vendredi', Saturday => 'samedi' };
  }
  elsif ( $language eq 'German' )
  {
    { Sunday    => 'Sonntag',  Monday   => 'Montag', Tuesday => 'Dienstag',
      Wednesday => 'Mittwoch', Thursday => 'Donnerstag',
      Friday    => 'Freitag',  Saturday => 'Samstag' };
  }
  else
  {
    die "Unknown language $language";
  }
}
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./anon_hash.pl
J'ai un rendez-vous vendredi soir
```

Perl doesn't require you to use the keyword **return** to return a value from a subroutine; the value of the last expression evaluated will be the returned value if there is no **return** statement. Still, I recommend that you do use it, especially in a case like this, because it's difficult to see that we've inserted anonymous hash refs rather than naked blocks (Perl *still* gets it right though). Go ahead and put the **return** statement in before each of those anonymous hash refs and see how much clearer the code is.

Of course, you can use any list expression inside the braces, not just a literal list. Create **stock.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $stock_ref = get_stock_ref( 'INSECTS' );
print "I have $$stock_ref{ants} ants\n";

sub get_stock_ref
{
  my $seeking = shift;

  my (@data, $type);
  while ( <DATA> )
  {
    if ( ( my ($what, $number) = /(.*)\s+(\d+)/ ) && ( $type eq $seeking ) )
    {
      push @data, $what, $number;
    }
    elsif ( /(\S+)/ )
    {
      $type = $1;
    }
    else
    {
      next;
    }
  }
  return { @data };
}

__END__

INSECTS
ants 1000000
beetles 200000

MAMMALS
aardvarks 4
antelopes 3

FISH
guppies 10
angel fish 40
king crabs 10
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./stock.pl
I have 1000000 ants
```

This isn't very good program design; the **get_stock_ref** routine can only be called once before running out of data. Still, the program is useful because it demonstrates that you can populate an anonymous hash ref with an array. (Although it would be better to populate a named hash and return a reference to it.)

Okay, now here's a handy idiom for setting defaults, used with a hash ref. Create **default.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

use Data::Dumper;

my $supplied_ref = { @ARGV };
my $default_ref = { pants => 3, shirts => 6, ties => 10 };
my $merged_ref = { %$default_ref, %$supplied_ref };
print Dumper $merged_ref;
```

Check Syntax and run it, first without parameters and then with parameters, as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./default.pl
$VAR1 = {
          'pants' => 3,
          'ties' => 10,
          'shirts' => 6
        };
cold:~/perl3$ ./default.pl socks 4 ties 6
$VAR1 = {
          'socks' => '4',
          'pants' => 3,
          'ties' => '6',
          'shirts' => 6
        };
```

Inside the last anonymous hash ref constructor, we dereference first the defaults, then the arguments that were supplied, so that the arguments take precedence over the defaults (in a list of key-value pairs assigned to a hash, if any key is repeated, the last occurrence wins).

Dereferencing: Two Rules

There are two rules for dereferencing in general. We've only exercised one of them so far.

You may construct a dereferencing expression by figuring out what the equivalent expression would be for the type of thing (scalar, array, or hash) your reference points to, and then substituting the *name* or *identifier* portion of that type of thing (that which follows the **$ @**, or **%**) with either:

- Rule 1: A simple scalar (like **$array_ref**) containing a reference to that kind of thing; or
- Rule 2: A block containing anything other than a simple scalar that evaluates to a reference to that kind of thing.

So far, we've only exercised rule 1, because we've only had references contained in simple scalars. But, while an array element like **$insects_ref[2]** is a scalar, it's not a simple one. An expression that evaluates to a reference, say, **make_stock_ref('INSECT')** is not a simple scalar either.

These rules are actually harder to describe than they are to understand, so let's try an example that makes them more clear. Create **ref_rules.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $furn_ref = [ qw(sofa chair loveseat piano credenza) ];
my $cost_ref = { sofa => 1500, chair => 300, loveseat => 800, piano => 6000, credenza =
> 500 };

# Rule 1
print "Cost of third item from \$furn_ref is: ",
      $$cost_ref{ $$furn_ref[2] },
      "\n";

# Rule 2
my $alt_furn_ref = [ qw(bed rug rocker stool) ];
my $alt_cost_ref = { bed => 3000, rug => 75, rocker => 450, stool => 125 };
my @refs = ( $cost_ref, $alt_cost_ref );

print "Cost of third item in \$alt_furn_ref from second reference in array is: ",
      ${ $refs[1] }{ $$alt_furn_ref[2] },
      "\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./ref_rules.pl
Cost of third item from $furn_ref is: 800
Cost of third item in $alt_furn_ref from second reference in array is: 450
```

That looks pretty tricky, but don't worry! We'll go over all of it, and in the next lesson, we'll show you how some of those constructions can be simplified.

In the first case—exercising Rule 1—we want to get the third item from **$furn_ref**, which is an array reference, **$$furn_ref[2]**, and then print the cost of that item, which is stored in **$cost_ref**, which is a hash reference, **$$cost_ref{ $$furn_ref[2] }**. The hash *key* is a complex value, but the *reference* is in a simple scalar, **$cost_ref**, so this is an application of Rule 1.

In the second case—exercising Rule 2—we want to get the third item from **$alt_furn_ref**, which is an array reference, **$$alt_furn_ref[2]** (application of Rule 1), and then print the cost of that item, which is stored in the second element in **@refs** (that is, **$refs[1]**), which is a hash reference. That is *not* a simple scalar, so we apply Rule 2, and put it in a *block*—that's the **{ $refs[1] }** part—and then add the complete expression as **${ $refs[1] }{ $$alt_furn_ref[2] }**.

Wow. Fortunately, that's about as complicated as it ever gets with references. We'll learn how to create much more complicated data structures, but for now, our syntax is complex enough. Great work today!

Once you finish the lesson, go back to the syllabus to complete the homework.

The Arrow Operator and Multidimensional Arrays

Lesson Objectives

When you complete this lesson, you will be able to:

- use the Arrow Operator.
- create lists of lists.
- reduce the time required for sorting using the Schwartzian Transform.

Welcome back to your Advanced Perl course! Can you believe it? We're already past the halfway point in the course.

The Arrow Operator

If you find some of the expressions involving references complicated, you're not alone. It can still be pretty challenging even after you know how to construct or read an expression like:

OBSERVE:

```
${ $refs[1] }{ $$alt_furn_ref[2] }
```

Fortunately, there's the **arrow operator**: **->**. The arrow operator makes more complex code easier to read. It's a *binary infix* operator, which means it operates on two things, one to its left and one to its right (**+** is another such operator):

LEFT->*RIGHT*

The operation it performs depends on what's on the *right* side. The arrow operator is used for several purposes; first we'll see how the arrow operator *accesses members of aggregates pointed to by references*.

That means we can use it to get at an element of an array, given an array ref, or an element of a hash, given a hash ref. Let's try an example. Create **arrow.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @clouds = qw(cumulus nimbus cumulonimbus stratus cirrus);
my $clouds_ref = \@clouds;

print "Cloud types:\n";
print "\t $clouds_ref->[$_]\n" for 0 .. $#$clouds_ref;

# http://www.navy.mil/navydata/questions/bells.html
my %mid_bells = ( one => '0030', two => '0100', three => '0130', four => '0200',
                  five => '0230', six => '0300', seven => '0330', eight => '0400' );
my $mid_ref = \%mid_bells;
print "Fourth mid bell is at $mid_ref->{four}\n";
```

Check Syntax and run it by typing the commands below as shown:

INTERACTIVE SESSION:

```
cold:~$ cd perl3
cold:~/perl3$  ./arrow.pl
Cloud types:
         cumulus
         nimbus
         cumulonimbus
         stratus
         cirrus
Fourth mid bell is at 0200
```

In the above example the arrow operator works with curly braces. When using the arrow operator, if the right side consists of curly braces around an expression, then the right side is interpreted as braces around a hash key, the left side must be a reference to a hash, and the result is the element with that key in the hash pointed to by the reference.

Now we'll use the arrow operator with square brackets to improve the readability of an earlier program. When using the arrow operator, if the right side consists of square brackets around an expression, then the right side is interpreted as brackets around an array index, the left side must be a reference to an array, and the result is the element at that index of the array pointed to by the reference.

Go to your **/perl3** folder, open **ref_rules.pl** and modify it as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $furn_ref = [ qw(sofa chair loveseat piano credenza) ];
my $cost_ref = { sofa => 1500, chair => 300, loveseat => 800, piano => 6000, credenza =
> 500 };

# Rule 1
print "Cost of third item from \$furn_ref is: ",
      ~~$$cost_ref{ $$furn_ref[2] },~~
      $cost_ref->{ $furn_ref->[2] },
      "\n";

# Rule 2
my $alt_furn_ref = [ qw(bed rug rocker stool) ];
my $alt_cost_ref = { bed => 3000, rug => 75, rocker => 450, stool => 125 };
my @refs = ( $cost_ref, $alt_cost_ref );

print "Cost of third item in \$alt_furn_ref from second reference in array is: ",
      ~~${ $refs[1] }{ $$alt_furn_ref[2] },~~
      $refs[1]->{ $alt_furn_ref->[2] },
      "\n";
```

Check Syntax and run it.

The output will be the same as before, only the code syntax has changed.

We could change some of our other earlier programs in the same way. This code fragment:

OBSERVE:

```
print "I have $$stock_ref{ants} ants\n";
```

...could be replaced with this:

OBSERVE:

```
print "I have $stock_ref->{ants} ants\n";
```

...because the arrow operator also works when interpolated in double-quoted strings.

The arrow operator can only be used to get *elements* of an aggregate using a reference, so there are still times that you'll need to use the dereferencing rules from the last lesson; if you want to access the whole aggregate, for instance, to get the keys of a hash. (You will see more uses of that in the next few lessons.) But we'll use the arrow syntax from now on for all accesses of aggregate members.

So, what happens if the reference doesn't point to the correct type of thing? Let's find out using this one-liner:

INTERACTIVE SESSION:

```
cold:~/perl3$ perl -wle '$h_ref = { J => 10, Q => 10, K => 10 }; print $h_ref->[42]'
Not an ARRAY reference at -e line 1.
```

That's a fatal run-time exception. The right side of the arrow operator has square brackets, so it expects the left side to be a reference to an array.

Let's do one more arrow operator example, this time using a possible solution to the Lesson 8 project. Create **hash_compare.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %first  = ( Wallace => 1, Gromit => 2 );
my %second = ( Gromit => 2, Wallace => 1 );
compare( \%first, \%second );

%first  = ( Kirk => 'Captain', Spock => 'First Officer', McCoy => 'Doctor' );
%second = ( Spock => 'First Officer', McCoy => 'Doctor' );
compare( \%first, \%second );

%first  = ();
%second = ();
compare( \%first, \%second );

%first  = ( Wallace => 1, Gromit => 2 );
%second = (Wallace => 2, Gromit => 1);
compare( \%first, \%second );

%first = (Wallace => 1, Gromit => 2);
%second= (Wallace => 1, Gromit => 2, Shawn => 3);
compare( \%first, \%second );


sub compare
{
  print "Hashes are ", compare_hashes( @_ ) ? "EQUAL" : "NOT EQUAL", "\n";
}

# The following routine was the homework
sub compare_hashes
{
  my ($h1_ref, $h2_ref) = @_;

  return 0 if keys %$h1_ref != keys %$h2_ref;  # Unequal # of elements

  for ( keys %$h1_ref )
  {
    return 0 if ! exists $$h2_ref{$_} || $$h1_ref{$_} ne $$h2_ref{$_};
  }
  return 1;     # Can't be any keys left unvisited
}
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./hash_compare.pl
Hashes are EQUAL
Hashes are NOT EQUAL
Hashes are EQUAL
Hashes are NOT EQUAL
Hashes are NOT EQUAL
```

Now we'll make this small modification using the arrow operator to make our program a bit more readable:

CODE TO TYPE:

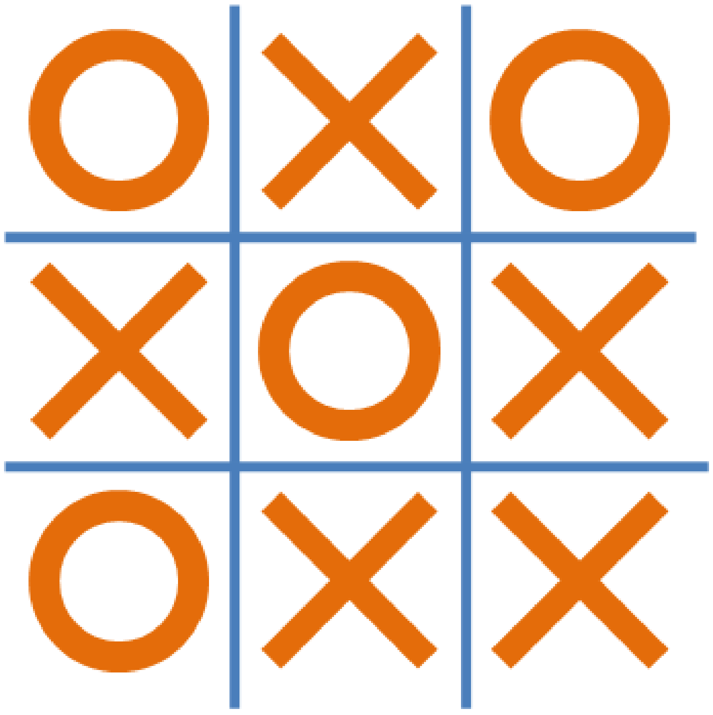
```
#!/usr/bin/perl
use strict;
use warnings;

my %first  = ( Wallace => 1, Gromit => 2 );
my %second = ( Gromit => 2, Wallace => 1 );
compare( \%first, \%second );

%first  = ( Kirk => 'Captain', Spock => 'First Officer', McCoy => 'Doctor' );
%second = ( Spock => 'First Officer', McCoy => 'Doctor' );
compare( \%first, \%second );

%first  = ();
%second = ();
compare( \%first, \%second );

%first  = ( Wallace => 1, Gromit => 2 );
%second = (Wallace => 2, Gromit => 1);
compare( \%first, \%second );

%first = (Wallace => 1, Gromit => 2);
%second= (Wallace => 1, Gromit => 2, Shawn => 3);
compare( \%first, \%second );


sub compare
{
  print "Hashes are ", compare_hashes( @_ ) ? "EQUAL" : "NOT EQUAL", "\n";
}

# The following routine was the homework
sub compare_hashes
{
  my ($h1_ref, $h2_ref) = @_;

  return 0 if keys %$h1_ref != keys %$h2_ref;   # Unequal # of elements

  for ( keys %$h1_ref )
  {
    ~~return 0 if ! exists $$h2_ref{$_} || $$h1_ref{$_} ne $$h2_ref{$_};~~
    return 0 if ! exists $h2_ref->{$_} || $h1_ref->{$_} ne $h2_ref->{$_};
  }
  return 1;     # Can't be any keys left unvisited
}
```

Lists of Lists

Time to dig into a vital topic that we call *Lists of Lists*. That's actually an inaccurate term, because you can't have lists of lists in Perl; what we really mean is "Aggregates containing references to other aggregates," where "aggregate" is an array or a hash, but that's a pretty big mouthful, so we'll stick with "lists of lists." That term is ingrained in Perl culture, and it's the title of the Perl document found through **perldoc perllol**.

Multidimensional Arrays

Let's start our experimentation with a familiar entity: the two-dimensional array. Even if you've never programmed one before, you have a sense of how it works if you've seen a chess board. We'll work on a smaller scale first with the tic-tac-toe board ("noughts and crosses" as it's called in some countries). The game is played on a 3x3 board and the first person to get three of their marks in a row in any direction wins:



We're not going to play the game, we'll just represent the board. Create **game.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @board = ( [ qw(O X O) ],
              [ qw(X O X) ],
              [ qw(O X X) ]);

print_board( \@board );

sub print_board
{
  my $board_ref = shift;

  print "-" x 13, "\n";
  for my $row ( @$board_ref )
  {
    for my $column ( @$row )
    {
      print "| $column ";
    }
    print "|\n";
  }
  print "-" x 13, "\n";
}
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./game.pl
-------------
| O | X | O |
| X | O | X |
| O | X | X |
-------------
```

Somehow **O** has managed to outwit **X**! Notice that we passed a reference to **@board** in order to **print_board**; we could have passed **@board** itself instead, as long as we wrote **print_board** to expect an array instead of an array ref. The programs you write will reflect your own subroutine interface design preferences.

Our board has been represented as an array of references to arrays. If we designate positions on the board according to the intersections of rows and columns numbered from 1 to 3, we can then add code to indicate specific locations. (Remember that Perl's arrays are indexed from zero, not one.) Let's give that a try. Modify **game.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @board = ( [ qw(O X O) ],
              [ qw(X O X) ],
              [ qw(O X X) ]);

print_board( \@board );

set_board( \@board, 2, 3, 'O' );

print_board( \@board );

sub set_board
{
  my ($board_ref, $row, $column, $piece) = @_;

  $board_ref->[$row-1]->[$column-1] = $piece;
}

sub print_board
{
  my $board_ref = shift;

  print "-" x 13, "\n";
  for my $row ( @$board_ref )
  {
    for my $column ( @$row )
    {
      print "| $column ";
    }
    print "|\n";
  }
  print "-" x 13, "\n";
}
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./game.pl
-------------
| O | X | O |
| X | O | X |
| O | X | X |
-------------
-------------
| O | X | O |
| X | O | O |
| O | X | X |
-------------
```

Success! We changed the mark at the end of the middle row.

Perl doesn't actually have a two-dimensional array as a proper type like many languages do. Instead, Perl lets you put anything you want into a one-dimensional array, including references to other one-dimensional arrays. An array that contains references to other arrays that are all of the same size is functionally equivalent to a two-dimensional array; but nothing stops you from changing that at any point during run-time. It's up to you to make sure the arrays are the correct size.

