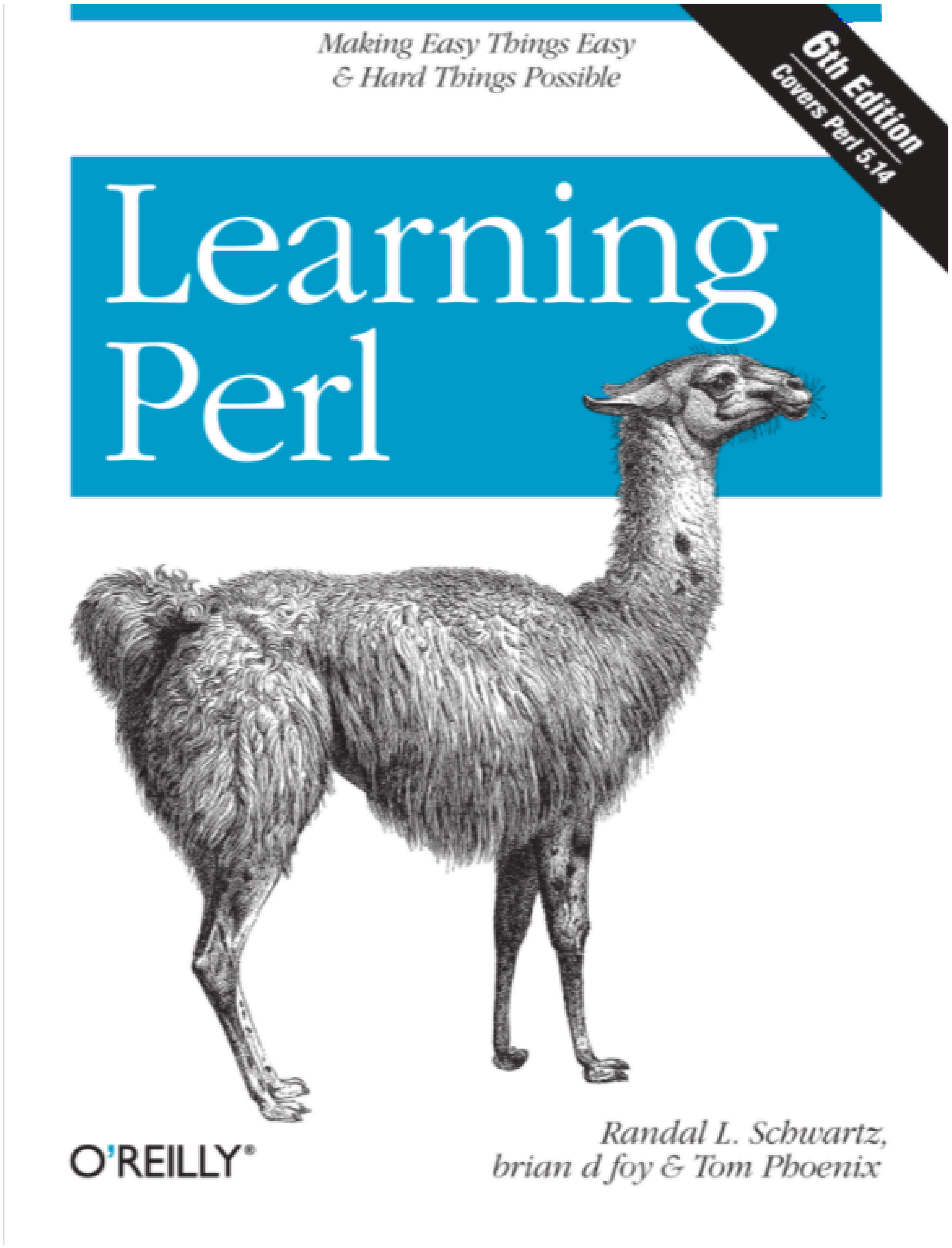
A Series of Notes Summarizing…



By Benjamin Cao

Table of Contents

This guide is in no way in partnership with O'Reilly Media or Randal L. Schwartz, brian d foy, and Tom Phoenix and should only used as supplement to "Learning Perl, 6th Edition" by Randal L. Schwartz, brian d foy, and Tom Phoenix (O'Reilly).

As a personal project to enrich my mind and increase my education, this document acts as an overall summary to the lessons learned in the listed textbook.

The distribution of these notes should only be allowed with permission from the owner, Benjamin Cao. If you wish to distribute these notes, contact Benjamin Cao at bencao@bcao.me.

Chapter 1 - Introduction

Questions and Answers

Is This the Right Book for You?

- This is not a reference book.
- A tutorial on the basics of Perl.

Why Are There So Many Footnotes?

- Perl has a lot of exceptions to its rules.

What About the Exercises and Their Answers?

- You need the chance to make mistakes.
- Answers to the exercises are in Appendix A.

What Do Those Numbers Mean at the Start of the Exercise?

- Rough estimate of how many minutes expected to spend on exercise.

What If I'm a Perl Course Instructor?

- Most exercises are short enough so students finish in 45 minutes to an hour.

What Does Perl Stand For?

Practical Extraction and Report Language

Why Did Larry Create Perl?

- Larry Wall is Perl's creator.
- Mid 1980's, tried to produce reports from a Usenet-news-like hierarchy of files for bug report system and `awk` ran out of steam.
- He wanted a general purpose tool to use in at least one other place.

Why Didn't Larry Just Use Some Other Language?

- Nothing met his needs.
- Perl tries to fill gap between low-level programming (C, C++, assembly) and high-level programming ("shell").
    - Low-level is hard to write and ugly, but fast and unlimited.
    - High-level is slow, hard, ugly, and limited.
    - Perl is easy, nearly unlimited, mostly fast, and kind of ugly.

Is Perl Easy or Hard?

- It is easy to use, but hard to learn.

How Did Perl Get to Be So Popular?

- Released to Usenet readers and features grew, along with portability.

What's Happening with Perl Now?

- Mostly maintained by group called Perl 5 Porters.

What's Perl Really Good For?

- Good for quick or long programs.
- Optimized for problems that are 90% working with text and 10% everything else.

What is Perl Not Good For?

- If you're trying to make an opaque binary.

How Can I Get Perl?

It comes pre-installed with most Linux, BSD systems, Mac OS X, etc.

What Is CPAN?

- Comprehensive Perl Archive Network
- One stop shopping for Perl.
- Comes with Perl source code, ports of Perl, documentation, etc.

How Can I Get Support for Perl?

- You get the complete source code for Perl… Bugs could potentially be fixed on your own.

Are There Any Other Kinds of Support?

- Perl Mongers (www.pm.org)
- Perl Documentation (www.cpan.org, www.perldoc.perl.org, www.faq.perl.org)
- The book "Programming Perl" (O'Rielly)
- Perl newsgroups on Usenet (located in comp.lang.perl)
- The Perl Monastery (www.perlmonks.org)
- Perl support on Stack Overflow.
- www.learn.perl.org
- Perlsphere (www.perlsphere.net)

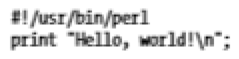
What If I Find a Bug in Perl?

- Check documentation.
- Ask around.
- Make test case.
  - Use perlbug utility (comes with Perl) to report bug.

How Do I Make a Perl Program?

Perl programs are text files. They can be created and edited using your favorite text editor.
Unix - emacs or vi
Mac OS X - BBEdit or TextMate
Windows - UltraEdit or PFE (Programmer's Favorite Editor)
If you use a word processor, make sure to save it as "text only".

A Simple Program

- 

```
#!/usr/bin/perl
print "Hello, world!\n";
```

- Certain systems may require you to do something so it knows it's an executable.
  - `chmod a=x my_program`
- Running a program.
  - `./my_program`
- say, instead of print, runs in Perl 5.10 or later.
- Put "use 5.010;" to indicate you're using new features of the specified version.

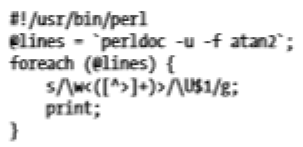
What's Inside That Program?

- Perl lets you use insignificant whitespace.
- Comments are denoted by "#". No block comments in Perl.
- #! denotes name of program executing the rest of the file.
- Most statements are an expression followed by a semi-colon.

How Do I Compile My Perl Program?

- `perl my_program`

A Whirlwind Tour of Perl



```
#!/usr/bin/perl
@lines = `perldoc -u -f atan2`;
foreach (@lines) {
    s/\w<([^>]+)>/\U$1/g;
    print;
}
```

Basically takes a command in an array and potentially makes changes to to it based on markers (< >) and then prints out the potentially modified line.

Chapter 2 - Scalar Data

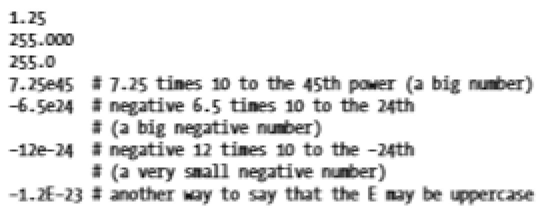
Numbers

scalar - simplest kind of data Perl manipulates
- most often a number or a string

All Numbers Have the Same Format Internally

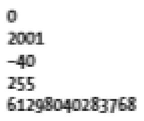
- Can declare integers and floating point.
- Perl computers with double-precision floating-point values.