Multidimensional Syntax Optimization

Perl may not have real multidimensional arrays, but you can still benefit from their syntactical advantages! Make this slight modification to **game.pl**:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @board = ( [ qw(O X O) ],
              [ qw(X O X) ],
              [ qw(O X X) ]);

print_board( \@board );

set_board( \@board, 2, 3, 'O' );

print_board( \@board );

sub set_board
{
  my ($board_ref, $row, $column, $piece) = @_;

  $board_ref->[$row-1]->[$column-1] = $piece;
}

sub print_board
{
  my $board_ref = shift;

  print "-" x 13, "\n";
  for my $row ( @$board_ref )
  {
    for my $column ( @$row )
    {
      print "| $column ";
    }
    print "|\n";
  }
  print "-" x 13, "\n";
}
```

That's right, remove the arrow. Check Syntax and run it. The results are unchanged. This is a nice piece of *syntactic sugar* from Perl:

Note When there are two pairs of subscripting brackets or braces separated by only an arrow, the arrow can be omitted.

So, you could represent a 3-D structure with, say, an array **@cube** of references to *references* to arrays. Then, addressing any cell in the cube would look like this: **$cube[$x][$y][$z]**. As you saw in our example above, when we *iterate through* or *traverse* a multidimensional structure, we usually extract each aggregate reference into a separate scalar as we go, so that we expand the next level with code like **@$ref** and not, for example, **@{ $cube[$x][$y] }**, which is harder to read.

The Schwartzian Transform

I think you're ready to learn an elegant optimization for sorting certain lists. The *Schwartzian Transform* is named after famed Perl guru, Randal Schwartz. It reduces the time required for sorting. You only use it when you're pressed to save time sorting, but really, you'd save even more time by using a CPAN module like Sort::Maker with the Guttman-Rosler Transform. In any case, you don't want to get into anything that complicated unless it's absolutely necessary.

So then, if this isn't the preferred way to sort, why am I showing it to you? Because the Schwartzian Transform is an excellent example of thinking through Perl, in this case, using references and **map()** together. Randal came up with it extemporaneously. With enough practice, you'll be able to come up with time-saving constructions of your own when you need them.

We can't always sort things in our programs just by passing them to **sort()**. Some things don't lend themselves to being sorted, like dates:

OBSERVE: Dates

```
1/10/10
4/9/08
12/31/99
5/23/09
12/17/07
```

Those aren't plain numbers, and sorting them as strings doesn't give us what we want either. Create **sort.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

chomp( my @dates = <DATA> );
print "Before:\n", map { "\t$_\n" } @dates;
@dates = sort @dates;
print "After:\n", map { "\t$_\n" } @dates;

__END__
1/10/10
4/9/08
12/31/99
5/23/09
12/17/07
```

Check Syntax and run it:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./sort.pl
Before:
        1/10/10
        4/9/08
        12/31/99
        5/23/09
        12/17/07
After:
        1/10/10
        12/17/07
        12/31/99
        4/9/08
        5/23/09
```

We know how we want to sort those dates, but we have to explain it to the computer. So let's modify **sort.pl** like this:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

chomp( my @dates = <DATA> );
print "Before:\n", map { "\t$_\n" } @dates;
@dates = sort bydate @dates;
print "After:\n", map { "\t$_\n" } @dates;

sub bydate
{
  date2str($a) cmp date2str($b);
}

sub date2str  # 4/9/08 -> 20080409
{
  my $date = shift;
  my ($month, $day, $year) = split m!/!, $date;
  $year += $year > 50 ? 1900 : 2000;
  return sprintf "%d%02d%02d", $year, $month, $day
}
__END__
1/10/10
4/9/08
12/31/99
5/23/09
12/17/07
```

Check Syntax and run it:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./sort.pl
Before:
        1/10/10
        4/9/08
        12/31/99
        5/23/09
        12/17/07
After:
        12/31/99
        12/17/07
        4/9/08
        5/23/09
        1/10/10
```

The **date2str** routine converts the date into a form that will sort in the order that we want. But let's think about how sorting happens; the *comparison function* (in this case, **bydate**) gets called by Perl's **sort()** function every time it needs to compare two elements to determine whether they're in the right order as it builds the result. **sort()** is going to call that comparison function more than once for at least some of the dates (technically, an optimum sort function requires *n log n* comparisons). But the comparison function has to call the *conversion* function **date2str**, which means that **date2str** gets called more than once with the same input, which is a waste, because it's going to produce the same output each time.

The Schwartzian Transform *caches* the results of calling the conversion routine for every item in the input list, uses those cached values for the sort, then recovers the original list values. With our example using dates, the conversion routine isn't particularly expensive; it doesn't eat up time and resources. But if the input list was something that required going to the system or the network, like looking up file modification times or doing a DNS query, then this conversion routine would become more of a concern.

Let's apply the Schwartzian Transform to our program. Modify **sort.pl** as shown:

CODE TO TYPE:

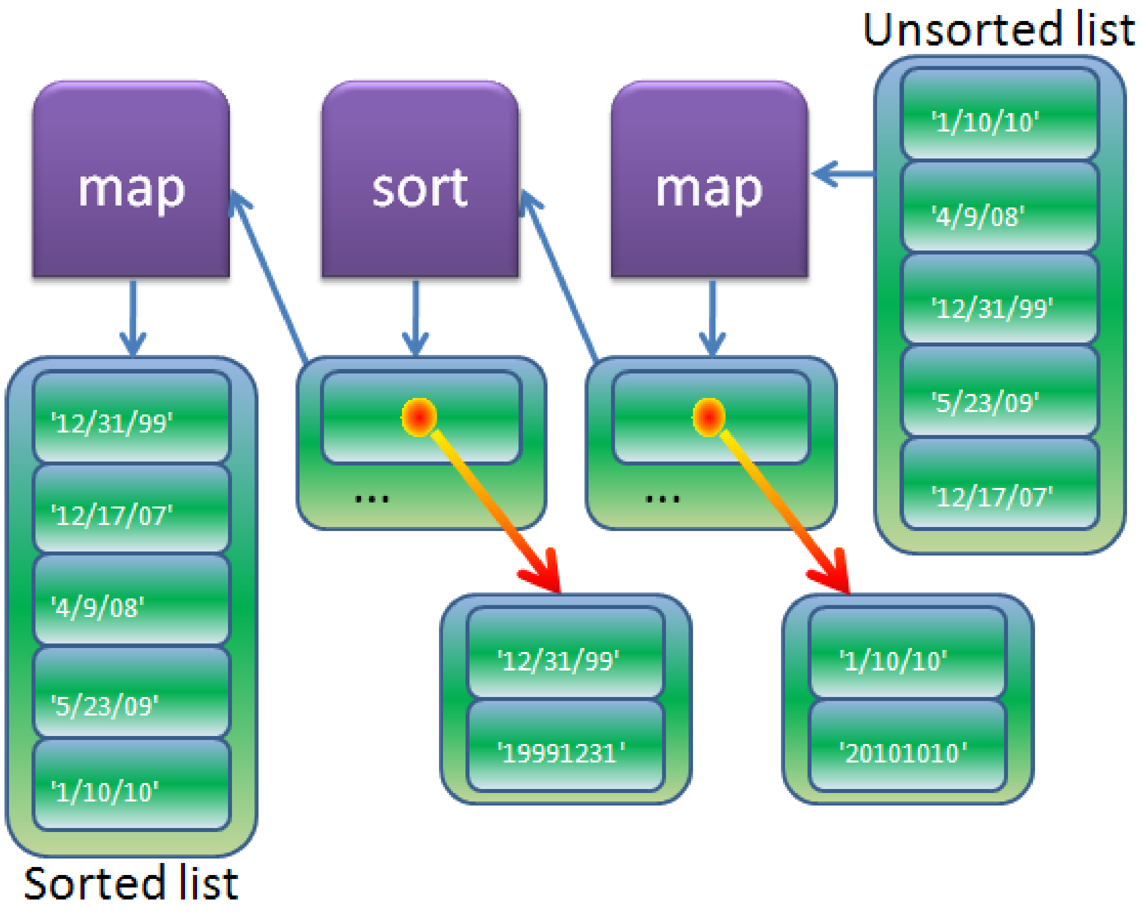
```
#!/usr/bin/perl
use strict;
use warnings;

chomp( my @dates = <DATA> );
print "Before:\n", map { "\t$_\n" } @dates;
~~@dates = sort bydate @dates;~~
@dates = map  { $_->[0] }
         sort { $a->[1] cmp $b->[1] }
         map  { [ $_, date2str($_) ] } @dates;

print "After:\n", map { "\t$_\n" } @dates;

~~sub bydate~~
~~{~~
~~  date2str($a) cmp date2str($b);~~
~~}~~

sub date2str  # 4/9/08 -> 20080409
{
  my $date = shift;
  my ($month, $day, $year) = split m!/!, $date;
  $year += $year > 50 ? 1900 : 2000;
  return sprintf "%d%02d%02d", $year, $month, $day
}
__END__
1/10/10
4/9/08
12/31/99
5/23/09
12/17/07
```

Check Syntax and run it. You get exactly the same result as before. How does this work? Take a look at the Schwartzian Transform in action:

OBSERVE:

```
@dates = map  { $_->[0] }
        sort { $a->[1] cmp $b->[1] }
        map  { [ $_, date2str($_) ] } @dates;
```

Remember the conveyor belt metaphor we saw earlier? It illustrated the way list items flow from right to left. So, elements come from **@dates** and go into the first **map()** statement, which turns them into an arrayref pointing to two elements: the **original element**, and the result of calling **date2str** on it. Then the conveyor belt moves things through the **sort()** function; in its comparison block each **$a** and **$b** is an arrayref, and we can compare them by dereferencing the second element of each one (the result of calling **date2str**). Then, the list of arrayrefs (sorted in the correct order) passes into the final **map()** statement, which extracts the first element from each arrayref, which is the original (date) element. Here's a picture of that process:



Most of the transform routine remains the same no matter what is being sorted, that's where our program is saving time.

Study this example, play around, and have fun with it! We're getting into some really deep Perl here, and you're getting better and better at data manipulation. Great work so far!

Once you finish the lesson, go back to the syllabus to complete the homework.

Hashes of Hashes

Lesson Objectives

When you complete this lesson, you will be able to:

- Represent Data with Hashes of Hashes.
- explain and use Autovivification.

Welcome back!

Representing Data with Hashes of Hashes

In this lesson, we will look at how hashes of hashes can be used to represent data in the same way that a relational database does.

We've used hashes before to store data in much the same way that a table stores data in a relational database. For example, a table of names of US states with their abbreviations would look like this:

Abbreviation	Name
AK	Alaska
AL	Alabama
AR	Arkansas
...	...

And a hash of that data could be defined like this:

OBSERVE:

```
%name = ( AK => 'Alaska', AL => 'Alabama', AR => 'Arkansas', ...)
```

This works fine until we encounter a table that has more than one column in addition to the primary key:

Abbreviation	Name	Capital
AK	Alaska	Juneau
AL	Alabama	Montgomery
AR	Arkansas	Little Rock
...	...	...

In that case we need two hashes:

OBSERVE:

```
%name = ( AK => 'Alaska', AL => 'Alabama', AR => 'Arkansas', ...)
%capital = ( AK => 'Juneau', AL => 'Montgomery', AR => 'Little Rock', ...)
```

I know that this is deeply unsatisfying to you. The code is repetitive and fragile, and it requires you to keep abbreviations synchronized between the two lists. And even worse, the names of the dependent columns (or attributes) are embedded not in data, but in the variable names themselves. This is the just the wrong approach. Programmers post this kind of code in forums all the time though, asking how to access a variable when another variable already contains its name. But we know better! We'll use references instead. Here's how it's done. Create **state_short.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %state = ( AK => { name    => 'Alaska',
                      capital => 'Juneau'
                    },
              AL => { name    => 'Alabama',
                      capital => 'Montgomery',
                    },
              AR => { name    => 'Arkansas',
                      capital => 'Little Rock',
                    },
            );

print "The capital of $state{AR}{name} is $state{AR}{capital}.\n";
```

Check Syntax and run it by typing the commands below as shown:

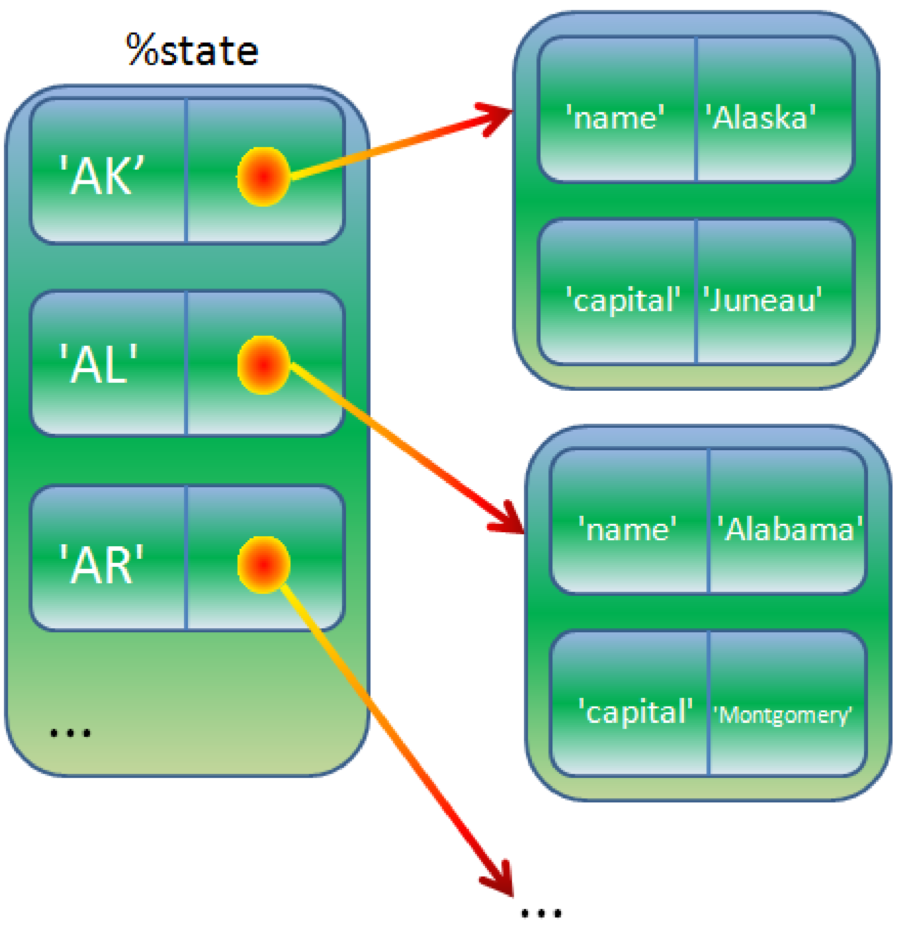
INTERACTIVE SESSION:

```
cold:~$ cd perl3
cold:~/perl3$ ./state_short.pl
The capital of Arkansas is Little Rock.
```

This is the hash equivalent of **@board** and an array of array refs. The keys in the top-level hash **%state** are what we would call as the primary keys in a database table. The values are hashrefs whose keys are the column names and its values are the column values.

We have used the syntactic sugar that allows us to omit the arrow in **{AR}->{capital}**; that rule works for any combination of brackets and braces.

That data structure looks like this:



Now let's try a more realistic example that reads in the data from an external file, **state.data**, which is included with your lesson files.

Create **state.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %state;
while ( <> )
{
  chomp;
  my ($abbreviation, $name, $capital) = split /\s*\|\s*/;
  next if $abbreviation eq 'key';
  $state{$abbreviation} = { name => $name, capital => $capital };
}

my $limit;
for my $abbr ( sort keys %state )
{
  print "$abbr: $state{$abbr}{name}'s capital is $state{$abbr}{capital}\n";
  exit if ++$limit >= 5;
}
```

Check Syntax and run it with **state.data** as input:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./state.pl /software/Perl3/state.data
AK: Alaska's capital is Juneau
AL: Alabama's capital is Montgomery
AR: Arkansas's capital is Little Rock
AZ: Arizona's capital is Phoenix
CA: California's capital is Sacramento
```

(In order to save screen space, we printed only the first five entries.) The loop shows you how we iterate through a hash-of-hashes, but it presumes that we know the keys of the second-level hashes. Even though the same code sets them a few lines earlier, let's pretend we didn't have that information and see what happens. Modify the code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %state;
while ( <> )
{
  chomp;
  my ($abbreviation, $name, $capital) = split /\s*\|\s*/;
  next if $abbreviation eq 'key';
  $state{$abbreviation} = { name => $name, capital => $capital };
}

my $limit;
for my $abbr ( sort keys %state )
{
  ~~print "$abbr: $state{$abbr}{name}'s capital is $state{$abbr}{capital}\n";~~
  print "$abbr: ";
  for my $key ( sort keys %{ $state{$abbr} } )
  {
    print "$key => $state{$abbr}{$key}, ";
  }
  print "\n";
  exit if ++$limit >= 5;
}
```

Check Syntax and run it:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./state.pl /software/Perl3/state.data
AK: capital => Juneau, name => Alaska,
AL: capital => Montgomery, name => Alabama,
AR: capital => Little Rock, name => Arkansas,
AZ: capital => Phoenix, name => Arizona,
CA: capital => Sacramento, name => California,
```

If you were dealing with a **sparse data structure** instead of a fully-populated table-equivalent, that approach would make sense. (One example of a sparse data structure would be a record in an LDAP directory—Lightweight Directory Access Protocol, is an internet protocol that email and other programs use to look up information from a server.) In such a data structure, there would be many keys that didn't have values in most records.

It might also make sense if the *schema* for the database was defined outside the program. In fact, the first line of the state.data file contains that schema—we've just been ignoring it. Let's modify the program now to use it:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %state;
my @names;
while ( <> )
{
  chomp;
  ~~my ($abbreviation, $name, $capital) = split /\s*\|\s*/;~~
  my ($abbreviation, @fields) = split /\s*\|\s*/;
  @names = @fields and next if $abbreviation eq 'key';
  my %data;
  @data{@names} = @fields;
  $state{$abbreviation} = \%data;
  ~~$state{$abbreviation} = { name => $name, capital => $capital };~~
}

my $limit;
for my $abbr ( sort keys %state )
{
  print "$abbr: ";
  for my $key ( sort keys %{ $state{$abbr} } )
  {
    print "$key => $state{$abbr}{$key}, ";
  }
  print "\n";
  exit if ++$limit >= 5;
}
```

Check Syntax and run it. You'll get exactly the same result. Now try running it on the **state_full.data** file:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./state.pl /software/Perl3/state_full.data
AK: bird => Willow Ptarmigan, capital => Juneau, flower => Forget-me-not, largest_city
=> Anchorage, name => Alaska,
AL: bird => Yellowhammer, capital => Montgomery, flower => Camellia, largest_city => Bi
rmingham, name => Alabama,
AR: bird => Mockingbird, capital => Little Rock, flower => Apple Blossom, largest_city
=> Little Rock, name => Arkansas,
AZ: bird => Cactus Wren, capital => Phoenix, flower => Saguaro Cactus Blossom, largest_
city => Phoenix, name => Arizona,
CA: bird => California Valley Quail, capital => Sacramento, flower => Golden Poppy, lar
gest_city => Los Angeles, name => California,
```

Because that data file is in the same format, we can now read it in without changing the program! That's impressive! This technique will let us do some serious data processing.

See how we used the **and** operator to get two things done in a conditional without having to create a block and an **if** statement? **@names = @fields and next if ...** will always execute the **next** statement because there will always be something in **@fields**, so the result of the assignment will be true (because it is a list assignment and the result will be the number of things assigned).

Also, we used the *hash slice* to set the hash **%data**. To be even more concise, you can slice using the reference itself—this is generally not recommended for the faint of heart, but hey, this *is Advanced* Perl! Let's give it a try. Modify **state.pl** like this:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %state;
my @names;
while ( <> )
{
  chomp;
  my ($abbreviation, @fields) = split /\s*\|\s*/;
  @names = @fields and next if $abbreviation eq 'key';
~~  my %data;~~
~~  @data{@names} = @fields;~~
~~  $state{$abbreviation} = \%data;~~
  @{ $state{$abbreviation} }{@names} = @fields;
}

my $limit;
for my $abbr ( sort keys %state )
{
  print "$abbr: ";
  for my $key ( sort keys %{ $state{$abbr} } )
  {
    print "$key => $state{$abbr}{$key} ";
  }
  print "\n";
  exit if ++$limit >= 5;
}
```

Check Syntax and run it; the output will be identical. You can see that this is an application of *Rule 2*.

Autovivification

Now let's learn a ten-dollar word that will impress our friends and our boss. The next time you're having a conversation, toss in, "I see your program is abusing autovivification." Provided you have used it correctly, they'll be in awe of your expertise.

The best way to explain Autovivification is to show it in action. Create **auto.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

use Data::Dumper;

my %state;

$state{AK}{capital} = 'Juneau';
print Dumper \%state;
```

It may look like it shouldn't work, but just go with it. Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./auto.pl
$VAR1 = {
          'AK' => {
                    'capital' => 'Juneau'
                  }
        };
```

Most other languages wouldn't let you get away with this—they'd demand that you create an inner hash in order to populate it. But not Perl, and that's a real advantage. In Perl you don't need to create the inner hash in order to populate it; instead, when Perl sees the assignment to **$state{AK}{capital}** this is what happens:

- It replaces the arrow we omitted through syntactic sugar: **$state{AK}->{capital}**
- It looks at the right side of the arrow and sees braces, which means they must enclose a hash key and the left side of the arrow must be a hash reference.
- **$state{AK}** doesn't exist, but throwing an exception wouldn't be particularly useful, so in this situation, Perl *creates* **$state{AK}**, knowing that it must contain a reference to a hash.
- Then Perl populates **$state{AK}** with a reference to an empty anonymous hash.
- Finally, Perl populates that anonymous hash with the key **capital** and the value **Juneau** as coded.

So there you have the principle of autovivification: if you use a nonexistent data slot (array or hash element) in a context that assumes it is a reference to something, Perl will first populate it with a reference to an empty version of that type of thing.

In case you didn't notice, the last change we made to **state.pl** used autovivification; the first element in the list implied by the slice will be assigned through a top-level key that hasn't been created yet. It could also be done with a loop:

OBSERVE:

```
$state{$abbreviation}{$_} = shift @fields for @names;
```

But that's a bit harder to read, because you have to puzzle out why one array is being **shift**ed and the other isn't.

So now you're thinking, "Okay, the data structure we've used in this lesson *does* hold all the data together cohesively; but it repeats the keys **name**, **capital**, and other stuff, unnecessarily." Good thinking! We'll look at ways to improve on *that* in the next lesson!

Once you finish the lesson, go back to the syllabus to complete the homework.

Heterogeneous Data Structures

Lesson Objectives

When you complete this lesson, you will be able to:

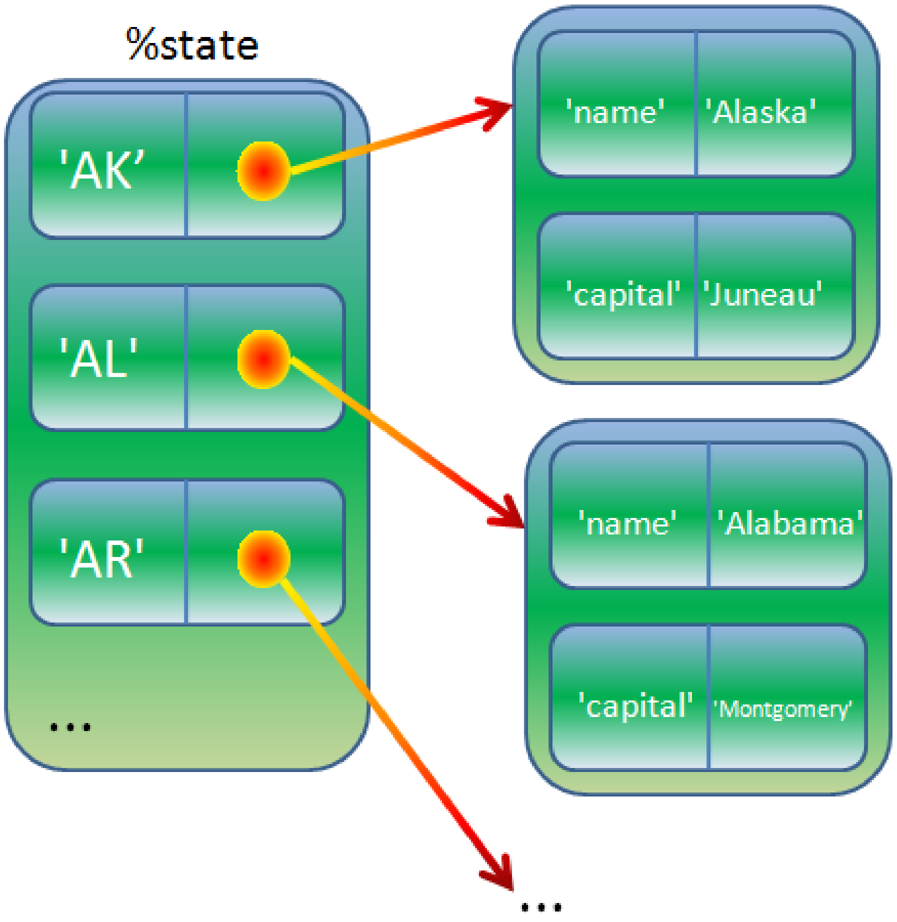
- handle Semi-Regular Data Structures.
- handle Irregular Data Structures.

We've seen arrays of arrays and hashes of hashes (okay, well technically, arrays of arrayrefs and hashes of hashrefs). Our data structures have been consistently regular because we designed them that way. But Perl really has no problem with more complex arrays that contain references to arrays, hashes, scalars, and such. And sometimes that's exactly what we want. This lesson is all about data structures that are not so regular.

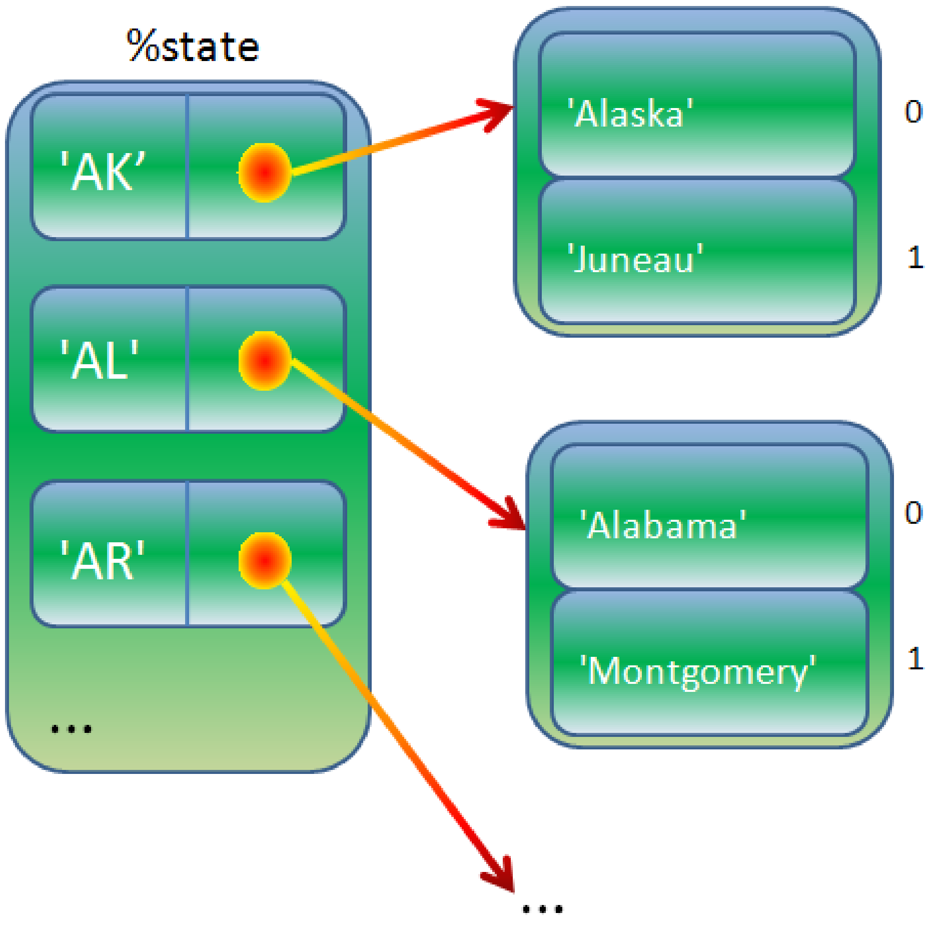
Semi-Regular Data Structures

Hashes of Arrays

Remember our data structure for the state database from the last lesson? It looked like this:



Unfortunately, it repeats the storage for the keys' **name**, **capital**, and so on, in every record. But, since this isn't a sparse dataset, we can shrink the storage required by using a hash of arrays:



Each entry in the hash is an arrayref containing the elements in a predetermined order. Let's write the code for that. Modify **state.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %state;
my @names;
while ( <> )
{
  chomp;
  my ($abbreviation, @fields) = split /\s*\|\s*/;
  @names = @fields and next if $abbreviation eq 'key';
  ~~@{ $state{$abbreviation} }{@names} = @fields;~~
  $state{$abbreviation} = \@fields;
}

my $limit;
for my $abbr ( sort keys %state )
{
  print "$abbr: ";
  my @values = @{ $state{$abbr} };
  ~~for my $key ( sort keys %{ $state{$abbr} } )~~
  for my $key ( @names )
  {
    ~~print "$key => $state{$abbr}{$key} ";~~
    print "$key => ", shift @values, ", ";
  }
  print "\n";
  exit if ++$limit >= 5;
}
```

Check Syntax and run it again on the **state_full.data** input file:

INTERACTIVE SESSION:

```
cold:~$ cd perl3
cold:~/perl3$ ./state.pl ./state_full.data
AK: bird => Willow Ptarmigan, capital => Juneau, flower => Forget-me-not, larges
t_city => Anchorage, name => Alaska,
AL: bird => Yellowhammer, capital => Montgomery, flower => Camellia, largest_cit
y => Birmingham, name => Alabama,
AR: bird => Mockingbird, capital => Little Rock, flower => Apple Blossom, larges
t_city => Little Rock, name => Arkansas,
AZ: bird => Cactus Wren, capital => Phoenix, flower => Saguaro Cactus Blossom, l
argest_city => Phoenix, name => Arizona,
CA: bird => California Valley Quail, capital => Sacramento, flower => Golden Pop
py, largest_city => Los Angeles, name => California,
```

You get exactly the same output as the last time you ran it.

The parsing loop looks for the header line and saves the column headings from it into the array **@names**. Unlike the hash-of-hashes version of this program, this one would actually work if the header line wasn't the first line.

When we print out the data structure, we copy the array pointed to by each arrayref and shift each successive element out while we iterate through the parallel array of field names (**@names**).

Now let's exercise our **map()** skills while we remove the unwanted trailing commas from the end of each line. Modify **state.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %state;
my @names;
while ( <> )
{
  chomp;
  my ($abbreviation, @fields) = split /\s*\|\s*/;
  @names = @fields and next if $abbreviation eq 'key';
  $state{$abbreviation} = \@fields;
}

my $limit;
for my $abbr ( sort keys %state )
{
  print "$abbr: ";
  my @values = @{ $state{$abbr} };
  print join( ', ', map { "$_ => " . shift @values } @names ), ".\n";
  ~~for my $key ( @names )~~
~~  {~~
~~    print "$key => ", shift @values, ", ";~~
~~  }~~
~~  print "\n";~~
  exit if ++$limit >= 5;
}
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./state.pl ./state_full.data
AK: name => Alaska, capital => Juneau, largest_city => Anchorage, bird => Willow
 Ptarmigan, flower => Forget-me-not.
AL: name => Alabama, capital => Montgomery, largest_city => Birmingham, bird =>
Yellowhammer, flower => Camellia.
AR: name => Arkansas, capital => Little Rock, largest_city => Little Rock, bird
=> Mockingbird, flower => Apple Blossom.
AZ: name => Arizona, capital => Phoenix, largest_city => Phoenix, bird => Cactus
 Wren, flower => Saguaro Cactus Blossom.
CA: name => California, capital => Sacramento, largest_city => Los Angeles, bird
 => California Valley Quail, flower => Golden Poppy.
```

The output is slightly improved. But what if we want to access specific named attributes from our code? We can no longer write **print $state{AK}{capital}** because we no longer have a hash of hashes. We'd need to write **print $state{AK}[1]** and that, to use a non-technical term, sucks. We can only tell that the index required is **1** by looking at the data file heading line, remembering that the first column doesn't count, and hoping that it doesn't change. Using code that depends on the format of external data is a bad idea. The code didn't care about the ordering of the data before, why should it now?

One solution to this problem would be to capture the positional information in a separate hash. Modify the program like this:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %state;
my @names;
while ( <> )
{
  chomp;
  my ($abbreviation, @fields) = split /\s*\|\s*/;
  @names = @fields and next if $abbreviation eq 'key';
  $state{$abbreviation} = \@fields;
}
my $count = 0;
my %index = map { $_ => $count++ } @names;
my ($attr, $abbr) = qw(capital AR);
print "The $attr of $state{$abbr}[ $index{name} ] is $state{$abbr}[ $index{$attr
} ]\n";
exit;

my $limit;
for my $abbr ( sort keys %state )
{
  print "$abbr: ";
  my @values = @{ $state{$abbr} };
  print join( ', ', map { "$_ => " . shift @values } @names ), ".\n";
  exit if ++$limit >= 5;
}
```

Check Syntax and run it (we inserted the **exit()** as a quick way to suppress the loop output that we're not interested in at the moment, while not losing the original loop code):

INTERACTIVE SESSION:

```
cold:~/perl3$ ./state.pl ./state_full.data
The capital of Arkansas is Little Rock
```

We could make things a little easier for ourselves with special index variables. Modify **state.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %state;
my @names;
while ( <> )
{
  chomp;
  my ($abbreviation, @fields) = split /\s*\|\s*/;
  @names = @fields and next if $abbreviation eq 'key';
  $state{$abbreviation} = \@fields;
}
my $count = 0;
my %index = map { $_ => $count++ } @names;
my $NAME = $index{name};
my ($attr, $abbr) = qw(capital AR);
print "The $attr of $state{$abbr}[ ~~$index{name}~~$NAME ] is $state{$abbr}[ $index{
$attr} ]\n";
exit;

my $limit;
for my $abbr ( sort keys %state )
{
  print "$abbr: ";
  my @values = @{ $state{$abbr} };
  print join( ', ', map { "$_ => " . shift @values } @names ), ".\n";
  exit if ++$limit >= 5;
}
```

Check Syntax and run it; the output is identical.

When we have a small number of fixed data (no more than three items) to pass around a program, an arrayref is often a good programming option. And in order to use an arrayref, you should have some way of keeping track of the specific location of each element, like we've done in our example with the hash of indexes and variables containing indexes.

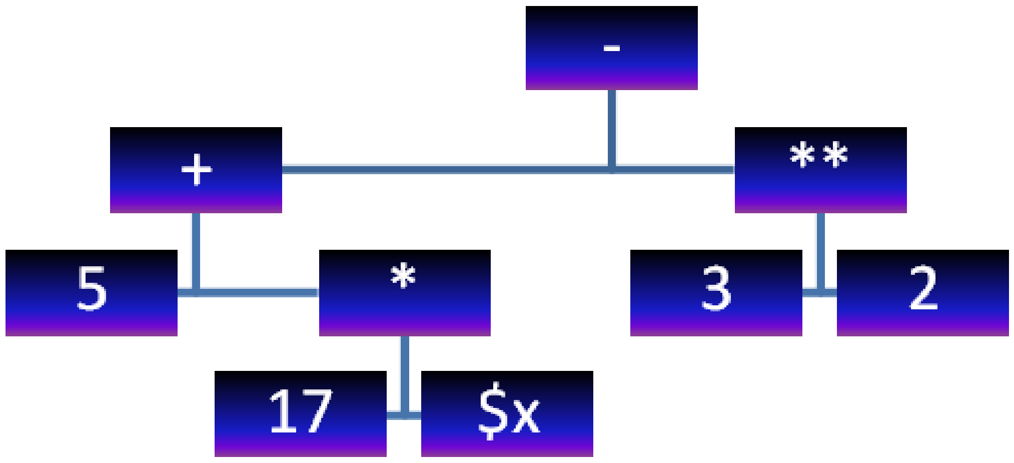
Imagine how we could use other kinds of semi-regular data structures—like arrays of hashrefs—for various tasks. There are countless possibilities!

Irregular Data Structures

Sometimes, you need a data structure that doesn't have any regularity to it at all; say, an array that might have a hashref in one element, an arrayref in another, an ordinary scalar in another, and so on, even repeated down through multiple levels. One example of such a data structure is a *tree.*

Parse Trees

Consider an arithmetic expression like **5 + 17 * $x - 3 ** 2**. When an expression like that appears in a computer program of some kind, the compiler parses it into a tree that can later be executed to figure out its value. A tree for that expression would look like this:



Each node contains either an operation to be performed on its children, or a value to perform an operation on. Let's write a parser like that now. Create **parse.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

use Data::Dumper;

my (@numbers, @operators);

my $exp = join '', <DATA>;
chomp $exp;

my $tree = parse( $exp );
print Dumper $tree;

my %VARIABLES = ( '$x' => 8 );

my $result = evaluate( $tree );

$exp =~ s/\Q$_\E/$VARIABLES{$_}/ for keys %VARIABLES;
print "$exp = $result\n";

sub parse
{
  local $_ = shift;

  if ( my ($left, $op, $right) = /(.+)([+-])(.+)/ )
  {
    return make_node( OP => parse($left), $op, parse($right) );
  }
  elsif ( ($left, $op, $right) = m!(.*[^*])([*/])([^*].*)! )
  {
    return make_node( OP => parse($left), $op, parse($right) );
  }
  elsif ( ($left, $op, $right) = /(.+)(\*\*)(.+)/ )
  {
    return make_node( OP => parse($left), $op, parse($right) );
  }
  elsif ( my ($const) = /\A\s*(\d+)\s*\z/ )
  {
    return make_node( CONST => $const );
  }
  elsif ( my ($var) = /\A\s*(\$\w+)\s*\z/ )
  {
    return make_node( VAR => $var );
  }
  else
  {
    die "Error parsing '$_'\n";
  }
}

sub make_node
{
  my $type = shift;

  if ( $type =~ /CONST|VAR/ )
  {
    return { type => $type, value => shift };
  }

  my ($left, $op, $right) = @_;
  return { type => $type, left => $left, right => $right, op => $op };
}

sub evaluate
{
  my $node = shift;

  if ( $node->{type} eq 'CONST' )
  {
    return $node->{value};
  }
  elsif ( $node->{type} eq 'VAR' )
  {
    return $VARIABLES{ $node->{value} };
  }
  elsif ( $node->{type} eq 'OP' )
  {
    my ($left, $right) = map { evaluate( $node->{$_} ) } qw(left right);
    if ( $node->{op} eq '+' )
    {
      return $left + $right;
    }
    elsif ( $node->{op} eq '-' )
    {
      return $left - $right;
    }
    elsif ( $node->{op} eq '*' )
    {
      return $left * $right;
    }
    elsif ( $node->{op} eq '/' )
    {
      return $left / $right;
    }
    elsif ( $node->{op} eq '**' )
    {
      return $left ** $right;
    }
    die "Unknown operator $node->{op}\n";
  }
  else
  {
    die "Unknown node type $node->{type}\n";
  }
}
__END__
5 + 17 * $x - 3 ** 2
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./parse.pl
$VAR1 = {
          'left' => {
                      'left' => {
                                  'value' => '5',
                                  'type' => 'CONST'
                                },
                      'right' => {
                                   'left' => {
                                               'value' => '17',
                                               'type' => 'CONST'
                                             },
                                   'right' => {
                                                'value' => '$x',
                                                'type' => 'VAR'
                                              },
                                   'type' => 'OP',
                                   'op' => '*'
                                 },
                      'type' => 'OP',
                      'op' => '+'
                    },
          'right' => {
                       'left' => {
                                   'value' => '3',
                                   'type' => 'CONST'
                                 },
                       'right' => {
                                    'value' => '2',
                                    'type' => 'CONST'
                                  },
                       'type' => 'OP',
                       'op' => '**'
                     },
          'type' => 'OP',
          'op' => '-'
        };
5 + 17 * 8 - 3 ** 2 = 132
```

Those regular expressions have made parsing considerably easier, but there's still a lot to explain here! Let's go over some of the more basic and incidental pieces first. We use the word **local** to create a scoped copy of **$_** so that our recursive subroutine doesn't stomp on other versions of **$_** used by subroutines higher up the calling stack. Remember, **$_** belongs to Perl, not us, so we wouldn't say "my $_". We use **$_** inside the **parse** subroutine to avoid having to bind the matching statements to a variable. This keeps them more concise.

Also, the **\Q...\E** construction inside a regular expression is the quoting form of the **quotemeta()** function (for more information on quotemeta, visit perldoc -f quotemeta). It causes the expansion of **$_** inside the regex to have special characters quoted, so that when **$_** is equal to **$x**, it is interpolated in the regex as **\$x**, which we need, because there is no variable **$x** in our program. We do this substitution so that when the input string is printed out for verification, it has any "variables" substituted already. This makes it easier to paste into a calculator to determine whether the calculated answer was correct.

The program itself works in two phases: parse and execute. First, we construct a parse tree for the expression, then we print the tree with **Dumper()**, and then we evaluate the tree with **evaluate** and print the result.

The parser constructs a binary tree by looking for the lowest-precedence operators first: **+** and **-**. It splits the expression into what's on the left of that operator and what's on the right, and builds a "node" (represented with a hashref) that contains pointers to the left and right parts.

Then the parser looks for higher-precedence operators, and finally it looks for numbers or "variables" (which for the sake of convenience are not actual variables in the program, but represented in the special hash **%VARIABLES**).

The recursion (the application of a function as a part of the definition of that same function) can be hard to follow at first, but the structure printed by **Dumper()** shows you the result.

The evaluation recurses through the same tree, extracting values and variables and applying operators to them.