Floating-Point Literals

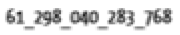
- literal - how you represent a value in your Perl source code
- Not the result of a calculation or an I/O operation; It's data you type directly into your program.
- 

```
1.25
255.000
255.0
7.25e45  # 7.25 times 10 to the 45th power (a big number)
-6.5e24  # negative 6.5 times 10 to the 24th
         # (a big negative number)
-12e-24  # negative 12 times 10 to the -24th
         # (a very small negative number)
-1.2E-23 # another way to say that the E may be uppercase
```

Integer Literals

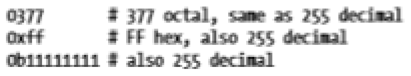
- 

```
0
2001
-40
255
61298040283768
```

- Perl lets you use underscores for clarity in integer literals.
  - 

```
61_298_040_283_768
```

Nondecimal Integer Literals

- Can use octal, hexadecimal, and binary literals as well.
- 

```
0377       # 377 octal, same as 255 decimal
0xff       # FF hex, also 255 decimal
0b11111111 # also 255 decimal
```

- Underscores can be used as well for clarity.

Numeric Operators

- Supports addition, subtraction, multiplication, and division.
- Modulus division (%) is supported.
- Also provides FORTRAN-like exponentiation operator (ex. **`2**3`**).

Strings

Perl fully supports Unicode, though Perl doesn't automatically interpret your source code as Unicode.
To use Unicode, you need to add utf8 pragma (**`use utf8;`**).

Single-Quoted String Literals

- To get backslash in single quote string, use '**`\\`**'.
- To get single quote in single quote string, use '**`\'`**'.

Double-Quoted String Literals

- Backslash has full power in specifying control characters (i.e. **`\n`**).

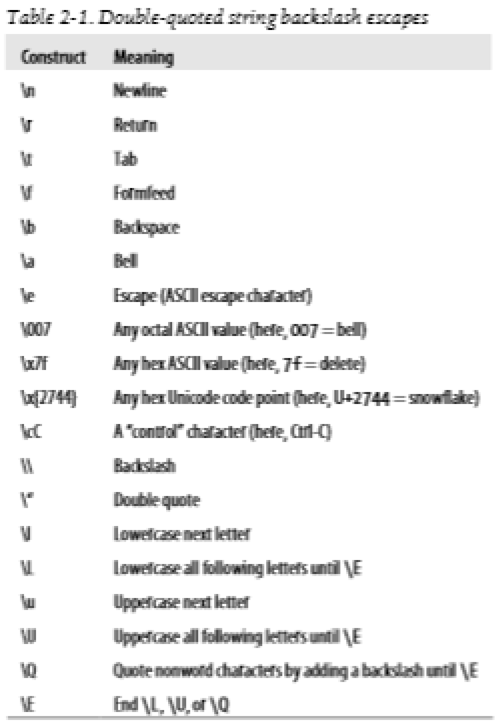


Table 2-1. Double-quoted string backslash escapes

Construct	Meaning
\n	Newline
\r	Return
\t	Tab
\f	Formfeed
\b	Backspace
\a	Bell
\e	Escape (ASCII escape character)
\007	Any octal ASCII value (here, 007 = bell)
\x7f	Any hex ASCII value (here, 7f = delete)
\x{2744}	Any hex Unicode code point (here, U+2744 = snowflake)
\cC	A "control" character (here, Ctrl-C)
\\	Backslash
\"	Double quote
\l	Lowercase next letter
\L	Lowercase all following letters until \E
\u	Uppercase next letter
\U	Uppercase all following letters until \E
\Q	Quote nonword characters by adding a backslash until \E
\E	End \L, \U, or \Q

- Variable interpolated - some variable names within string are replaced with their current values when strings are used.

String Operators

- Concatenate, or join strings using `.` operator.
- String repetition operator is lowercase **`x`**.
    - Not commutative. String is always the left operand.

    -
      ```
      "fred" x 3       # is "fredfredfred"
      "barney" x (4+1) # is "barney" x 5, or "barneybarneybarneybarneybarney"
      5 x 4.8          # is really "5" x 4, which is "5555"
      ```

Automatic Conversion Between Numbers and Strings

- Conversion depends on operator used.