Take the time to study this program and how it works. Throw other expressions at it. This program can only deal with the operators **+**, **-**, *****, **/**, and ******—that last one requires the regex looking for ***** to ensure that it finds only one *****. It can handle integers and "variables" that have values in **%VARIABLES**. Anything else will cause a failure.

The ref() Function

For all its depth and tree-ness, the tree that we built in the last example contains only hashrefs. Suppose it were more heterogeneous and contained other reference types; for example, suppose we decided to save space when building our tree and put integers in as ordinary scalars instead of inside hashrefs. How would we indicate inside the **evaluate** routine, the way to dereference **$node**?

This is where we bring in the **ref()** function. It indicates the type of reference you pass it:

Argument to ref()	Result
Not a reference	False (empty string)
Reference to a scalar	'SCALAR'
Reference to an array	'ARRAY'
Reference to a hash	'HASH'

Let's see that in action. Modify **parse.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

use Data::Dumper;

my (@numbers, @operators);

my $exp = join '', <DATA>;
chomp $exp;

my $tree = parse( $exp );
print Dumper $tree;

my %VARIABLES = ( '$x' => 8 );

my $result = evaluate( $tree );

$exp =~ s/\Q$_\E/$VARIABLES{$_}/ for keys %VARIABLES;
print "$exp = $result\n";

sub parse
{
  local $_ = shift;

  if ( my ($left, $op, $right) = /(.+)([+-])(.+)/ )
  {
    return make_node( OP => parse($left), $op, parse($right) );
  }
  elsif ( ($left, $op, $right) = m!(.*[^*])([*/])([^*].*)! )
  {
    return make_node( OP => parse($left), $op, parse($right) );
  }
  elsif ( ($left, $op, $right) = /(.+)(\*\*)(.+)/ )
  {
    return make_node( OP => parse($left), $op, parse($right) );
  }
  elsif ( my ($const) = /\A\s*(\d+)\s*\z/ )
  {
    return make_node( CONST => $const );
  }
  elsif ( my ($var) = /\A\s*(\$\w+)\s*\z/ )
  {
    return make_node( VAR => $var );
  }
  else
  {
    die "Error parsing '$_'\n";
  }
}


sub make_node
{
  my $type = shift;

  ~~if ( $type =~ /CONST|VAR/ )~~
~~  {~~
~~    return { type => $type, value => shift };~~
~~  }~~
  if ( $type eq 'CONST' )
  {
    return shift;
  }
  elsif ( $type eq 'VAR' )
  {
    return [ shift ];
  }

  my ($left, $op, $right) = @_;
  return { type => $type, left => $left, right => $right, op => $op };
}


sub evaluate
{
  my $node = shift;

  if ( ! ref $node )
  {
    return $node;
  }
  elsif ( ref $node eq 'ARRAY' )
  {
    return $VARIABLES{ $node->[0] };
  }
  ~~if ( $node->{type} eq 'CONST' )~~
~~  {~~
~~    return $node->{value};~~
~~  }~~
~~  elsif ( $node->{type} eq 'VAR' )~~
~~  {~~
~~    return $VARIABLES{ $node->{value} };~~
~~  }~~
  elsif ( $node->{type} eq 'OP' )
  {
    my ($left, $right) = map { evaluate( $node->{$_} ) } qw(left right);
    if ( $node->{op} eq '+' )
    {
      return $left + $right;
    }
    elsif ( $node->{op} eq '-' )
    {
      return $left - $right;
    }
    elsif ( $node->{op} eq '*' )
    {
      return $left * $right;
    }
    elsif ( $node->{op} eq '/' )
    {
      return $left / $right;
    }
    elsif ( $node->{op} eq '**' )
    {
      return $left ** $right;
    }
    die "Unknown operator $node->{op}\n";
  }
  else
  {
    die "Unknown node type $node->{type}\n";
  }
}
__END__
5 + 17 * $x - 3 ** 2
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./parse.pl
$VAR1 = {
          'left' => {
                      'left' => '5',
                      'right' => {
                                   'left' => '17',
                                   'right' => [
                                                '$x'
                                              ],
                                   'type' => 'OP',
                                   'op' => '*'
                                 },
                      'type' => 'OP',
                      'op' => '+'
                    },
          'right' => {
                       'left' => '3',
                       'right' => '2',
                       'type' => 'OP',
                       'op' => '**'
                     },
          'type' => 'OP',
          'op' => '-'
        };
5 + 17 * 8 - 3 ** 2 = 132
```

In this program, we also changed the representation of "variables" to an arrayref just to see what that looks like.

Study the output from **Dumper()** and see how it has changed. Then, look at the changes in the program code and see how they created that tree and parsed it.

ref() looked at the value returned by **ref()**. Most programs that use **ref()** in practice just test whether the result is true or false; that is, whether it's a reference or not. That would have been the case in our last example if we weren't storing an arrayref in the tree on occasion there.

References to Lists

Here's what happens when you take a reference to a list:

OBSERVE:

```
\(1, 2, 3, $x)
```

You don't get an arrayref and you don't get some special kind of reference that we've never seen before. You get the list of references to the individual elements, just as though you had written this code:

OBSERVE:

```
(\1, \2, \3, \$x)
```

Good work! That's enough about less-than-regular data structures—for now. In the next lesson we'll tackle references to subroutines and more! See you there...

Once you finish the lesson, go back to the syllabus to complete the homework.

References to Subroutines

Lesson Objectives

When you complete this lesson, you will be able to:

- apply Coderefs.
- use the Closure subroutine.
- utilize a few other miscellaneous Code Reference behaviors.

The information about references just keeps coming! You're doing well to keep up with all of it. Good job!

Coderefs

One of the most interesting types of reference that you can have is to a *subroutine*. That may seem a bit mind-boggling to you at first. I mean, how can *code* be the same as *data*? Well, coderefs are an unusual type of reference. Let's dig in and see how coderefs work!

References to Named Subroutines

In order to create a reference to a subroutine, you type a backslash and then an ampersand before the name of the subroutine. The syntax goes like this:

OBSERVE:

```
$code_ref = \&subroutine_name
```

We would apply that syntax like this:

OBSERVE:

```
sub square
{
  my $arg = shift;

  return $arg ** 2;
}

my $square_ref = \&square;
```

Without that ampersand, Perl would think you were *calling* the routine **square**, and wanted to take a reference to the results—you actually saw this happen in one of the homework questions in the previous lesson. If you're wondering why an ampersand, it's possible to *call* a subroutine by putting an ampersand in front of its name, but that's not considered good practice.

To dereference, we use the arrow operator. We use square brackets after an arrow for an arrayref, curly braces after an arrow for a hashref, so what do you suppose comes after an arrow for a coderef? We use parentheses, which contain any arguments to pass to the subroutine:

OBSERVE:

```
print $square_ref->( 42 );   # 1764
```

Now try it out for yourself. Create **area.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $triangle_ref = \&triangle;
my $square_ref   = \&square;
my $pentagon_ref = \&pentagon;

my $side = shift or die;  # Length
print "Area of triangle of side $side = ", $triangle_ref->( $side ), "\n";
print "Area of square of side   $side = ", $square_ref->( $side ), "\n";
print "Area of pentagon of side $side = ", $pentagon_ref->( $side ), "\n";

sub triangle  # Equilateral
{
  my $side = shift;

  return sqrt(3) / 4 * $side ** 2
}

sub square
{
  my $side = shift;

  return $side ** 2;
}

sub pentagon   # Regular
{
  my $side = shift;

  return sqrt(25 + 10 * sqrt(5)) / 4 * $side ** 2;
}
```

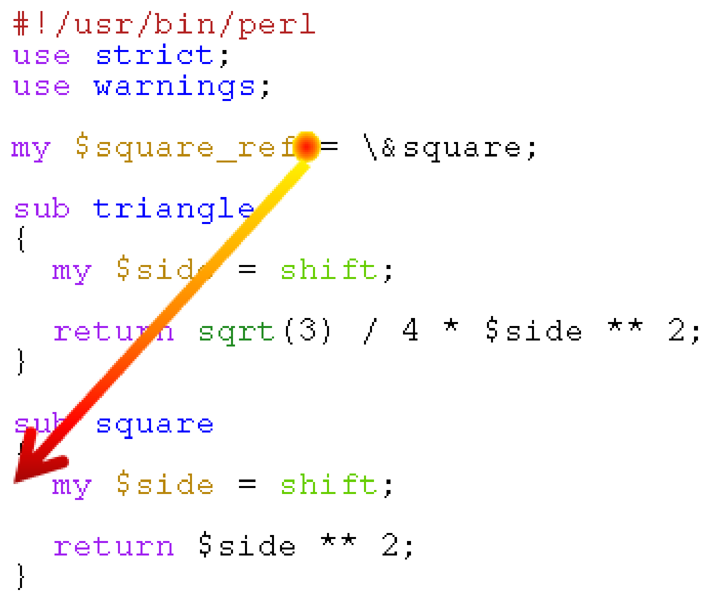
Check Syntax and run it by typing the commands below as shown:

INTERACTIVE SESSION:

```
cold:~$ cd perl3
cold:~/perl3$ ./area.pl 6
Area of triangle of side 6 = 15.5884572681199
Area of square of side 6 = 36
Area of pentagon of side 6 = 61.9371864212028
```

The subroutine does *not* need to appear in the program before your references to it, because Perl makes two passes through your program. It compiles subroutines on the first pass, and resolves references on the second.

Our next graphical representation of a code reference will be a bit different from those we've seen before. The code that the reference points to is in your program, not in some dynamically-allocated memory that gets created, deleted, expanded, or compressed while your program is running. Think of it like this:



```
#!/usr/bin/perl
use strict;
use warnings;

my $square_ref = \&square;

sub triangle
{
  my $side = shift;

  return sqrt(3) / 4 * $side ** 2;
}

sub square
{
  my $side = shift;

  return $side ** 2;
}
```

I intentionally drew the arrow to point at the contents of the subroutine **square**, and not at the word "square" itself. Just as with the other references, when using coderefs, you cannot retrieve the name of what is being referenced. But unlike other references, you cannot make a new copy of what is being referenced. Any reference to that code will point to the same code, not a new version. Since you can't change the code that is being referenced, this distinction is mostly academic.

Our polygon area program exhibits some regularity. Let's see what happens when we take advantage of that regularity by putting the coderefs into some data structures. Modify **area.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

~~my $triangle_ref = \&triangle;~~
~~my $square_ref   = \&square;~~
~~my $pentagon_ref = \&pentagon;~~

my $side = shift or die;  # Length
my @area_refs = ( 0 .. 2, \&triangle, \&square, \&pentagon );

for my $sides ( 3..5 )
{
  print "Area of regular $sides-gon of side $side = ",
        $area_refs[$sides]->( $side ), "\n";
}

~~print "Area of triangle of side $side = ", $triangle_ref->( $side ), "\n";~~
~~print "Area of square of side   $side = ", $square_ref->( $side ), "\n";~~
~~print "Area of pentagon of side $side = ", $pentagon_ref->( $side ), "\n";~~

sub triangle  # Equilateral
{
  my $side = shift;

  return sqrt(3) / 4 * $side ** 2
}

sub square
{
  my $side = shift;

  return $side ** 2;
}

sub pentagon   # Regular
{
  my $side = shift;

  return sqrt(25 + 10 * sqrt(5)) / 4 * $side ** 2;
}
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./area.pl 7
Area of regular 3-gon of side 7 = 21.2176223927187
Area of regular 4-gon of side 7 = 49
Area of regular 5-gon of side 7 = 84.3033926288594
```

Sorry, I couldn't resist playing a little trick on you here. The number of sides that comprise the regular shapes for which we have area subroutines, are small integers; I represented them using indexes in an array. But it's more common, and in general more useful, to use a hash as the data structure, like this:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

~~my @area_refs = ( 0 .. 2, \&triangle, \&square, \&pentagon );~~
my %area_ref = ( 3 => \&triangle, 4 => \&square, 5 => \&pentagon );

my $side = shift or die;  # Length
for my $sides ( 3..5 )
{
  print "Area of regular $sides-gon of side $side = ",
        ~~$area_refs[$sides]->( $side ), "\n";~~
        $area_ref{$sides}->( $side ), "\n";
}

sub triangle  # Equilateral
{
  my $side = shift;

  return sqrt(3) / 4 * $side ** 2
}

sub square
{
  my $side = shift;

  return $side ** 2;
}

sub pentagon   # Regular
{
  my $side = shift;

  return sqrt(25 + 10 * sqrt(5)) / 4 * $side ** 2;
}
```

Check Syntax and run it. You'll see exactly the same results.

Dispatch Tables

Because we're now treating code like data, we can employ a highly useful programming technique called a *dispatch table*. Let's see how to use the dispatch table in an example. There were once text-based adventure games that involved dialogs like this one:

OBSERVE:

```
Command: look
You are in a small cave with torches flickering on the blood-stained walls.
There are exits to the north, south, and east.
A hungry troll bars your way.
There is a sword here.
Command: give troll jewel
The troll scorns your feeble attempt at appeasement.
Command: take sword
Taken.
Command: kill troll with sword
(gory scenes omitted)
```

We can implement a parser for the commands this game allows by using a separate subroutine for each major verb, indexed through a hash that will become the dispatch table. It's not necessarily the best way, but we'll use it for now. Create **dispatch.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %verb = ( give => \&give, drop => \&drop,
             take => \&take, kill => \&kill,
             look => \&look, have => \&inventory, quit => \&quit );

# Minimum definition of the game to make demonstration work

my %inventory     = ( me => { jewel => 1 }, troll => { diamond => 1 } );
my %room_contents = ( cave => { sword => 1 } );
my %location      = ( me => 'cave', troll => 'cave', thief => 'attic' );

for ( prompt(); $_ = <STDIN>; prompt() )
{
  chomp;
  next unless /(\S+)(?:\s+(.+))?/;
  $verb{$1} or warn "\tI don't know how to $1\n" and next;
  $verb{$1}->($2);
}

sub prompt { print "Command: " }

sub quit { exit }

sub give
{
  local $_ = shift;
  /(\S+)\s+to\s+(\S+)/ or return warn "\tGive what to who?\n";
  delete $inventory{me}{$1} or return warn "\tYou don't have a $1\n";
  $inventory{$2}{$1}++;
  print "\tGiven\n";
}

sub drop
{
  my $what = shift;
  delete $inventory{me}{$what} or return warn "\tYou don't have a $what\n";
  my $here = $location{me};
  $room_contents{$here}{$what}++;
  print "\tDropped\n";
}

sub take
{
  my $what = shift;
  my $here = $location{me};
  delete $room_contents{$here}{$what} or return warn "\tThere's no $what here\n"
;
  $inventory{me}{$what}++;
  print "\tTaken\n";
}

sub inventory
{
  for my $have ( keys %{ $inventory{me} } )
  {
    print "\tYou have a $have\n";
  }
}

sub look
{
  my $here = $location{me};
  print "\tYou are in the $here\n";
  for my $around ( keys %{ $room_contents{$here} } )
  {
    print "\tThere is a $around on the ground\n";
  }
  for my $actor ( keys %location )
  {
    next if $actor eq 'me';
    print "\tThere is a $actor here\n" if $location{$actor} eq $here;
  }
}

sub kill
{
  local $_ = shift;
  /(\S+)\s+with\s+(\S+)/ or return warn "\tKill who with what?\n";
  $inventory{me}{$2} or return warn "\tYou don't have a $2\n";
  my $here = $location{me};
  my $its_at = $location{$1} or return warn "\tNo $1 to kill\n";
  $its_at eq $here or return warn "\tThe $1 isn't here\n";
  delete $location{$1};
  my $had_ref = delete $inventory{$1};
  $room_contents{$here}{$_}++ for keys %$had_ref;
  print "Dispatched!\n";
}
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./dispatch.pl
Command: look
        You are in the cave
        There is a sword on the ground
        There is a troll here
Command: have
        You have a jewel
Command: take knife
        There's no knife here
Command: take sword
        Taken
Command: look
        You are in the cave
        There is a troll here
Command: have
        You have a sword
        You have a jewel
Command: kill troll
        Kill who with what?
Command: kill thief with knife
        You don't have a knife
Command: kill thief with sword
        The thief isn't here
Command: kill frog with sword
        No frog to kill
Command: kill troll with sword
Dispatched!
Command: look
        You are in the cave
        There is a diamond on the ground
Command: quit
```

Is that fun or what? There is so much functionality packed into that little stretch of code! We're using data structures and references to their maximum advantage. Let's look at what's happening in the **kill** subroutine:

OBSERVE:

```
sub kill
{
  local $_ = shift;
  /(\S+)\s+with\s+(\S+)/ or return warn "\tKill who with what?\n";
  $inventory{me}{$2} or return warn "\tYou don't have a $2\n";
  my $here = $location{me};
  my $its_at = $location{$1} or return warn "\tNo $1 to kill\n";
  $its_at eq $here or return warn "\tThe $1 isn't here\n";
  delete $location{$1};
  my $had_ref = delete $inventory{$1};
  $room_contents{$here}{$_}++ for keys %$had_ref;
  print "Dispatched!\n";
}
```

1. Use the **regex** to make sure that the operand matches the phrase "<**someone**> with <**something**>". If so, the "**someone**" will be in **$1** and the "**something**" will be in **$2**.
2. Use **$inventory{me}** to make sure that I have the "**something**."
3. Get my current **$location** into the variable **$here** to simplify later expressions.
4. Get the **$location** of "**someone ($1)**" into **$its_at**; if "**someone ($1)**" doesn't exist, they will have no entry in **%location**, the assignment will be false, and we will return with a warning.
5. Verify that the **location** of "**someone**" is **here** (you can't kill someone in a different room).
6. Begin the execution process: remove "**someone**" from the location listing.
7. Retrieve a list of items that "**someone**" owned into **$had_ref**, and remove it from the **inventory** in the process.
8. Add the contents of everything **$had_ref** points at to the contents of the current **location**.

This program illustrates several advanced Perl techniques> when you get a handle on them, you'll be able to construct complex data processing programs.

References to Anonymous Subroutines

It seems a little excessive to have to create an entire subroutine **quit** just to call **exit**. Can't we just remove the function call and place its contents directly where the call would be made? Yes we can, using *anonymous* subroutines. We can create references to blocks of code that don't have a subroutine name. Modify **dispatch.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

 my %verb = ( give => \&give, drop => \&drop,
              take => \&take, kill => \&kill,
              look => \&look, have => \&inventory, quit => ~~\&quit~~ sub { exit } )
;

# Minimum definition of the game to make demonstration work

my %inventory     = ( me => { jewel => 1 }, troll => { diamond => 1 } );
my %room_contents = ( cave => { sword => 1 } );
my %location      = ( me => 'cave', troll => 'cave', thief => 'attic' );

for ( prompt(); $_ = <STDIN>; prompt() )
{
  chomp;
  next unless /(\S+)(?:\s+(.+))?/;
  $verb{$1} or warn "\tI don't know how to $1\n" and next;
  $verb{$1}->($2);
}

sub prompt { print "Command: " }

~~sub quit { exit }~~

sub give
{
  local $_ = shift;
  /(\S+)\s+to\s+(\S+)/ or return warn "\tGive what to who?\n";
  delete $inventory{me}{$1} or return warn "\tYou don't have a $1\n";
  $inventory{$2}{$1}++;
  print "\tGiven\n";
}

sub drop
{
  my $what = shift;
  delete $inventory{me}{$what} or return warn "\tYou don't have a $what\n";
  my $here = $location{me};
  $room_contents{$here}{$what}++;
  print "\tDropped\n";
}

sub take
{
  my $what = shift;
  my $here = $location{me};
  delete $room_contents{$here}{$what} or return warn "\tThere's no $what here\n"
;
  $inventory{me}{$what}++;
  print "\tTaken\n";
}

sub inventory
{
  for my $have ( keys %{ $inventory{me} } )
  {
    print "\tYou have a $have\n";
  }
}

sub look
{
  my $here = $location{me};
  print "\tYou are in the $here\n";
  for my $around ( keys %{ $room_contents{$here} } )
  {
    print "\tThere is a $around on the ground\n";
  }
  for my $actor ( keys %location )
  {
    next if $actor eq 'me';
    print "\tThere is a $actor here\n" if $location{$actor} eq $here;
  }
}

sub kill
{
  local $_ = shift;
  /(\S+)\s+with\s+(\S+)/ or return warn "\tKill who with what?\n";
  $inventory{me}{$2} or return warn "\tYou don't have a $2\n";
  my $here = $location{me};
  my $its_at = $location{$1} or return warn "\tNo $1 to kill\n";
  $its_at eq $here or return warn "\tThe $1 isn't here\n";
  delete $location{$1};
  my $had_ref = delete $inventory{$1};
  $room_contents{$here}{$_}++ for keys %$had_ref;
  print "Dispatched!\n";
}
```

Check Syntax and run it. It will work just like it did before. You now have the syntax of an anonymous subroutine reference:

OBSERVE:

```
$code_ref = sub { ... }
```

sub { ... } can appear anywhere you want to have a code reference in your program. In particular, this enables you to create a subroutine that returns a reference to a subroutine! We'll explore that in detail a bit later.