Perl's Built-in Warnings

Use warnings pragma (**`use warnings;`**) for Perl version 5.6 or later.
`-w` option when compiling.

```
$ perl -w my_program
```

`-w` on shebang.

```
#!/usr/bin/perl -w
```

Works on non-Unix systems (**`#!perl -w`**)
Longer description of problem using diagnostics pragma.

```
#!/usr/bin/perl
use diagnostics;
```

`-M` to load pragma only when needed.

```
$ perl -Mdiagnostics ./my_program
```

Scalar Variables

variable - name for a container that holds one or more values
Begins with a dollar sign (sigil), followed by a letter or underscore.

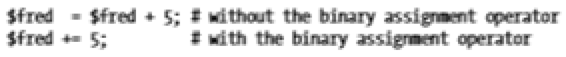
Choosing Good Variable Names

- Choose names that mean something.

Scalar Assignment

- Use ‘=’ to assign values to variables.

Binary Assignment Operators

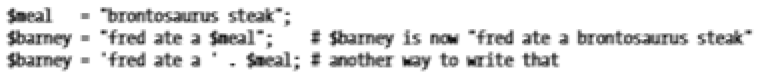
- Shorthand for expression where the same variable appears on both sides of an assignment.
  - 

```
$fred  = $fred + 5; # without the binary assignment operator
$fred += 5;         # with the binary assignment operator
```

Output with print

`print` takes scalar argument and puts it out.

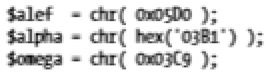
Interpolation of Scalar Variables into Strings

- When string literal is double-quoted, it is subject to variable interpolation.
  - Variable name is replaced with its value upon output.
  - 

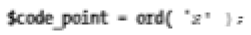
```
$meal   = "brontosaurus steak";
$barney = "fred ate a $meal";    # $barney is now "fred ate a brontosaurus steak"
$barney = 'fred ate a ' . $meal; # another way to write that
```

- Delimiter { } for variables for better recognition.

Creating Characters by Code Point

- For using characters not on the keyboard.
- Use code point with **`chr()`**.
  - 

```
$alef  = chr( 0x05D0 );
$alpha = chr( hex('03B1') );
$omega = chr( 0x03C9 );
```

- Reverse with **`ord()`**.
  - 

```
$code_point = ord( 'א' );
```

Operator Precedence and Associativity

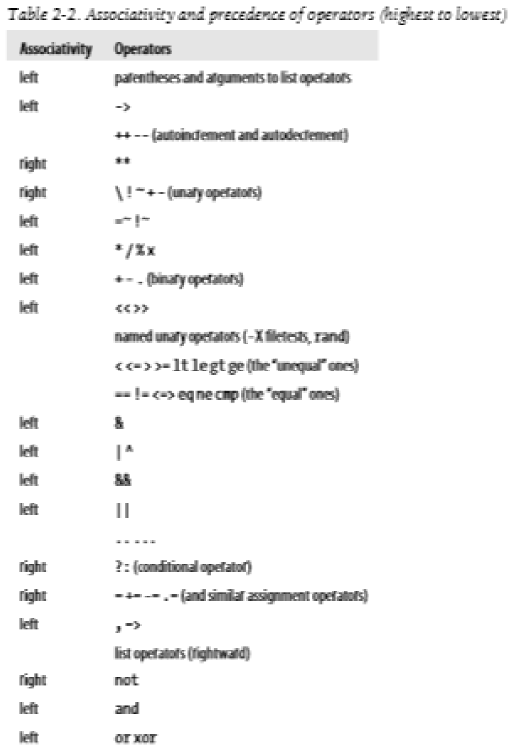
- 

Table 2-2. Associativity and precedence of operators (highest to lowest)

Associativity	Operators
left	parentheses and arguments to list operators
left	->
	++ -- (autoincrement and autodecrement)
right	**
right	\ ! ~ + - (unary operators)
left	=~ !~
left	* / % x
left	+ - . (binary operators)
left	<< >>
	named unary operators (-X filetests, rand)
	< <= > >= lt le gt ge (the "unequal" ones)
	== != <=> eq ne cmp (the "equal" ones)
left	&
left	| ^
left	&&
left	||
	.. ...
right	?: (conditional operator)
right	= += -= .= (and similar assignment operators)
left	, =>
	list operators (rightward)
right	not
left	and
left	or xor

Comparison Operators

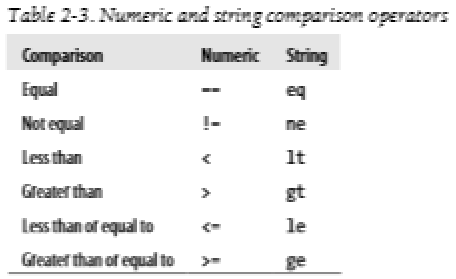
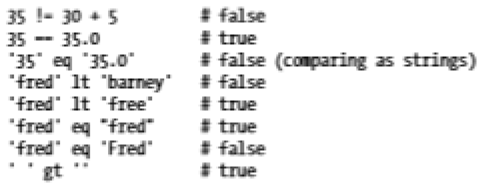
- 

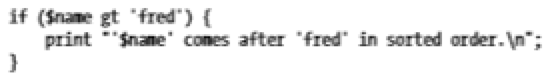
Table 2-3. Numeric and string comparison operators

Comparison	Numeric	String
Equal	==	eq
Not equal	!=	ne
Less than	<	lt
Greater than	>	gt
Less than or equal to	<=	le
Greater than or equal to	>=	ge

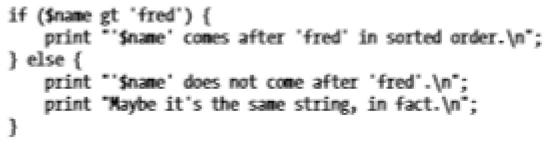
- 

```
35 != 30 + 5          # false
35 == 35.0            # true
'35' eq '35.0'        # false (comparing as strings)
'fred' lt 'barney'    # false
'fred' lt 'free'      # true
'fred' eq "fred"      # true
'fred' eq 'Fred'      # false
' ' gt ''             # true
```

The if Control Structure

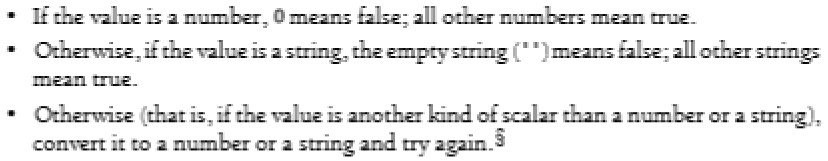


```
if ($name gt 'fred') {
    print "'$name' comes after 'fred' in sorted order.\n";
}
```



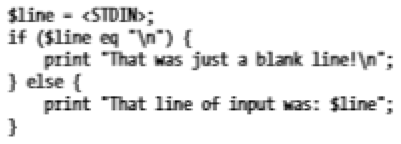
```
if ($name gt 'fred') {
    print "'$name' comes after 'fred' in sorted order.\n";
} else {
    print "'$name' does not come after 'fred'.\n";
    print "Maybe it's the same string, in fact.\n";
}
```

Boolean Values

- 
  - If the value is a number, 0 means false; all other numbers mean true.
  - Otherwise, if the value is a string, the empty string ('') means false; all other strings mean true.
  - Otherwise (that is, if the value is another kind of scalar than a number or a string), convert it to a number or a string and try again.§
- To get the opposite of a boolean value, use unary not (!) operator.

Getting User Input

Use **`<STDIN>`** operator.



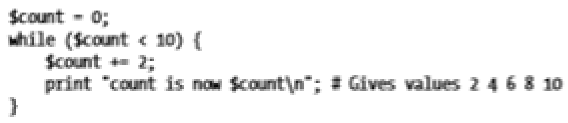
```
$line = <STDIN>;
if ($line eq "\n") {
    print "That was just a blank line!\n";
} else {
    print "That line of input was: $line";
}
```

The chomp Operator

Removes newline at the end of a variable if it exists.
`chomp()`'s return value is the number of newlines it removes (typically and most likely 1).
If a line ends with one or more newlines, **`chomp()`** removes only one.

The while Control Structure

Repeats a block of code as long as the condition is true.



```
$count = 0;
while ($count < 10) {
    $count += 2;
    print "count is now $count\n"; # Gives values 2 4 6 8 10
}
```

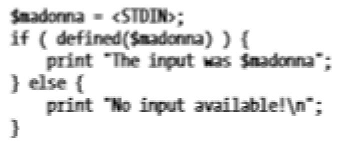
The undef Value

The value given to a scalar variable before giving it a value.

The defined Function

<STDIN> can return value undef.

Use defined function to tell if a value is undef. If undef, it returns false.



```
$madonna = <STDIN>;
if ( defined($madonna) ) {
    print "The input was $madonna";
} else {
    print "No input available!\n";
}
```

Chapter 3 - Lists and Arrays

list - ordered collection of scalars
array - variable that contains a list
List is the data, array is a variable that stores the data.
Each element in an array or list is a separate scalar value.
First element is always indexed as element zero.
Each element can hold a number, string, undef, or a mixture of scalars.