Closures

"To hear some people talking about closures, you'd think they were discussing quantum physics, brain surgery, or VCR programming."
-Damian Conway, *Object-Oriented Perl*

A **closure** is a subroutine that uses a variable that is defined outside of the subroutine; the subroutine is said to be *closed over* the variable. You already know how to do that. For example, the **kill** routine in **dispatch.pl** uses the variable **%location**, which is defined outside of itself.

The most useful closure is one that uses a variable that isn't global, for example:

OBSERVE:

```
{
  my $fh;

  sub open_log
  {
    open $fh, '>', $LOG_FILE or die "Can't open $LOG_FILE: $!\n";
  }
  sub write_log
  {
    print {$fh} localtime().": @_\n";
  }
}
```

The two subroutines **open_log** and **write_log** are the only pieces of code that can use the variable **$fh**, because it is contained within the naked block.

An even *more* useful kind of closure will have its code inside an anonymous subroutine. One application of this is the technique known as **currying**, which allows you to create a subroutine dynamically that acts like another subroutine, with some of its arguments fixed to certain values.

That's easier done than said though. Create **curry.pl** in the usual **/perl3** folder:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

sub add
{
  my ($left, $right) = @_;

  return $left + $right;
}

sub make_adder
{
  my $operand = shift;

  return sub { add( $operand, @_ ) };
}


my $add_three = make_adder( 3 );
my $add_seven = make_adder( 7 );

my $i = 12;

print "\$i = $_\n" for $i, $add_three->($i), $add_seven->($i);
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./curry.pl
$i = 12
$i = 15
$i = 19
```

I admit, this isn't the most exciting example, but it does demonstrate the concept of currying well. Let's go over it in detail:

OBSERVE:

```
#!/usr/bin/perl
use strict;
use warnings;

sub add
{
  my ($left, $right) = @_;

  return $left + $right;
}

sub make_adder
{
  my $operand = shift;

  return sub { add( $operand, @_ ) };
}


my $add_three = make_adder( 3 );
my $add_seven = make_adder( 7 );

my $i = 12;

print "\$i = $_\n" for $i, $add_three->($i), $add_seven->($i);
```

The anonymous subroutine is closed over the variable **$operand**. Each coderef that is returned from **make_adder** is to the *same* piece of code, but each coderef has a *different* idea of what **$operand** contains: it's whatever it contained at the time the anonymous subroutine was constructed. The @_ inside the anonymous code refers to the parameters that are passed to the coderef when it is called, *not* the parameters that are passed to the enclosing subroutine (in this case, **make_adder**).

This is the basis for much more complex, and more useful code. We'll try out a more practical example here, using a program from an earlier lesson.

Open **stock.pl**, click the **Save As** icon ( ) and save it as **stock2.pl** in your **/perl3** folder. Now make these changes:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

~~my $stock_ref = get_stock_ref( 'INSECTS' );~~
~~print "I have $$stock_ref{ants} ants\n";~~

my @DATA = <DATA>;

my $insect_ref = get_stock_ref( 'INSECTS' );
my $fish_ref = get_stock_ref( 'FISH' );
print "I have ", $insect_ref->( 'ants' ), " ants\n";
print "I have ", $fish_ref->( 'guppies' ), " guppies\n";
print "When looking under insects:\n";
$insect_ref->( 'guppies' );

sub get_stock_ref
{
  my $seeking = shift;

  ~~my (@data, $type);~~
~~  while ( <DATA> )~~
~~  {~~
~~    if ( ( my ($what, $number) = /(.*)\s+(\d+)/ ) && ( $type eq $seeking ) )~~
~~    {~~
~~      push @data, $what, $number;~~
~~    }~~
~~    elsif ( /(\S+)/ )~~
~~    {~~
~~      $type = $1;~~
~~    }~~
~~    else~~
~~    {~~
~~      next;~~
~~    }~~
~~  }~~
~~  return { @data };~~

  return sub {
    my $want = shift;
    my $type;
    for ( @DATA )
    {
      if ( ( my ($what, $number) = /(.*)\s+(\d+)/ ) && ( $type eq $seeking ) )
      {
        return $number if $what eq $want;
      }
      elsif ( /(\S+)/ )
      {
        $type = $1;
      }
      else
      {
        next;
      }
    }
    die "Couldn't find any $want\n";
  };
}

__END__

INSECTS
ants 1000000
beetles 200000
```

```
MAMMALS
aardvarks 4
antelopes 3

FISH
guppies 10
angel fish 40
king crabs 10
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./stock2.pl
I have 1000000 ants
I have 10 guppies
When looking under insects:
Couldn't find any guppies
```

Here we've created routines that specialize in searching *insects* and *fish*, respectively.

Callbacks

A *callback* is a subroutine reference passed to another subroutine to be executed as part of some other code. Essentially, it lets you turn a program *inside-out*. We'll give it a try. Suppose you have written some code that goes through all of the files in a directory. Create **files.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

process_directory( shift || '.' );

sub process_directory
{
  my $dir = shift;

  opendir my $dh, $dir or die "Couldn't open $dir: $!\n";
  while ( my $file = readdir $dh )
  {
    next if $file =~ /\A\.\.?\z/;
    print "Found an executable file: $file\n" if -x $file;
  }
}
```

Check Syntax and run it (the results may not match exactly):

INTERACTIVE SESSION:

```
cold:~/perl3$ ./files.pl
Found an executable file: files.pl
Found an executable file: graphic.pl
Found an executable file: stock.pl
Found an executable file: dispatch.pl
Found an executable file: area.pl
Found an executable file: curry.pl
```

But this isn't very generalizable code. Our routine for processing a directory is specialized to look for executable files. But we want to pass in *behavior* as a parameter. A callback lets us do that. Modify the program as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

process_directory( shift || '.' , \&is_executable  );

sub is_executable
{
  my $file = shift;

  print "Found an executable file: $file\n" if -x $file;
}

sub process_directory
{
  my $dir = shift;
  my $callback = shift;

  opendir my $dh, $dir or die "Couldn't open $dir: $!\n";
  while ( my $file = readdir $dh )
  {
    next if $file =~ /\A\.\.?\z/;
    $callback->( $file );
    ~~print "Found an executable file: $file\n";~~
  }
}
```

Check Syntax and run it. The results will be the same as before. But now we can make our directory processing code do all kinds of things without having to rewrite that **readdir()** loop every time. Modify your code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

process_directory( shift || '.', ~~\&is_executable~~ \&print_size );

sub print_size
{
  my $file = shift;

  print "$file is ", -s $file, " bytes long\n";
}

sub is_executable
{
  my $file = shift;

  print "Found an executable file: $file\n" if -x $file;
}

sub process_directory
{
  my $dir = shift;
  my $callback = shift;

  opendir my $dh, $dir or die "Couldn't open $dir: $!\n";
  while ( my $file = readdir $dh )
  {
    next if $file =~ /\A\.\.?\z/;
    $callback->( $file );
  }
}
```

Check Syntax and run it:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./files.pl
files.pl is 514 bytes long
graphic.pl is 205 bytes long
stock.pl is 858 bytes long
dispatch.pl is 2088 bytes long
area.pl is 523 bytes long
curry.pl is 335 bytes long
```

Can you see the power and flexibility that this technique unlocks?

A Few More Code Reference Notes

Before we close out this lesson you want to be aware of a few more coderef behaviors.

ref()

The result of calling **ref()** on a subroutine reference is **CODE**.

Syntax

Expressions dereferencing subroutines *cannot* be interpolated in double-quoted strings. But you *can* leave out the arrow between braces and parentheses, for example:

OBSERVE:

```
$area_ref{$sides}( $side );
```

This is not common practice though. Perl programmers are used to arrows when code is being called by reference, and some of that expectation is due to the syntax for *method calls*. Method calls are investigated further in a later course.

If you're curious and wondering how you would apply Rule 1 and Rule 2 to dereferencing subroutine references (and I know that you are!), the answer lies in the ampersand syntax. Check it out:

OBSERVE:

```
&{ expression_returning_coderef }( arguments )
```

But really, that's so ugly, it's just not worth the trouble.

Nice work so far. You've got a handle on references—no easy feat! Next up: exceptions and how to deal with them. See you in the next lesson!

Once you finish the lesson, go back to the syllabus to complete the homework.

Exception Handling

Lesson Objectives

When you complete this lesson, you will be able to:

- handle Exceptions.
- us the special hash %SIG to handle *signals*.

Welcome back to *Advanced Perl!* We've learned as much as we planned to about references in the course; now we're ready to explore some useful Perl features that *use* those references.

The term "transfer of control" refers to the way the run-time system determines which statement in a program to execute next.

So far, the transfer of control in our programs has been *local*, and the next statement to be executed has been predictable. Our programs have either the statements in sequence, or in a loop, or the statement after returning from a subroutine. But now we'll see some *nonlocal* transfer of control using *exceptions.*

Exceptions

An exception is when your program does something *exceptional*, that is, different from the usual or typical pattern of execution. This almost always means that an error has occurred and the program needs to handle it by going to a special place in the code.

The terminology of exceptions is that we *throw* an exception when the error is first recognized in a program, and *catch* an exception when the flow of control arrives to the *exception handler*. The notion that program execution is somehow "up in the air" between these events is an appropriate one, since the transfer of control may jump over many lines of code and *stack frames* (every time you call a subroutine, a "frame" containing data concerning where it was called from gets pushed onto a stack in the Perl run-time, and every time you return from a subroutine, the frame is popped off; so nested subroutine calls result in multiple frames on the stack).

The term *exception* is often used in a abstract sense. We will make it more concrete by saying that the exception is an error message, a string.

Throwing Exceptions

Perl exceptions are thrown with the function **die()**—the same **die()** that causes your program to exit with an error message. That is an exception's default behavior: quit the program. Nothing caught the exception, so a *default exception handler* just prints the exception to the standard error stream and then calls **exit()** with a number other than zero (the actual code that it exits with varies and is described in perldoc -f die).

Let's go to work using exceptions right now by creating **exception.pl** in our **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

init( shift || 'nonesuch.conf' );
print "Cell at (4,2) = ", field_value(4,2), "\n";

{
  my $data_ref;

  sub init
  {
    handle_file( config => shift );
    $data_ref = handle_file( data => config( 'datafile' ) );
  }

  sub field_value
  {
    my ($row, $column) = @_;
    $data_ref or die "No data collected yet";
    return $data_ref->[$row-1][$column-1];
  }
}

{
  my $conf_ref;
  sub handle_file
  {
    my ($file_type, $file_name) = @_;

    if ( $file_type eq 'config' )
    {
      $conf_ref = read_config( $file_name, '=' );
    }
    elsif ( $file_type eq 'data' )
    {
      return read_data( $file_name, $conf_ref->{separator} );
    }
  }

  sub config { $conf_ref->{ shift() } }
}


sub read_config
{
  my ($file, $delimiter) = @_;

  open my $fh, '<', $file or die "Can't open $file: $!";
  my %config;
  while ( <$fh> )
  {
    chomp;
    /(.+)$delimiter(.+)/ or next;
    $config{$1} = $2;
  }
  return \%config;
}

sub read_data
{
  my ($file, $separator) = @_;

  open my $fh, '<', $file or die "Can't open $file: $!";
  my @output;
  while ( <$fh> )
  {
    chomp;
    my @fields = split /\Q$separator\E/;
    @fields > 1 or die "Separator not found at line $.";
    push @output, \@fields;
  }
  return \@output;
}
```

Now create the input file **config.in** in your **/perl3** folder as shown:

CODE TO TYPE:

```
separator=|
datafile=data.dat
```

...and create the data file **data.dat** as shown:

CODE TO TYPE:

```
Date|High|Low|Close
10/1/10|100.5|95.6|96.7
10/2/10|96.7|94.4|94.6
10/3/10|94.5|92.2|92.2
10/4/10|93.4|91.2|91.8
```

Check Syntax and run **exception.pl** by typing the commands below as shown:

INTERACTIVE SESSION:

```
cold:~$ cd perl3
cold:~/perl3$ ./exception.pl
Can't open nonesuch.conf: No such file or directory at ./exception.pl line 49.
cold: ~$ ./exception.pl ./config.in
Cell at (4,2) = 94.5
```

Okay, now suppose the program was modified to wrap a loop around the main processing code. Edit **exception.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

~~init( shift || 'nonesuch.conf' );~~
~~print "Cell at (4,2) = ", field_value(4,2), "\n";~~
while ( my $config = shift )
{
  init( $config );
  print "Cell at (4,2) = ", field_value(4,2), "\n";
}

{
  my $data_ref;

  sub init
  {
    handle_file( config => shift );
    $data_ref = handle_file( data => config( 'datafile' ) );
  }

  sub field_value
  {
    my ($row, $column) = @_;
    $data_ref or die "No data collected yet";
    return $data_ref->[$row-1][$column-1];
  }
}

{
  my $conf_ref;
  sub handle_file
  {
    my ($file_type, $file_name) = @_;

    if ( $file_type eq 'config' )
    {
      $conf_ref = read_config( $file_name, '=' );
    }
    elsif ( $file_type eq 'data' )
    {
      return read_data( $file_name, $conf_ref->{separator} );
    }
  }

  sub config { $conf_ref->{ shift() } }
}


sub read_config
{
  my ($file, $delimiter) = @_;

  open my $fh, '<', $file or die "Can't open $file: $!";
  my %config;
  while ( <$fh> )
  {
    chomp;
    /(.+)$delimiter(.+)/ or next;
    $config{$1} = $2;
  }
  return \%config;
}

sub read_data
{
  my ($file, $separator) = @_;

  open my $fh, '<', $file or die "Can't open $file: $!";
  my @output;
  while ( <$fh> )
  {
    chomp;
    my @fields = split /\Q$separator\E/;
    @fields > 1 or die "Separator not found at line $.";
    push @output, \@fields;
  }
  return \@output;
}
```

Now we can run it with a list of config file names. But let's see what happens if one of those names is bad. Run the program like this:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./exception.pl ./nonesuch ./config.in
Can't open nonesuch: No such file or directory at ./exception.pl line 52.
```

Let's say we want our program to keep going to the next config file. We might end up modifying code that's far away from the part of the code that dictates that change; and the code we modify might be part of a library that other programs need to use. (We haven't seen how to do that, but it is in the next course!) Now we're changing code that may not even belong to us and may mess with the plans of other users of that code.

Catching Exceptions

The way to avoid such pitfalls is to catch the exceptions. Modify **exception.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

while ( my $config = shift )
{
  eval {
    init( $config );
    print "Cell at (4,2) = ", field_value(4,2), "\n";
  };
  if ( $@ )
  {
    chomp $@;
    warn "Caught exception '$@', continuing...\n";
  }
  ~~init( $config );~~
  ~~print "Cell at (4,2) = ", field_value(4,2), "\n";~~
}

{
  my $data_ref;

  sub init
  {
    handle_file( config => shift );
    $data_ref = handle_file( data => config( 'datafile' ) );
  }

  sub field_value
  {
    my ($row, $column) = @_;
    $data_ref or die "No data collected yet";
    return $data_ref->[$row-1][$column-1];
  }
}

{
  my $conf_ref;
  sub handle_file
  {
    my ($file_type, $file_name) = @_;

    if ( $file_type eq 'config' )
    {
      $conf_ref = read_config( $file_name, '=' );
    }
    elsif ( $file_type eq 'data' )
    {
      return read_data( $file_name, $conf_ref->{separator} );
    }
  }

  sub config { $conf_ref->{ shift() } }
}


sub read_config
{
  my ($file, $delimiter) = @_;

  open my $fh, '<', $file or die "Can't open $file: $!";
  my %config;
  while ( <$fh> )
  {
    chomp;
    /(.+)$delimiter(.+)/ or next;
    $config{$1} = $2;
  }
  return \%config;
}

sub read_data
{
  my ($file, $separator) = @_;

  open my $fh, '<', $file or die "Can't open $file: $!";
  my @output;
  while ( <$fh> )
  {
    chomp;
    my @fields = split /\Q$separator\E/;
    @fields > 1 or die "Separator not found at line $.";
    push @output, \@fields;
  }
  return \@output;
}
```

Check Syntax and run the program again as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./exception.pl ./nonesuch ./config.in
Caught exception 'Can't open nonesuch: No such file or directory at ./exception.
pl line 59.', continuing...
Cell at (4,2) = 94.5
```

So, how does it work?

OBSERVE:

```
while ( my $config = shift )
{
  eval {
    init( $config );
    print "Cell at (4,2) = ", field_value(4,2), "\n";
  };
  if ( $@ )
  {
    chomp $@;
    warn "Caught exception '$@, continuing...\n";
  }
  init( $config );
  print "Cell at (4,2) = ", field_value(4,2), "\n";
}
```

First, we have an **eval()** statement, which takes a *block* as argument. **eval()** causes any **die** situation to "die" only in the scope of the **eval()** block; it's as though the **eval()** block is now the whole program, so **die** will only kill the program within the block.

Second, the special variable **$@** is set to the argument that was given to **die** (you can see this in the program output above).

You can even catch implicit **die**s generated by Perl itself; for example, when you divide a number by zero. Check that out with this one-liner:

INTERACTIVE SESSION:

```
cold:~/perl3$ perl -e 'eval { 1/0 } or print "Caught exception $@"'
Caught exception Illegal division by zero at -e line 1.
```

You can also have nested **eval()**s. Make these changes to the **handle_file** routine in **exception.pl**:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

while ( my $config = shift )
{
  eval {
    init( $config );
    print "Cell at (4,2) = ", field_value(4,2), "\n";
  };
  if ( $@ )
  {
    chomp $@;
    warn "Caught exception '$@', continuing...\n";
  }
}

{
  my $data_ref;

  sub init
  {
    handle_file( config => shift );
    $data_ref = handle_file( data => config( 'datafile' ) );
  }

  sub field_value
  {
    my ($row, $column) = @_;
    $data_ref or die "No data collected yet";
    return $data_ref->[$row-1][$column-1];
  }
}

{
  my $conf_ref;
  sub handle_file
  {
    my ($file_type, $file_name) = @_;

    if ( $file_type eq 'config' )
    {
      $conf_ref = read_config( $file_name, '=' );
    }
    elsif ( $file_type eq 'data' )
    {
      ~~return read_data( $file_name, $conf_ref->{separator} );~~
      return eval { read_data( $file_name, $conf_ref->{separator} ) }
                  || [ 0,0,0,[0,42]];
    }
  }

  sub config { $conf_ref->{ shift() } }
}


sub read_config
{
  my ($file, $delimiter) = @_;

  open my $fh, '<', $file or die "Can't open $file: $!";
  my %config;
  while ( <$fh> )
  {
    chomp;
    /(.+)$delimiter(.+)/ or next;
    $config{$1} = $2;
  }
  return \%config;
}

sub read_data
{
  my ($file, $separator) = @_;

  open my $fh, '<', $file or die "Can't open $file: $!";
  my @output;
  while ( <$fh> )
  {
    chomp;
    my @fields = split /\Q$separator\E/;
    @fields > 1 or die "Separator not found at line $.";
    push @output, \@fields;
  }
  return \@output;
}
```

Now create the file **config.bad** to refer to a nonexistent data file:

CODE TO TYPE:

```
separator=|
datafile=nodata.dat
```

Check Syntax and run the program as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./exception.pl ./nonesuch ./config.bad ./config.in
Caught exception 'Can't open nonesuch: No such file or directory at ./exception.
pl line 60.', continuing
Cell at (4,2) = 42
Cell at (4,2) = 94.5
```

The exception from the **read_data** subroutine is being caught by the **eval()** wrapped around the call to **read_data** inside the **handle_file** routine. Here, we aren't even looking at **$@**; we know that if an exception is thrown inside of an **eval()** block, the result of the block will be **undef**. We know that if there is no exception, the result from **read_data** will be a reference, which is true, so this code effectively provides a fallback reference if an exception is raised (the fallback is just big enough to contain the element that the higher level code is printing; this code is a bit contrived in that respect, but hey, we do whatever it takes to illustrate our points).

$@

As you've seen, **$@** contains the string that was passed to **die** (and since we didn't terminate those strings with a newline, Perl added "at ./exception.pl line 60."). We can test the value of **$@** to determine whether to take different action depending on the type of exception. Modify **exception.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

while ( my $config = shift )
{
  eval {
    init( $config );
    print "Cell at (4,2) = ", field_value(4,2), "\n";
  };
  if ( $@ )
  {
    chomp $@;
    warn "Caught exception '$@', continuing...\n";
  }
}

{
  my $data_ref;

  sub init
  {
    handle_file( config => shift );
    $data_ref = handle_file( data => config( 'datafile' ) );
  }

  sub field_value
  {
    my ($row, $column) = @_;
    $data_ref or die "No data collected yet";
    return $data_ref->[$row-1][$column-1];
  }
}

{
  my $conf_ref;
  sub handle_file
  {
    my ($file_type, $file_name) = @_;

    if ( $file_type eq 'config' )
    {
      $conf_ref = read_config( $file_name, '=' );
    }
    elsif ( $file_type eq 'data' )
    {
      ~~return eval { read_data( $file_name, $conf_ref->{separator} ) }~~
~~                         || [ 0,0,0,[0,42]];~~
      my $ref = eval { read_data( $file_name, $conf_ref->{separator} ) };
      if ( $@ )
      {
        $@ =~ /separator/i and return [ 0,0,0,[0,42]];
        die;
      }
      return $ref;
    }
  }

  sub config { $conf_ref->{ shift() } }
}

sub read_config
{
  my ($file, $delimiter) = @_;

  open my $fh, '<', $file or die "Can't open $file: $!";
  my %config;
  while ( <$fh> )
  {
    chomp;
    /(.+)$delimiter(.+)/ or next;
    $config{$1} = $2;
  }
  return \%config;
}

sub read_data
{
  my ($file, $separator) = @_;

  open my $fh, '<', $file or die "Can't open $file: $!";
  my @output;
  while ( <$fh> )
  {
    chomp;
    my @fields = split /\Q$separator\E/;
    @fields > 1 or die "Separator not found at line $.";
    push @output, \@fields;
  }
  return \@output;
}
```

Now create a file **data.badsep** that has no separator characters in at least one of the lines:

CODE TO TYPE:

```
Date,High,Low,Close
10/1/10|100.5|95.6|96.7
10/2/10|96.7|94.4|94.6
10/3/10|94.5|92.2|92.2
10/4/10|93.4|91.2|91.8
```

...and finally, create **config.badsep** to refer to that data file:

CODE TO TYPE:

```
separator=|
datafile=data.badsep
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./exception.pl ./config.badsep ./config.bad
Cell at (4,2) = 42
Caught exception 'Can't open nodata.dat: No such file or directory at ./exceptio
n.pl line 79.
        ...propagated at ./exception.pl line 51.', continuing...
```

We've changed the behavior so that the "fallback" reference is only returned in the case of a missing separator. If we just call **die** without any arguments, because **$@** contains a value, Perl appends the string "...propagated..." to it and uses that as the exception text.

%SIG

Process Signals

The special hash **%SIG** is used to handle *signals*: an event delivered to your program from another process that interrupts it. A signal is a small number, typically represented by a string. For example, when you hit **[Ctrl+C]** to interrupt a program, you're sending it the INT (interrupt) signal. You can see that with this example:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$SIG{INT} = sub { die "Caught INT signal" };

sleep 10;

print "Normal termination\n";
```

Check Syntax and run it. The first time you run it, wait ten seconds; the second time, press **[Ctrl+C]** (^C won't appear onscreen) before the ten seconds are up:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./signal.pl
Normal termination
cold:~/perl3$ ./signal.pl
^CCaught INT signal at ./signal.pl line 5.
```

A signal handler is a code reference stored in a special hash in the key with the name of the signal being handled. There are a few dozen types of signal, but only a select few are particularly useful:

Signal Name	**Meaning**
ALRM	Triggered by a call to **alarm** going off (see **perldoc -f alarm**
QUIT	Stronger version of **[Ctrl+C]** triggered by **[Ctrl+\\]**
TERM	Termination signal sent by another process
INT	Triggered by **[Ctrl+C]**
HUP	A general-purpose signal that another process can send to get attention with **kill -HUP**

You can have the same handler for multiple signals, because the name of the signal is passed as the first argument to a signal handler. Modify **signal.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$SIG{INT} = $SIG{QUIT} = sub { die "Caught ~~INT~~$_[0] signal" };

sleep 10;

print "Normal termination\n";
```

Check Syntax and run it twice, first interrupting it with **[Ctrl+C]**, and then with **[Ctrl+\]**:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./signal.pl
^CCaught INT signal at ./signal.pl line 5.
cold:~/perl3$ ./signal.pl
^\Caught QUIT signal at ./signal.pl line 5.
```

If you assign the string **'IGNORE'** (not a coderef) to a **%SIG** element, the signal is ignored.

Beware of creating a signal handler that ignores a signal or fails to die, and potentially assigning it to all the ways of interrupting your program. If that kind of program runs out of control, you have to send the untrappable signal KILL to it from another process. This is to be avoided!

Pseudo-Signals

Signals like INT and TERM are real properties of Unix-like and other operating systems. But there are also two special signals that you can define handlers for: **__WARN__** and **__DIE__**.

$SIG{__WARN__}, if defined, will be called whenever your program calls **warn()** (though you can call **warn()** from inside a __WARN__ handler without fear of it recursing). The argument is not the signal name, but instead, it's the message from **warn()**. Create **warn.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

$SIG{__WARN__} = \&warn_handler;

sub warn_handler
{
  my $msg = shift;

  $msg =~ /uninitialized value (\S+)/ and return warn "Undef: $1\n";
  die "Exception: $msg";
}

my $x;
print $x;
my %x = (1);
print "Unreached\n";
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./warn.pl
Undef: $x
Exception: Odd number of elements in hash assignment at ./warn.pl line 17.
```

We've altered the text of one type of warning and "promoted" all other warnings to fatal exceptions.

The pseudo-signal handler **$SIG{__DIE__}** is controversial. It's usually best to avoid it. Although there is one relatively safe usage: forcing a stack trace. The **cluck()** routine exported by the **Carp** module will do that. Copy the program **exception.pl** into **sigdie.pl** and modify it as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

use Carp qw(cluck);

$SIG{__DIE__} = sub { cluck shift };

while ( my $config = shift )
{
  eval {
    init( $config );
    print "Cell at (4,2) = ", field_value(4,2), "\n";
  };
  if ( $@ )
  {
    chomp $@;
    warn "Caught exception '$@', continuing...\n";
  }
}

{
  my $data_ref;

  sub init
  {
    handle_file( config => shift );
    $data_ref = handle_file( data => config( 'datafile' ) );
  }

  sub field_value
  {
    my ($row, $column) = @_;
    $data_ref or die "No data collected yet";
    return $data_ref->[$row-1][$column-1];
  }
}

{
  my $conf_ref;
  sub handle_file
  {
    my ($file_type, $file_name) = @_;

    if ( $file_type eq 'config' )
    {
      $conf_ref = read_config( $file_name, '=' );
    }
    elsif ( $file_type eq 'data' )
    {
      my $ref = eval { read_data( $file_name, $conf_ref->{separator} ) };
      if ( $@ )
      {
        $@ =~ /separator/i and return [ 0,0,0,[0,42]];
        die;
      }
      return $ref;
    }
  }

  sub config { $conf_ref->{ shift() } }
}

sub read_config
{
  my ($file, $delimiter) = @_;
  open my $fh, '<', $file or die "Can't open $file: $!";
  my %config;
  while ( <$fh> )
  {
    chomp;
    /(.+)$delimiter(.+)/ or next;
    $config{$1} = $2;
  }
  return \%config;
}

sub read_data
{
  my ($file, $separator) = @_;

  open my $fh, '<', $file or die "Can't open $file: $!";
  my @output;
  while ( <$fh> )
  {
    chomp;
    my @fields = split /\Q$separator\E/;
    @fields > 1 or die "Separator not found at line $.";
    push @output, \@fields;
  }
  return \@output;
}
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./sigdie.pl ./config.badsep
Separator not found at line 1 at ./sigdie.pl line 90, <$fh> line 1.
 at ./sigdie.pl line 7
        main::__ANON__('Separator not found at line 1 at ./sigdie.pl line 90, <$
fh> l...') called at ./sigdie.pl line 90
        main::read_data('data.badsep', '|') called at ./sigdie.pl line 51
        eval {...} called at ./sigdie.pl line 51
        main::handle_file('data', 'data.badsep') called at ./sigdie.pl line 28
        main::init('config.badsep') called at ./sigdie.pl line 12
        eval {...} called at ./sigdie.pl line 11
Cell at (4,2) = 42
```

This is handy when you want to see how a routine was called. The __ANON__ portion of the program refers to the anonymous subroutine used as the __DIE__ handler.

So now, not only are you empowered to work with Perl, you even know how to handle exceptions when things go weird in Perl! That's excellent—keep going! See you soon...

Once you finish the lesson, go back to the syllabus to complete the homework.

Processing Command-Line Options

Lesson Objectives

When you complete this lesson, you will be able to:

- Process Options in Perl.
- use the Getopt::Std module to handle the parsing of single-letter options.
- use the Getopt::Long module to handle the parsing of multi-letter options.

Welcome back to your *Advanced Perl* course. You've come so far! In this lesson we'll learn about a capability for processing *command-line options.* When the Perl interpreter sees a statement, it breaks the statement down into smaller units of information. Each of these smaller units of information is called a token. As we learned in earlier lessons, when you run a program, everything you type after the program name on the command line ends up in the array **@ARGV**. The shell that interprets your keystrokes decides how to turn your command line into strings that Perl will pick up in **@ARGV**. Usually it just does the equivalent of **@ARGV = split /\s+/**, but certain shell *metacharacters* change the interpretation and may add extra tokens through wildcard expansion or result in fewer tokens, through whitespace quoting.

There are *conventions* for what we type after after we type the name of a program: specific ways to *format* text that have agreed-upon meanings. These conventions aren't noticed or implemented by the shell or by Perl.

Options

The command line is comprised of *options* and *arguments*. An option begins with one or two minus signs and may or may not be followed by a value. An argument follows any and all options, and does not begin with a minus sign.

In programs available on Unix, some options may be mandatory and others may be optional. User interface design and consistency isn't one of Unix's strong points. The correct way to find out how to run a program is to look at its documentation. Accurate and complete documentation is vital in this respect because it takes much longer to figure out how a program can be run by reading the source. So, when you write a program for other people, be sure to provide ample documentation that explains to how run it!

Even within established conventions, there is plenty of variation. Some programs support options that begin with a plus sign (the **find** command, for example). And options that begin with a single minus sign are followed by a single letter, while options that begin with a double minus sign are followed by a word. Single-letter options that take values may be called with or without a space between the letter and the value; multi-letter options must have a space, or be separated from the value by an equals sign. Some programs mix single-letter and multi-letter capabilities.

Option Processing in Perl

Of course, we could figure out all the options on a command line in Perl ourselves, by looking at **@ARGV**, parsing the contents, and unpacking them into a helpful data structure of some kind (or maybe just a list of variables), issuing warnings and exceptions where the convention wasn't obeyed. But that job gets tedious pretty quickly. Fortunately, there are a couple of *modules* that come with Perl that we can use to provide functions that make our job easier.

Here are a couple of sample calling sequences that we will be able to handle in this class:

OBSERVE:

```
dine -t lunch -dv -b modest me grace scott trish
dine --type lunch --debug --verbose --budget modest me grace scott trish
```

Another word for "calling sequence" that you'll see used in documentation is "synopsis."

Okay, let's see how to implement the parsing of those styles!

Getopt::Std

The **Getopt::Std** module handles the parsing of single-letter options. Let's start out with an example that doesn't use **Getopt::Std** and then change it to use it. Create **grep2.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my ($negated, $case_insensitive);
if ( $ARGV[0] eq '-v' )
{
  shift;
  $negated = 1;
}
if ( $ARGV[0] eq '-i' )
{
  shift;
  $case_insensitive = 1;
}

my $regex = shift;

while ( <> )
{
  if ( $case_insensitive )
  {
    print if $negated xor /$regex/i;
  }
  else
  {
    print if $negated xor /$regex/;
  }
}
```

This is an implementation of the homework from one of the Intermediate Perl lessons; it replicates the **-i** (ignore case) and **-v** (invert test) options of the standard Unix **grep** program. Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~$ cd perl3
cold:~/perl3$ ./grep2.pl REGEX ./grep2.pl
cold:~/perl3$ ./grep2.pl -i REGEX ./grep2.pl
my $regex = shift;
    print if $negated xor /$regex/i;
    print if $negated xor /$regex/;
```

Now modify the program so it looks like this:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

use Getopt::Std;

my %Opt;
getopts( 'iv', \%Opt );

my ($negated, $case_insensitive) = @Opt{ qw(v i) };
~~if ( $ARGV[0] eq '-v' )~~
~~{~~
~~  shift;~~
~~  $negated = 1;~~
~~}~~
~~if ( $ARGV[0] eq '-i' )~~
~~{~~
~~  shift;~~
~~  $case_insensitive = 1;~~
~~}~~

my $regex = shift;

while ( <> )
{
  if ( $case_insensitive )
  {
    print if $negated xor /$regex/i;
  }
  else
  {
    print if $negated xor /$regex/;
  }
}
```

Check Syntax and run it. It works just like it did before.

Pay special attention to the use of the **xor** operator—it works like this:

A xor B	A = 0	A = 1
B = 0	0	1
B = 1	1	0

That's what allows us to use the value of **$negated** to invert the sense of the match tests. You can experiment with the **-v** option as well if you like. Type the commands below to create a **test_input** file in your **/perl3** folder and view it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ cat > test_input
This string has an oddity.  Can you suss out what it is?
This sentence is more conventional.
(Press [Ctrl+C] here.)
cold:~/perl3$ ./grep2.pl -v e ./test_input
This string has an oddity.  Can you suss out what it is?
```

That's quite a code bargain for only two options! Here's how it works: **Getopt::Std** provides the function **getopts()**, which takes two arguments:

OBSERVE:

```
getopts( $option_string, \%options )
```

$option_string contains all the letters that you want to be able to parse, and **%options** is where **getopts** will put the results; note that we pass a *reference* to a hash there.

getopts() runs through **@ARGV** from the beginning, looking for anything beginning with a minus sign, and removing it from **@ARGV** as it sets **%options**. So, anything left in **@ARGV** after **getopts** finishes, will be what we consider *arguments*; in this case, the regex and the input filenames (which will be parsed by the <> operator).

Let's see what happens if we run this command:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./grep2.pl -f -w -v E ./test_input
Unknown option: f
Unknown option: w
This string has an oddity. Can you suss out what it is?
This sentence is more conventional.
```

That's neat. We get warnings for unknown options, but the valid processing still occurs!

Both of the options **-v** and **-i** expect no value, which means that if they are set on the command line, the value in **%Opt** will be **1** by default. To require an option to take a value, follow its letter in **$option_string** with a colon. Just to illustrate this, let's make the regex an option instead of an argument. Modify **grep2.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

use Getopt::Std;

my $USAGE = "Usage: $0 -r regex [-v] [-i] [file...]\n";
my %Opt;
~~getopts( 'iv', \%Opt );~~
getopts( 'ir:v', \%Opt ) or die $USAGE;

my ($negated, $case_insensitive) = @Opt{ qw(v i) };

my $regex = $Opt{r} or die $USAGE;
~~my $regex = shift;~~

while ( <> )
{
  if ( $case_insensitive )
  {
    print if $negated xor /$regex/i;
  }
  else
  {
    print if $negated xor /$regex/;
  }
}
```

Check Syntax and run it like this:

CODE TO TYPE:

```
cold:~/perl3$  ./grep2.pl -f -w -r E ./test_input
Unknown option: f
Unknown option: w
Usage: ./grep2.pl -r regex [-v] [-i] [file...]
cold:~/perl3$ ./grep2.pl -v regex ./grep2.pl
Usage: ./grep2.pl -r regex [-v] [-i] [file...]
```

Here, we added the **r** option to the options string, and by following it with a colon, required it to take a value; if no value is provided, there will be an error (but if anything were to follow **-r** on the command line, that would be taken as the value, so the only way to trigger that error is if *nothing* follows **-r** on the command line).

Next, we used the return value from **getopts()** to determine whether to quit altogether. **getopts()** will return false if it found an error in the options. Finally, we inserted logic requiring the **-r** option to be present. (I know that doesn't seem very "optional," but sometimes programs work like that.) **Getopt::Std** doesn't have a way to require an option, so we did it ourselves.

Option Clustering

Now, run the program like this:

CODE TO TYPE:

```
cold:~/perl3$ ./grep2.pl -v -i -r E ./test_input
This string has an oddity.  Can you suss out what it is?
```

You can merge single-letter options together now:

CODE TO TYPE:

```
cold:~/perl3$ ./grep2.pl -vi -r E ./test_input
This string has an oddity.  Can you suss out what it is?
cold:~/perl3$  ./grep2.pl -iv -r E ./test_input
This string has an oddity.  Can you suss out what it is?
cold:~/perl3$  ./grep2.pl -ivr E ./test_input
This string has an oddity.  Can you suss out what it is?
```

That's conventional for single-letter options, and here **getopts()** takes care of it for you. If you have an option that takes a value (like **-r**), it must be the *last* option in the cluster.

And, if you want to, you can leave out the space between an option and its value if it takes one:

CODE TO TYPE:

```
cold:~/perl3$ ./grep2.pl -vi -rE ./test_input
This string has an oddity.  Can you suss out what it is?
```

And although this works, it's going a bit far for readability:

CODE TO TYPE:

```
cold:~/perl3$ ./grep2.pl -virE ./test_input
This string has an oddity.  Can you suss out what it is?
```

Getopt::Long

Some programs take so many options that developers ran out of letters to use to represent them (see the manual page for **tar** for an example). In those cases developers created options that were more than one letter long. By convention, such options begin with two minus signs so that you don't think they're single-letter options that have been clustered. In this course, we'll restrict ourselves to creating programs that use either single- *or* multi-letter options, not both.