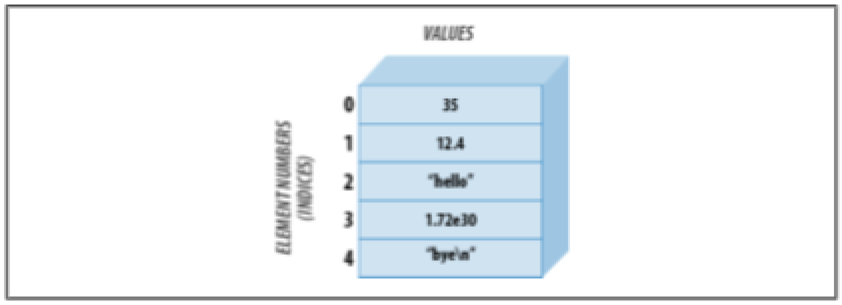


Figure 3-1. A list with five elements

Accessing Elements of an Array

```
$fred[0] = "yabba";
$fred[1] = "dabba";
$fred[2] = "doo";

$number = 2.71828;
print $fred[$number - 1]; # Same as printing $fred[1]
```

Number is truncated to the next lowest integer when used as index.

```
$blank = $fred[ 142_857 ]; # unused array element gives undef
$blanc = $mel;             # unused scalar $mel also gives undef
```

If the subscript indicates element is beyond end of array, the value is undef.

Special Array Indices

```
$rocks[0]  = 'bedrock';      # One element...
$rocks[1]  = 'slate';        # another...
$rocks[2]  = 'lava';         # and another...
$rocks[3]  = 'crushed rock'; # and another...
$rocks[99] = 'schist';       # now there are 95 undef elements
```

$#name gives last element index in array

```
$end = $#rocks;                  # 99, which is the last element's index
$number_of_rocks = $end + 1;     # okay, but you'll see a better way later
$rocks[ $#rocks ] = 'hard rock'; # the last rock
```

Can use negative indices. Works from last element to first.

```
$rocks[ -1 ]   = 'hard rock';   # easier way to do that last example
$dead_rock     = $rocks[-100]; # gets 'bedrock'
$rocks[ -200 ] = 'crystal';     # fatal error!
```

List Literals

This is the way you represent a list value in your program.
List of comma-separated values enclosed in parentheses.

```
(1, 2, 3)      # list of three values 1, 2, and 3
(1, 2, 3,)     # the same three values (the trailing comma is ignored)
("fred", 4.5)  # two values, "fred" and 4.5
( )            # empty list - zero elements
(1..100)       # list of 100 integers
```

Last example above uses the range operator.

The qw Shortcut

- Allows for list creation w/o commas or quotes
- AKA “quoted words” or “quoted by whitespace”

-
  ```
  qw( fred barney betty wilma dino ) # same as above, but less typing
  ```
- Can use any punctuation as delimiter.

  -
    ```
    qw! fred barney betty wilma dino !
    qw/ fred barney betty wilma dino /
    qw# fred barney betty wilma dino #   # like in a comment!
    ```
  - If you need to use the enclosing delimiter in the string, preface it with a backslash.

  -
    ```
    qw! yahoo\! google ask msn ! # include yahoo! as an element
    ```

List Assignment

`@` refers to the entire array.

```
@rocks
```

Arrays can’t contain other arrays.

The pop and push Operators

- pop takes the last element off the array and returns it

  -
    ```
    @array  = 5..9;
    $fred   = pop(@array);  # $fred gets 9, @array now has (5, 6, 7, 8)
    $barney = pop @array;   # $barney gets 8, @array now has (5, 6, 7)
    pop @array;             # @array now has (5, 6). (The 7 is discarded.)
    ```
- returns undef if array is empty
- push adds element to end of array

  -
    ```
    push(@array, 0);      # @array now has (5, 6, 0)
    push @array, 8;       # @array now has (5, 6, 0, 8)
    push @array, 1..10;   # @array now has those 10 new elements
    @others = qw/ 9 0 2 1 0 /;
    push @array, @others; # @array now has those five new elements (19 total)
    ```