The **Getopt::Long** module handles the parsing of multi-letter options. Let's try using **Getopt::Long** with the

grep2.pl example we used before so we can compare it with the one we just made. Save **grep2.pl** as **grep_long.pl** and then edit it as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;
~~use Getopt::Std;~~
use Getopt::Long;

~~my $USAGE = "Usage: $0 -r regex [-v] [-i] [file...]\n";~~
my $USAGE = "Usage: $0 --regex regex [--invert] [--ignore-case] [file...]\n";
~~my %Opt;~~
~~getopts( 'ir:v', \%Opt ) or die $USAGE;~~


~~my ($negated, $case_insensitive) = @Opt{ qw(v i) };~~
my ($regex, $negated, $case_insensitive);
~~my $regex = $Opt{r} or die $USAGE;~~

GetOptions( 'regex=s'     => \$regex,
            'invert'      => \$negated,
            'ignore-case' => \$case_insensitive
          ) or die $USAGE;


while ( <> )
{
  if ( $case_insensitive )
  {
    print if $negated xor /$regex/i;
  }
  else
  {
    print if $negated xor /$regex/;
  }
}
```

Check Syntax and run it by typing the commands below as shown:

CODE TO TYPE:

```
cold:~/perl3$ ./grep_long.pl --invert --regex e ./test_input
This string has an oddity.  Can you suss out what it is?
cold:~/perl3$ ./grep_long.pl --regex e ./test_input
This sentence is more conventional.
cold:~/perl3$ ./grep_long.pl --invert --ignore-case --regex E ./test_input
This string has an oddity.  Can you suss out what it is?
```

Here's how it works. **Getopt::Long** provides the function **GetOptions()**, which takes a list of arguments:

OBSERVE:

```
GetOptions( $option_specifier, $scalar_ref, ... )
```

Each pair of arguments in the list is a string describing a command-line option, followed by a reference to a scalar in which to store the value if the option is present on the command line.

For options that don't take values (boolean options), the specifier is just the name of the option, that is, whatever follows **--**. Options that take values are denoted by adding certain strings after the name of the option:

Suffix	Meaning	Example
=s	Required: Any string	--name=scott

=i	Required: Integer only	--course=3
=f	Required: Floating point number only	--balance=170.50
:s	Value is optional: string	--owner
:i	Value is optional: integer	--dependents
:f	Value is optional: floating-point number	--debit

You can shorten option names to their shortest unambiguous prefixes. Go ahead and type the commands as shown:

CODE TO TYPE:

```
cold:~/perl3$ ./grep_long.pl --invert --ignore-case --regex=E ./test_input
This string has an oddity.  Can you suss out what it is?
cold:~/perl3$ ./grep_long.pl --in --ig --r=E ./test_input
This string has an oddity.  Can you suss out what it is?
```

The above lines of code are equivalent.

You can also give synonyms for options with a vertical bar between them; for example:

OBSERVE:

```
'ignore_case|case-insensitive' => \$case_insensitive
```

And you can specify that an option may take a fixed number or arbitrary number of values rather than just one. Try this example. Create **opt_long.pl**:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

use Getopt::Long;

my $size = 42;
GetOptions( 'library=s@'       => \my @libraries,
            'coordinates=f{2}' => \my @coordinates,
            'size:i'           => \$size,
          );

print join( "\n\t", "Libraries:", @libraries ), "\n";
print join( "\n\t", "Coordinates:", @coordinates ), "\n";
print "Size: $size\n";
print "Remaining in \@ARGV: @ARGV\n";
```

Save and run that program. You'll see this:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./opt_long.pl --lib=/etc --lib /var/tmp --lib /tmp --coordinates 1.4 -2.5
 --size
Libraries:
        /etc
        /var/tmp
        /tmp
Coordinates:
        1.4
        -2.5
Size: 0
Remaining in @ARGV:
```

We save a line by putting the declarations of **@libraries** and **@coordinates** *inside* the call to **GetOptions()**. That may seems strange, but bear in mind that **my** is basically just a kind of qualifier on a variable and can appear just about anywhere you use a variable. There is no scoping issue because parentheses don't impose a scoping level; braces do.

Also, we don't need to provide an argument to **--size**; the result is that the value is set to **0**. If we omit the option altogether, we get this:

INTERACTIVE SESSION:

```
cold:~/perl3$  ./opt_long.pl --lib=/etc --lib /var/tmp --lib /tmp --coordinates 1.4 -2.
5 3.4
Libraries:
        /etc
        /var/tmp
        /tmp
Coordinates:
        1.4
        -2.5
Size: 42
Remaining in @ARGV: 3.4
```

...and then it reverts to the default we provided. A third value for the **--coordinates** option is ignored because we specified that it takes exactly two. Try providing only one:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./opt_long.pl --lib /var/tmp /tmp --coordinates 1.4
Insufficient arguments for option coordinates
Libraries:
        /var/tmp
Coordinates:
        1.4
Size: 42
Remaining in @ARGV: /tmp
```

The **--library** option must be given for each of its values; **--library** does not work the same way as explicitly quantified **--coordinates**.

There are *many* other features of **GetOptions()**; for more information on them, see perldoc Getopt::Long. Just one lesson to go, can you believe it? I can. You're doing some great work! See you in lesson 15...

Once you finish the lesson, go back to the syllabus to complete the homework.

Final Wrap-Up

Lesson Objectives

When you complete this lesson, you will be able to:

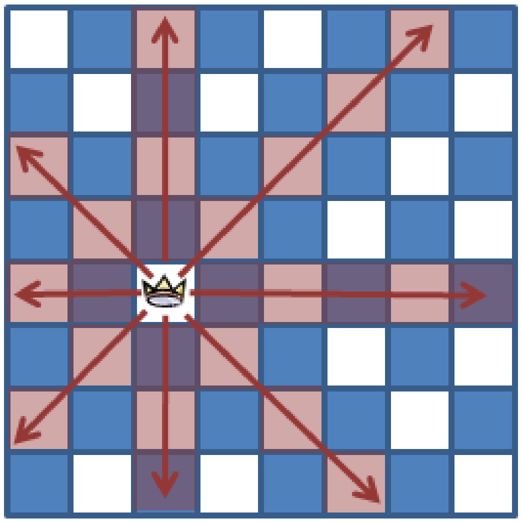
- write a program to solve the *Eight Queens Problem*.

Welcome to the final lesson of **Advanced Perl!** You've come a long way since the beginning of this course; take a moment to review and reflect on everything you've learned. You've made some serious progress.

Putting It All Together

The Eight Queens Problem

In this lesson, we'll write a program to solve the *Eight Queens Problem*. The *Eight Queens Problem* is a traditional programming exercise. Our goal is to print all the ways that 8 queens can be placed on a chess board such that no queen can attack another queen. We will ignore any solution that is a reflection (about the vertical or horizontal dividing lines) or rotation (by 90, 180, or 270 degrees) of one that has already been printed. Queens attack horizontally, vertically, and diagonally:



We'll develop this program incrementally, applying lessons we've learned throughout the course. The strategy we'll use, while not the most efficient one out there, is clear and easy to understand. We'll use a *recursive* approach, understanding that there must be a queen in every row: For each of the 8 squares in the first row. Then we'll place a queen on the first square in the next row that meets our conditions, and so on, until we have either placed 8 queens (in which case we have a solution provided it isn't a transformation of an existing one) or we have run out of places to place another queen. Here we go! Create **queens.pl** in your **/perl3** folder as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

run( 8 );

my $TEMP_COUNTER;

sub run
{
  my $size = shift;

  my $solution = [];
  my $row = 0;
  try( $size-1, $solution, $row );
}

sub try
{
  my ($max_cell, $solution, $row) = @_;

  if ( $row > $max_cell )
  {
    print_solution( $max_cell, $solution ) if unique( $solution );
    exit if $TEMP_COUNTER++ > 15;
  }
  else
  {
    for my $column ( 0 .. $max_cell )
    {
      next if attacked( $solution, $row, $column );
      my $new_solution = copy( $solution );
      put_queen_at( $new_solution, $row, $column );
      try( $max_cell, $new_solution, $row + 1 );
    }
  }
}

sub unique
{
  return 1;  # Temporary
}

sub attacked
{
  return 0;  # Temporary
}

sub put_queen_at
{
  my ($solution, $row, $column) = @_;

  $solution->[$row][$column] = 'Q';
}

sub copy
{
  my $solution = shift;

  my @rows;
  push @rows, [ @$_ ] for @$solution;
  return \@rows;
}

sub print_solution
{
  my ($max_cell, $solution) = @_;

  for my $row ( @$solution )
  {
    for my $column ( 0 .. $max_cell )
    {
      print $row->[$column] ? " Q " : " . ";
    }
    print "\n";
  }
  print "\n";  # Extra blank line for spacing
}
```

Check Syntax and run it; this is the most basic program we could create and still get some output. This program will end up being about 200 lines long, but you'll be able to manage it by breaking it down into smaller chunks. Run it by typing the commands you see below:

INTERACTIVE SESSION:

```
cold:~$ cd perl3
cold:~/perl3$ ./queens.pl
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 

 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 .  Q  .  .  .  .  .  . 

 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 .  .  Q  .  .  .  .  . 

 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 Q  .  .  .  .  .  .  . 
 .  .  .  Q  .  .  .  . 
  (results abbreviated to save space)
```

Here's what's going on: we parameterize the size of the board from the beginning, because we don't want or need to hard-code the number 8 throughout the program. Parameterizing will allow us to solve the problem for other board sizes later as well.

OBSERVE:

```
sub run
{
  my $size = shift;

  my $solution = [];
  my $row = 0;
  try( $size-1, $solution, $row );
}
```

We represent a (potential or actual) **solution()** with an arrayref. It will start out empty, as in **run**. Each time we put a queen in it, we will autovivify it by treating the arrayref as if it were pointing to a 2-D array (see **put_queen_at**).

OBSERVE:

```
sub try
{
  my ($max_cell, $solution, $row) = @_;

  if ( $row > $max_cell )
  {
    print_solution( $max_cell, $solution ) if unique( $solution );
    exit if $TEMP_COUNTER++ > 15;
  }
  else
  {
    for my $column ( 0 .. $max_cell )
    {
      next if attacked( $solution, $row, $column );
      my $new_solution = copy( $solution );
      put_queen_at( $new_solution, $row, $column );
      try( $max_cell, $new_solution, $row + 1 );
    }
  }
}
```

try is the recursive routine. It's first called on row number 0 from **run**, and will **call itself with the next higher row**. **$max_cell** is set to one less than the size of the board. When **try** is called with **$row > $max_cell** (in this case, $row = 8), the solution being passed must be full (because it now has rows 0 through 7 populated with queens), so we **test the solution to see if it is unique** (not a rotation or reflection of an existing one), and if so, **print it**.

If we're in the middle of recursing (**$row *is not* > $max_cell**), we **iterate through each column in the current row**. We see whether a queen at the proposed location would be attacked by an existing one, and if it would, we **loop**. Otherwise we make a **new solution that is a copy** of the one we're working on, so that we can pass that to **try** without that call altering the solution we're currently working toward. Then we add the queen to the new solution and recurse.

OBSERVE:

```
sub print_solution
{
  my ($max_cell, $solution) = @_;

  for my $row ( @$solution )
  {
    for my $column ( 0 .. $max_cell )
    {
      print $row->[$column] ? " Q " : " . ";
    }
    print "\n";
  }
  print "\n";  # Extra blank line for spacing
}
```

Because each row of a solution will only have as many elements as the last queen in each row, **print_solution** needs to know the size of the board (the number of **rows** multiplied by number of **columns**) so it can **print()** the full width of each row.

OBSERVE:

```
sub copy
{
  my $solution = shift;

  my @rows;
  push @rows, [ @$_ ] for @$solution;
  return \@rows;
}
```

The **copy** subroutine has to make a copy of a 2-D array, so it starts with a new array **@rows** and for each row in the **solution**, pushes onto it an anonymous arrayref containing the contents of that row; study that line until you understand how that works. You'll make deep copies or "clones" of data structures frequently.

The routines that tell whether a solution is unique or whether a queen is attacked by another are stubbed out. So we added a counter to limit the number of "solutions" printed; otherwise, our program would print every possible permutation of 8 queens on the board.

Now let's rework our program to figure out whether a configuration is valid. Modify **queens.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

run( 8 );

my $TEMP_COUNTER;

sub run
{
  my $size = shift;

  my $solution = [];
  my $row = 0;
  try( $size-1, $solution, $row );
}

sub try
{
  my ($max_cell, $solution, $row) = @_;

  if ( $row > $max_cell )
  {
    print_solution( $max_cell, $solution ) if unique( $solution );
    ~~exit if $TEMP_COUNTER++ > 15;~~
    print " *** ", ++$TEMP_COUNTER, "\n";
  }
  else
  {
    for my $column ( 0 .. $max_cell )
    {
      next if attacked( $solution, $row, $column );
      my $new_solution = copy( $solution );
      put_queen_at( $new_solution, $row, $column );
      try( $max_cell, $new_solution, $row + 1 );
    }
  }
}

sub unique
{
  return 1;  # Temporary
}

sub attacked  # Is queen at proposed position attacked in this solution?
{
  my ($solution, $proposed_row, $proposed_column) = @_;

  for my $row ( 0 .. $#$solution )
  {
    my $elements_ref = $solution->[$row];
    for my $column ( grep { defined $solution->[$row][$_] } 0 .. $#$elements_ref
 )
    {
      return 1 if attacks( $row, $column, $proposed_row, $proposed_column );
    }
  }
  return 0;
}

sub attacks
{
  my ($r1, $c1, $r2, $c2) = @_;

  my $row_diff = abs( $r1 - $r2 );
  my $col_diff = abs( $c1 - $c2 );
  return $row_diff == $col_diff || $row_diff == 0 || $col_diff == 0;
}

sub put_queen_at
{
  my ($solution, $row, $column) = @_;

  $solution->[$row][$column] = 'Q';
}

sub copy
{
  my $solution = shift;

  my @rows;
  push @rows, [ @$_ ] for @$solution;
  return \@rows;
}

sub print_solution
{
  my ($max_cell, $solution) = @_;

  for my $row ( @$solution )
  {
    for my $column ( 0 .. $max_cell )
    {
      print $row->[$column] ? " Q " : " . ";
    }
    print "\n";
  }
  print "\n";  # Extra blank line for spacing
}
```

Check Syntax and run it. The output starts like this:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./queens.pl
 Q  .  .  .  .  .  .  . 
 .  .  .  .  Q  .  .  . 
 .  .  .  .  .  .  .  Q 
 .  .  .  .  .  Q  .  . 
 .  .  Q  .  .  .  .  . 
 .  .  .  .  .  .  Q  . 
 .  Q  .  .  .  .  .  . 
 .  .  .  Q  .  .  .  . 

 *** 1
```

We're printing the counter out now and no longer limiting it, because now the program prints only the valid solutions, including rotations and reflections; you see 92 of them.

To fully understand this change, first let's look at the **attacks** subroutine:

OBSERVE:

```
sub attacks
{
  my ($r1, $c1, $r2, $c2) = @_;

  my $row_diff = abs( $r1 - $r2 );
  my $col_diff = abs( $c1 - $c2 );
  return $row_diff == $col_diff || $row_diff == 0 || $col_diff == 0;
}
```

The **attacks** subroutine takes two (**row**, **column**) pairs, representing two queens, and determines whether they can attack each other. We compute the differences between the two rows and the two columns; **if either one is zero**, the two pieces can attack horizontally or vertically. **If the absolute differences are equal**, the two pieces can attack along a diagonal (try a couple of examples).

Now let's look at the **attacked** subroutine:

OBSERVE:

```
sub attacked  # Is queen at proposed position attacked in this solution?
{
  my ($solution, $proposed_row, $proposed_column) = @_;

  for my $row ( 0 .. $#$solution )
  {
    my $elements_ref = $solution->[$row];
    for my $column ( grep { defined $solution->[$row][$_] } 0 .. $#$elements_ref
 )
    {
      return 1 if attacks( $row, $column, $proposed_row, $proposed_column );
    }
  }
  return 0;
}
```

It goes through the position of every queen currently in the solution and calls **attacks** with that position and the position of the candidate queen. This code looks pretty challenging; remember that **$solution** is a reference to an array of references to arrays that contain either **undef** or **Q** in each slot.

Now we'll add code to check for rotations and reflections. Let's do that in stages. Modify **queens.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

run( 8 );

my $TEMP_COUNTER;
my @Solutions;

sub run
{
  my $size = shift;

  my $solution = [];
  my $row = 0;
  try( $size-1, $solution, $row );
}

sub try
{
  my ($max_cell, $solution, $row) = @_;

  if ( $row > $max_cell )
  {
    ~~print_solution( $max_cell, $solution ) if unique( $solution );~~
    return unless unique( $solution );
    print_solution( $max_cell, $solution );
    print " *** ", ++$TEMP_COUNTER, "\n";
    push @Solutions, $solution;
  }
  else
  {
    for my $column ( 0 .. $max_cell )
    {
      next if attacked( $solution, $row, $column );
      my $new_solution = copy( $solution );
      put_queen_at( $new_solution, $row, $column );
      try( $max_cell, $new_solution, $row + 1 );
    }
  }
}

sub unique
{
  my $solution = shift;

  for my $so_far ( @Solutions )
  {
    return 0 if is_like( $so_far, $solution );
  }
  return 1;~~   # Temporary~~
}

sub is_like
{
  return 1;   # Temporary
}

sub attacked  # Is queen at proposed position attacked in this solution?
{
  my ($solution, $proposed_row, $proposed_column) = @_;

  for my $row ( 0 .. $#$solution )
  {
    my $elements_ref = $solution->[$row];
    for my $column ( grep { defined $solution->[$row][$_] } 0 .. $#$elements_ref
 )
    {
      return 1 if attacks( $row, $column, $proposed_row, $proposed_column );
    }
  }
  return 0;
}
sub attacks
{
  my ($r1, $c1, $r2, $c2) = @_;

  my $row_diff = abs( $r1 - $r2 );
  my $col_diff = abs( $c1 - $c2 );
  return $row_diff == $col_diff || $row_diff == 0 || $col_diff == 0;
}

sub put_queen_at
{
  my ($solution, $row, $column) = @_;

  $solution->[$row][$column] = 'Q';
}

sub copy
{
  my $solution = shift;

  my @rows;
  push @rows, [ @$_ ] for @$solution;
  return \@rows;
}

sub print_solution
{
  my ($max_cell, $solution) = @_;

  for my $row ( @$solution )
  {
    for my $column ( 0 .. $max_cell )
    {
      print $row->[$column] ? " Q " : " . ";
    }
    print "\n";
  }
  print "\n";  # Extra blank line for spacing
}
```

Check Syntax and run it. Now it prints only one solution. We're not done yet; I just didn't want to add too much all at once. Here's what's going on: we've added a global array **@Solutions** to keep track of the ones we've found so far. Whenever we have a candidate solution, we see if it's unique, and if so, print it and add it to **@Solutions**. **unique** compares the candidate solution against all of those in **@Solutions**; only the first one gets printed because the stub for **is_like** is always true.

Now let's add the rest of the code. Modify **queens.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

init( 8 );
run( 8 );

my @Transforms;
my $TEMP_COUNTER;
my @Solutions;

sub run
{
  my $size = shift;

  my $solution = [];
  my $row = 0;
  try( $size-1, $solution, $row );
}

sub init
{
  my $size = shift;
  my $max = $size - 1;

  push @Transforms,
    sub { (        $_[1],        $_[0] ) },
    sub { ( $max - $_[1], $max - $_[0] ) },
    sub { ( $max - $_[0],        $_[1] ) },
    sub { (        $_[0], $max - $_[1] ) },
    sub { ( $max - $_[1],        $_[0] ) },
    sub { ( $max - $_[0], $max - $_[1] ) },
    sub { (        $_[1], $max - $_[0] ) };
}

sub try
{
  my ($max_cell, $solution, $row) = @_;

  if ( $row > $max_cell )
  {
    return unless unique( $solution );
    print_solution( $max_cell, $solution );
    print " *** ", ++$TEMP_COUNTER, "\n";
    push @Solutions, $solution;
  }
  else
  {
    for my $column ( 0 .. $max_cell )
    {
      next if attacked( $solution, $row, $column );
      my $new_solution = copy( $solution );
      put_queen_at( $new_solution, $row, $column );
      try( $max_cell, $new_solution, $row + 1 );
    }
  }
}

sub unique
{
  my $solution = shift;

  for my $so_far ( @Solutions )
  {
    return 0 if is_like( $so_far, $solution );
  }
  return 1;
}

sub is_like
{
  my ($sol1, $sol2) = @_;

  for my $func ( @Transforms )
  {
    return 1 if same( $sol1, transform( $func, $sol2 ) )
  }
  return 0;
~~  return 1;   # Temporary~~
}

sub same
{
  my ($sol1, $sol2) = @_;

  return string_of( $sol1 ) eq string_of( $sol2 );
}

sub transform
{
  my ($func, $solution) = @_;

  my $new_solution = [];
  for my $row ( 0 .. $#$solution )
  {
    my $elements_ref = $solution->[$row];
    for my $column ( grep { defined $solution->[$row][$_] } 0 .. $#$elements_ref
 )
    {
      my ($new_row, $new_column) = $func->( $row, $column );
      put_queen_at( $new_solution, $new_row, $new_column );
    }
  }
  return $new_solution;
}

sub string_of
{
  my $solution = shift;

  my $string = '';
  for my $row ( @$solution )
  {
    $string .= join '', map { defined($_) ? 'Q' : ' ' } @$row;
    $string .= "x";
  }

  return $string;
}

sub attacked  # Is queen at proposed position attacked in this solution?
{
  my ($solution, $proposed_row, $proposed_column) = @_;

  for my $row ( 0 .. $#$solution )
  {
    my $elements_ref = $solution->[$row];
    for my $column ( grep { defined $solution->[$row][$_] } 0 .. $#$elements_ref
 )
    {
      return 1 if attacks( $row, $column, $proposed_row, $proposed_column );
    }
  }
  return 0;
```

```
}

sub attacks
{
  my ($r1, $c1, $r2, $c2) = @_;

  my $row_diff = abs( $r1 - $r2 );
  my $col_diff = abs( $c1 - $c2 );
  return $row_diff == $col_diff || $row_diff == 0 || $col_diff == 0;
}

sub put_queen_at
{
  my ($solution, $row, $column) = @_;

  $solution->[$row][$column] = 'Q';
}

sub copy
{
  my $solution = shift;

  my @rows;
  push @rows, [ @$_ ] for @$solution;
  return \@rows;
}

sub print_solution
{
  my ($max_cell, $solution) = @_;

  for my $row ( @$solution )
  {
    for my $column ( 0 .. $max_cell )
    {
      print $row->[$column] ? " Q " : " . ";
    }
    print "\n";
  }
  print "\n";  # Extra blank line for spacing
}
```

Check Syntax and run it. It prints 12 solutions, then pauses for a bit while it checks some redundant possibilities before finishing. Okay, now that's a big change! Let's break it down piece by piece. First, the **string_of** subroutine:

OBSERVE:

```
sub string_of
{
  my $solution = shift;

  my $string = '';
  for my $row ( @$solution )
  {
    $string .= join '', map { defined($_) ? 'Q' : ' ' } @$row;
    $string .= "x";
  }

  return $string;
}
```

string_of() generates a **string** from a **solution** so we can compare two solutions with **eq** in the **same()** subroutine below. It makes a representation of each row and joins them together with the letter **x**.

OBSERVE:

```
sub same
{
  my ($sol1, $sol2) = @_;

  return string_of( $sol1 ) eq string_of( $sol2 );
}
```

The **same()** routine uses the **string_of** results to check whether two **solutions** are identical.

OBSERVE:

```
sub is_like
{
  my ($sol1, $sol2) = @_;

  for my $func ( @Transforms )
  {
    return 1 if same( $sol1, transform( $func, $sol2 ) )
  }
  return 0;
}
```

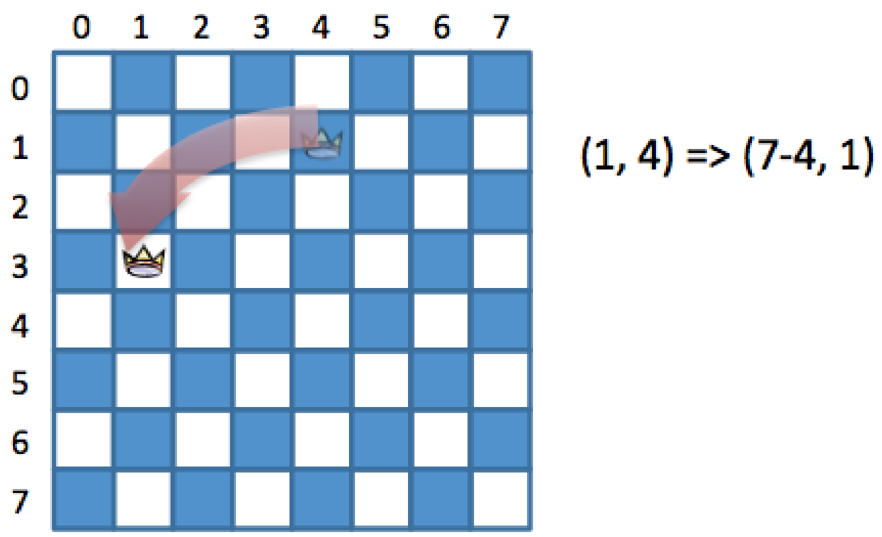
is_like() now iterates through each of the seven **transformations** representing rotations and reflections (more on that in a moment), **transforms** one of the **solutions** and determines **if they are the same**; that is, whether the result matches the other solution.

OBSERVE:

```
sub transform
{
  my ($func, $solution) = @_;

  my $new_solution = [];
  for my $row ( 0 .. $#$solution )
  {
    my $elements_ref = $solution->[$row];
    for my $column ( grep { defined $solution->[$row][$_] } 0 .. $#$elements_ref
 )
    {
      my ($new_row, $new_column) = $func->( $row, $column );
      put_queen_at( $new_solution, $new_row, $new_column );
    }
  }
  return $new_solution;
}
```

The **transform()** subroutine creates a **new blank solution** and then **iterates through the queens** in the solution that is passed, calling the transformation routine that was passed to **insert a queen into the new solution** at the transformed location. We're using the same looping code we used in **attacked** to do the iteration.

Each transformation routine takes two arguments, a row and a column, and returns a row and column that represent where a queen at that row and column winds up after the transformation. There are 7 such transformations, and they are put into the array **@Transforms** by the new **init** routine. For example, the list **($max - $_[1], $_[0])** represents this transformation:



That is, a 90-degree counter-clockwise rotation.

The coderefs in **@Transforms** are *closed over* **$max**. This means they are used in the subroutine, but defined outside if it.

We now have a working program! But it's not polished yet. For one thing, we have duplicated code between the **attacked** and **transform** routines. We can remove that by using a *callback*. Modify **queens.pl** like this:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

init( 8 );
run( 8 );

my @Transforms;
my $TEMP_COUNTER;
my @Solutions;

sub run
{
  my $size = shift;

  my $solution = [];
  my $row = 0;
  try( $size-1, $solution, $row );
}

sub init
{
  my $size = shift;
  my $max = $size - 1;

  push @Transforms,
    sub { (        $_[1],        $_[0] ) },
    sub { ( $max - $_[1], $max - $_[0] ) },
    sub { ( $max - $_[0],        $_[1] ) },
    sub { (        $_[0], $max - $_[1] ) },
    sub { ( $max - $_[1],        $_[0] ) },
    sub { ( $max - $_[0], $max - $_[1] ) },
    sub { (        $_[1], $max - $_[0] ) };
}

sub try
{
  my ($max_cell, $solution, $row) = @_;

  if ( $row > $max_cell )
  {
    return unless unique( $solution );
    print_solution( $max_cell, $solution );
    print " *** ", ++$TEMP_COUNTER, "\n";
    push @Solutions, $solution;
  }
  else
  {
    for my $column ( 0 .. $max_cell )
    {
      next if attacked( $solution, $row, $column );
      my $new_solution = copy( $solution );
      put_queen_at( $new_solution, $row, $column );
      try( $max_cell, $new_solution, $row + 1 );
    }
  }
}

sub unique
{
  my $solution = shift;

  for my $so_far ( @Solutions )
  {
    return 0 if is_like( $so_far, $solution );
  }
  return 1;
}

sub is_like
{
  my ($sol1, $sol2) = @_;

  for my $func ( @Transforms )
  {
    return 1 if same( $sol1, transform( $func, $sol2 ) )
  }
  return 0;
}

sub same
{
  my ($sol1, $sol2) = @_;

  return string_of( $sol1 ) eq string_of( $sol2 );
}

sub transform
{
  my ($func, $solution) = @_;

  my $new_solution = [];
  ~~for my $row ( 0 .. $#$solution )~~
~~  {~~
~~    my $elements_ref = $solution->[$row];~~
~~    for my $column ( grep { defined $solution->[$row][$_] } 0 .. $#$elements_ref~~
~~ )~~
~~    {~~
~~      my ($new_row, $new_column) = $func->( $row, $column );~~
~~      put_queen_at( $new_solution, $new_row, $new_column );~~
~~    }~~
~~  }~~
  for_row_col( $solution, sub {
        my ($row, $column) = @_;
        my ($new_row, $new_column) = $func->( $row, $column );
        put_queen_at( $new_solution, $new_row, $new_column );
      } );
  return $new_solution;
}

sub for_row_col
{
  my ($solution, $callback) = @_;

  for my $row ( 0 .. $#$solution )
  {
    my $elements_ref = $solution->[$row];
    for my $column ( grep { defined $solution->[$row][$_] } 0 .. $#$elements_ref
 )
    {
      $callback->( $row, $column );
    }
  }
}

sub string_of
{
  my $solution = shift;

  my $string = '';
  for my $row ( @$solution)
  {
    $string .= join '', map { defined($_) ? 'Q' : ' ' } @$row;
    $string .= "x";
  }

  return $string;
}

sub attacked  # Is queen at proposed position attacked in this solution?
{
  my ($solution, $proposed_row, $proposed_column) = @_;

  ~~for my $row ( 0 .. $#$solution )~~
~~  {~~
~~    my $elements_ref = $solution->[$row];~~
~~    for my $column ( grep { defined $solution->[$row][$_] } 0 .. $#$elements_ref~~
~~ )~~
~~    {~~
~~      return 1 if attacks( $row, $column, $proposed_row, $proposed_column );~~
~~    }~~
~~  }~~
~~  return 0;~~
  eval {
    for_row_col( $solution, sub {
        my ($row, $column) = @_;
        die 1 if attacks( $row, $column, $proposed_row, $proposed_column );
      } );
  };
  return $@;
}

sub attacks
{
  my ($r1, $c1, $r2, $c2) = @_;

  my $row_diff = abs( $r1 - $r2 );
  my $col_diff = abs( $c1 - $c2 );
  return $row_diff == $col_diff || $row_diff == 0 || $col_diff == 0;
}

sub put_queen_at
{
  my ($solution, $row, $column) = @_;

  $solution->[$row][$column] = 'Q';
}

sub copy
{
  my $solution = shift;

  my @rows;
  push @rows, [ @$_ ] for @$solution;
  return \@rows;
}

sub print_solution
{
  my ($max_cell, $solution) = @_;

  for my $row ( @$solution )
  {
    for my $column ( 0 .. $max_cell )
    {
      print $row->[$column] ? " Q " : " . ";
    }
    print "\n";
  }
  print "\n";  # Extra blank line for spacing
}
```

Check Syntax and run it; the output will be the same as before. Let's go over the **for_row_col** subroutine first:

OBSERVE:

```
sub for_row_col
{
  my ($solution, $callback) = @_;

  for my $row ( 0 .. $#$solution )
  {
    my $elements_ref = $solution->[$row];
    for my $column ( grep { defined $solution->[$row][$_] } 0 .. $#$elements_ref
 )
    {
      $callback->( $row, $column );
    }
  }
}
```

for_row_col performs the same iteration that we duplicated before in the **transform** and **attacked** routines, but calls a callback inside the loop.

Now look at the **transform()** subroutine:

OBSERVE:

```
sub transform
{
  my ($func, $solution) = @_;

  my $new_solution = [];
  for_row_col( $solution, sub {
        my ($row, $column) = @_;
        my ($new_row, $new_column) = $func->( $row, $column );
        put_queen_at( $new_solution, $new_row, $new_column );
      } );
  return $new_solution;
}
```

transform() calls **for_row_col**, passing a coderef that does the same thing inside the loop that it did before. The callback is closed over **$func**, **$solution**, and **$new_solution**.

Now let's look at the **attacked** subroutine:

OBSERVE:

```
sub attacked  # Is queen at proposed position attacked in this solution?
{
  my ($solution, $proposed_row, $proposed_column) = @_;

  eval {
    for_row_col( $solution, sub {
        my ($row, $column) = @_;
        die 1 if attacks( $row, $column, $proposed_row, $proposed_column );
      } );
  };
  return $@;
}
```

The code that was originally inside the loop **return**s from **attacked** if the result of **attacks** is true. But putting **return** inside a callback will do no good; it will just return from the callback. We need a way of doing something similar—jumping out of the callback at that point. This suggests that we should raise an exception, so we'll use **die** instead of **return**, and wrap the call to **for_row_col** in an **eval** block, so that we can test **$@** afterward. If it is no longer undef, then the code **die**d.

We may not have reduced the line count, but we have removed duplication of functionality, so that if we change the way we iterate through a solution in the future, we only need to change it in one place.

We can still simplify our code even more. The result of **string_of** is very close to the output of **print_solution**; if we made **string_of** return a printable board representation, we could eliminate **print_solution** altogether. And while we're at it, let's make the board size an option. We can do all of that. Modify **queens.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

use Getopt::Long;

my $size = 8;
GetOptions( 'size:i' => \$size );

init( ~~8~~ $size );
run( ~~8~~ $size );

my @Transforms;
~~my $TEMP_COUNTER;~~
my @Solutions;
my $String_Func;

sub run
{
  my $size = shift;

  my $solution = [];
  my $row = 0;
  try( $size-1, $solution, $row );
}

sub init
{
  my $size = shift;
  my $max = $size - 1;

  push @Transforms,
    sub { (        $_[1],        $_[0] ) },
    sub { ( $max - $_[1], $max - $_[0] ) },
    sub { ( $max - $_[0],        $_[1] ) },
    sub { (        $_[0], $max - $_[1] ) },
    sub { ( $max - $_[1],        $_[0] ) },
    sub { ( $max - $_[0], $max - $_[1] ) },
    sub { (        $_[1], $max - $_[0] ) };

  $String_Func = sub { string_of( $max, @_ ) };
}

sub try
{
  my ($max_cell, $solution, $row) = @_;

  if ( $row > $max_cell )
  {
    return unless unique( $solution );
    ~~print " *** ", ++$TEMP_COUNTER, "\n";~~
    ~~print_solution( $max_cell, $solution );~~
    print $String_Func->( $solution );
    push @Solutions, $solution;
  }
  else
  {
    for my $column ( 0 .. $max_cell )
    {
      next if attacked( $solution, $row, $column );
      my $new_solution = copy( $solution );
      put_queen_at( $new_solution, $row, $column );
      try( $max_cell, $new_solution, $row + 1 );
    }
  }
}
```

```
sub unique
{
  my $solution = shift;

  for my $so_far ( @Solutions )
  {
    return 0 if is_like( $so_far, $solution );
  }
  return 1;
}

sub is_like
{
  my ($sol1, $sol2) = @_;

  for my $func ( @Transforms )
  {
    return 1 if same( $sol1, transform( $func, $sol2 ) )
  }
  return 0;
}

sub same
{
  my ($sol1, $sol2) = @_;

  return ~~string_of( $sol1 ) eq string_of( $sol2 );~~ $String_Func->( $sol1 ) eq $S
tring_Func->( $sol2 );
}

sub transform
{
  my ($func, $solution) = @_;

  my $new_solution = [];
  for_row_col( $solution, sub {
        my ($row, $column) = @_;
        my ($new_row, $new_column) = $func->( $row, $column );
        put_queen_at( $new_solution, $new_row, $new_column );
      } );
  return $new_solution;
}


sub for_row_col
{
  my ($solution, $callback) = @_;

  for my $row ( 0 .. $#$solution )
  {
    my $elements_ref = $solution->[$row];
    for my $column ( grep { defined $solution->[$row][$_] } 0 .. $#$elements_ref
 )
    {
      $callback->( $row, $column );
    }
  }
}

sub string_of
{
~~  my $solution = shift;~~
  my ($max_cell, $solution) = @_;

  my $string = '';
  for my $row ( @$solution )
  {
```

```
~~    $string .= join '', map { defined($_) ? 'Q' : ' ' } @$row;~~
~~    $string .= "x";~~
    my @temp = @$row;
    $#temp = $max_cell;
    $string .= join '', map { defined($_) ? ' Q ' : ' . ' } @temp;
    $string .= "\n";
  }

~~  return $string;~~
  return "$string\n";  # Extra newline for spacing
}

sub attacked  # Is queen at proposed position attacked in this solution?
{
  my ($solution, $proposed_row, $proposed_column) = @_;

  eval {
    for_row_col( $solution, sub {
        my ($row, $column) = @_;
        die 1 if attacks( $row, $column, $proposed_row, $proposed_column );
      } );
  };
  return $@;
}

sub attacks
{
  my ($r1, $c1, $r2, $c2) = @_;

  my $row_diff = abs( $r1 - $r2 );
  my $col_diff = abs( $c1 - $c2 );
  return $row_diff == $col_diff || $row_diff == 0 || $col_diff == 0;
}

sub put_queen_at
{
  my ($solution, $row, $column) = @_;

  $solution->[$row][$column] = 'Q';
}

sub copy
{
  my $solution = shift;

  my @rows;
  push @rows, [ @$_ ] for @$solution;
  return \@rows;
}

~~sub print_solution~~
~~{~~
~~  my ($max_cell, $solution) = @_;~~

~~  for my $row ( @$solution )~~
~~  {~~
~~    for my $column ( 0 .. $max_cell )~~
~~    {~~
~~      print $row->[$column] ? " Q " : " . ";~~
~~    }~~
~~    print "\n";~~
~~  }~~
~~  print "\n";  # Extra blank line for spacing~~
~~}~~
```

Check Syntax and run it as shown:

INTERACTIVE SESSION:

```
cold:~/perl3$ ./queens.pl --size=7
 Q  .  .  .  .  .  .
 .  .  Q  .  .  .  .
 .  .  .  .  Q  .  .
 .  .  .  .  .  .  Q
 .  Q  .  .  .  .  .
 .  .  .  Q  .  .  .
 .  .  .  .  .  Q  .

 Q  .  .  .  .  .  .
 .  .  .  Q  .  .  .
 .  .  .  .  .  .  Q
 .  .  Q  .  .  .  .
 .  .  .  .  .  Q  .
 .  Q  .  .  .  .  .
 .  .  .  .  Q  .  .

 .  Q  .  .  .  .  .
 .  .  .  Q  .  .  .
 Q  .  .  .  .  .  .
 .  .  .  .  .  .  Q
 .  .  .  .  Q  .  .
 .  .  Q  .  .  .  .
 .  .  .  .  .  Q  .

 .  Q  .  .  .  .  .
 .  .  .  .  Q  .  .
 Q  .  .  .  .  .  .
 .  .  .  Q  .  .  .
 .  .  .  .  .  .  Q
 .  .  Q  .  .  .  .
 .  .  .  .  .  Q  .

 .  Q  .  .  .  .  .
 .  .  .  .  Q  .  .
 .  .  .  .  .  .  Q
 .  .  .  Q  .  .  .
 Q  .  .  .  .  .  .
 .  .  Q  .  .  .  .
 .  .  .  .  .  Q  .

 .  Q  .  .  .  .  .
 .  .  .  .  .  Q  .
 .  .  Q  .  .  .  .
 .  .  .  .  .  .  Q
 .  .  .  Q  .  .  .
 Q  .  .  .  .  .  .
 .  .  .  .  Q  .  .
```

(We ran it for a different-sized board—the default size is 8—to reduce the amount of output.) We created the function to get the string representation through *currying;* the **string_of** routine now needs to know the width of a row in order to print out cells all the way across (otherwise it will stop at the last queen). But passing the **$max_cell** parameter to it would mean that we'd need to pass that parameter to **same**, which would mean we'd have to pass it to **is_like**, and that's just plain tedious! Instead, we created a reference in **init** to a routine that calls **string_of** with the **$max_cell** parameter, since the variable **$max_cell** is accessible inside the routine **init**.

Even though we added the option parsing code, the program is still shorter, weighing in at around 175 lines. That's pretty good, considering all that it does.

Congratulations!

That was a fairly long and complicated example, but I knew you'd be up to the challenge! It's good to have gone through a typical development cycle and see the stages of development for a complex program like that.

And congratulations on reaching the end of **Advanced Perl**! You have learned some truly important skills that are way beyond those of the average programmer. We encourage you to go on to our fourth Perl course, where you'll learn how to apply those skills to practical domains such as sending email, visiting websites, processing web forms, and querying databases.

Now pause for a moment. Revel in your accomplishments, and give yourself a well-deserved pat on the back! It's been great working with you, I hope to work with you again soon. Thanks for being here and for your commitment to becoming a conscientious and excellent Perl programmer! Good luck with your final project and beyond!

Once you finish the lesson, go back to the syllabus to complete the homework.