The shift and unshift Operators

- Perform similar actions to push and pop on the “start” of an array.

-
  ```
  @array = qw# dino fred barney #;
  $m = shift(@array);      # $m gets "dino", @array now has ("fred", "barney")
  $n = shift @array;       # $n gets "fred", @array now has ("barney")
  shift @array;            # @array is now empty
  $o = shift @array;       # $o gets undef, @array is still empty
  unshift(@array, 5);      # @array now has the one-element list (5)
  unshift @array, 4;       # @array now has (4, 5)
  @others = 1..3;
  unshift @array, @others; # @array now has (1, 2, 3, 4, 5)
  ```

The splice Operator

- Splice works in the middle of the array.

-
  ```
  @array = qw( pebbles dino fred barney betty );
  @removed = splice @array, 2; # remove everything after fred
                               # @removed is qw(fred barney betty)
                               # @array is qw(pebbles dino)
  ```
- Takes 4 arguments, 2 are optional.
  - 1. Array 2. Position to start 3. Length 4. Replacement List

  -
    ```
    @array = qw( pebbles dino fred barney betty );
    @removed = splice @array, 1, 2, qw(wilma); # remove dino, fred
                                               # @removed is qw(dino fred)
                                               # @array is qw(pebbles wilma
                                               #                  barney betty)
    ```

Interpolating Arrays into Strings

Can interpolate array values into a double-quoted string.

```
@rocks = qw{ flintstone slate rubble };
print "quartz @rocks limestone\n";  # prints five rocks separated by spaces
```

Perl expands array and automatically adds spaces between elements.
No extra spaces before or after an interpolated array.
@ like for email, Perl will try to interpolate it.

```
$email = "fred@bedrock.edu";  # WRONG! Tries to interpolate @bedrock
```

use **\@** in double-quotes or use **@** in single quotes.

```
$email = "fred\@bedrock.edu"; # Correct
$email = 'fred@bedrock.edu';  # Another way to do that
```

The foreach Control Structure

Loops through a list of values, executing one iteration for each value.
Value of control variable is the same as before the loop started.
Perl automatically saves and restores the value of the control value of a foreach loop.

```
foreach $rock (qw/ bedrock slate lava /) {
    print "One rock is $rock.\n";  # Prints names of three rocks
}
```

Perl's Favorite Default: $_

- Default control variable of foreach loop.

```
foreach (1..10) {  # Uses $_ by default
    print "I can count to $_!\n";
}
```

The reverse Operator

- Takes a list of values and returns the list in the opposite order.

```
@fred   = 6..10;
@barney = reverse(@fred);  # gets 10, 9, 8, 7, 6
@wilma  = reverse 6..10;   # gets the same thing, without the other array
@fred   = reverse @fred;   # puts the result back into the original array
```

The sort Operator

- Takes a list of values and sorts them in the internal character ordering.
- For strings, it's sorted by code point order.

```
@rocks   = qw/ bedrock slate rubble granite /;
@sorted  = sort(@rocks);        # gets bedrock, granite, rubble, slate
@back    = reverse sort @rocks; # these go from slate to bedrock
@rocks   = sort @rocks;         # puts sorted result back into @rocks
@numbers = sort 97..102;        # gets 100, 101, 102, 97, 98, 99
```

The each Operator

- Useable starting with Perl 5.12 and is used on arrays.
- Using each on an array returns two values for the next element in the array - the index and the value.

```
@rocks   = qw/ bedrock slate rubble granite /;
while( my( $index, $value ) = each @rocks ) {
    say "$index: $value";
}
```

Scalar and List Context

The context of how you use an expression is important.

```
42 + something # The something must be a scalar
sort something # The something must be a list
```

```
@people = qw( fred barney betty );
@sorted = sort @people;  # list context: barney, betty, fred
$number = 42 + @people;  # scalar context: 42 + 3 gives 45

@list = @people; # a list of three people
$n = @people;    # the number 3
```

Using List-Producing Expressions in Scalar Context

- Some expressions don't have a scalar-context value.
- sort always returns undef in scalar context.
- reverse returns reversed string in scalar context

    -
      ```
      @backwards = reverse qw/ yabba dabba doo /;
          # gives doo, dabba, yabba
      $backwards = reverse qw/ yabba dabba doo /;
          # gives oodabbadabbay
      ```

-
  ```
  $fred = something;            # scalar context
  @pebbles = something;         # list context
  ($wilma, $betty) = something; # list context
  ($dino) = something;          # still list context!
  ```

Using Scalar-Producing Expressions in List Context

- If an expression doesn't normally have a list value, the scalar values if automatically promoted to make a one-element list.

-
  ```
  @fred = 6 * 7; # gets the one-element list (42)
  @barney = "hello" . ' ' . "world";
  ```
- A catch is using undef since it's scalar value, assigning it to an array doesn't clear the array

    -
      ```
      @wilma = undef; # OOPS! Gets the one-element list (undef)
          # which is not the same as this:
      @betty = ( );    # A correct way to empty an array
      ```

Forcing Scalar Context

- Use "fake" function scalar
    - Just tells Perl to provide scalar context.

    -
      ```
      @rocks = qw( talc quartz jade obsidian );
      print "How many rocks do you have?\n";
      print "I have ", @rocks, " rocks!\n";        # WRONG, prints names of rocks
      print "I have ", scalar @rocks, " rocks!\n"; # Correct, gives a number
      ```

<STDIN> in List Context

<STDIN> returns the next line of input in scalar context.
In list context, it returns all of the remaining lines up to end of file.
How to get end of file from keyboard?
Linux & Mac OS X - Control-D
DOS/Windows - Ctrl-Z
Use **chomp** to remove newlines from each item in the list.

```
chomp(@lines = <STDIN>); # Read the lines, not the newlines
```

Chapter 4 - Subroutines

subroutine - user defined function
- allow you to recycle a chunk of code many times in one program
- name (anything except digit) with ampersand (&) in front when calling it

Defining a Subroutine

Use keyword “sub”, then name of subroutine, with the block of code in curly braces.

```
sub marine {
    $n += 1;  # Global variable $n
    print "Hello, sailor number $n!\n";
}
```

Invoking a Subroutine

```
&marine;  # says Hello, sailor number 1!
&marine;  # says Hello, sailor number 2!
&marine;  # says Hello, sailor number 3!
&marine;  # says Hello, sailor number 4!
```

Return Values

All subroutines have a return value.

```
sub sum_of_fred_and_barney {
    print "Hey, you called the sum_of_fred_and_barney subroutine!\n";
    $fred + $barney;  # That's the return value
}
```

Return value of example above is the sum of **`$fred`** and **`$barney`**.

Arguments

To pass arguments, place them in a list expression in parentheses, after subroutine invocation.

```
$n = &max(10, 15);  # This sub call has two parameters
```

Perl stores parameter list in array variable **`@_`**, which is private to the subroutine.

Private Variables in Subroutines

All Perl variables are global by default.
To create private (lexical) variables, use **`my`** operator.

```
sub max {
    my($m, $n);       # new, private variables for this block
    ($m, $n) = @_;    # give names to the parameters
    if ($m > $n) { $m } else { $n }
}
```

Variables in this case are scoped to enclosing block.

Variable-Length Parameter Lists

Use **`@_`** array to check if subroutine has right number of arguments.

```
sub max {
    if (@_ != 2) {
        print "WARNING! &max should get exactly two arguments!\n";
    }
    # continue as before...
    .
    .
    .
}
```

A Better &max Routine

```
$maximum = &max(3, 5, 10, 4, 6);

sub max {
    my($max_so_far) = shift @_;  # the first one is the largest yet seen
    foreach (@_) {               # look at the remaining arguments
        if ($_ > $max_so_far) {  # could this one be bigger yet?
            $max_so_far = $_;
        }
    }
    $max_so_far;
}
```

-
- Uses “high-water mark” algorithm, which keeps track of the largest number seen.

Empty Parameter Lists

- If you pass empty parameters, the subroutine returns undef.

Notes on Lexical (my) Variables

Variable is private to the enclosing block.

```
foreach (1..10) {
    my($square) = $_ * $_;  # private variable in this loop
    print "$_ squared is $square.\n";
}
```

Note also that the my operator doesn’t change the context of an assignment:

```
    my($num) = @_;  # list context, same as ($num) = @_;
    my $num  = @_;  # scalar context, same as $num = @_;
```

The use strict Pragma

Enforces good programming rules.

The return Operator

Way to stop subroutine.

```
my @names = qw/ fred barney betty dino wilma pebbles bamm-bamm /;
my $result = &which_element_is("dino", @names);

sub which_element_is {
    my($what, @array) = @_;
    foreach (0..$#array) {  # indices of @array's elements
        if ($what eq $array[$_]) {
            return $_;         # return early once found
        }
    }
    -1;                   # element not found (return is optional here)
}
```

Omitting the Ampersand

- You can omit the ampersand if the compiler sees the subroutine definition before the invocation.
- Or if Perl can tell from syntax that it’s a subroutine call.
- The catch: if you use the same name as a built-in function

Non-Scalar Return Values

If you call your subroutine in list context, it can return a list.

```
sub list_from_fred_to_barney {
    if ($fred < $barney) {
        # Count upwards from $fred to $barney
        $fred..$barney;
    } else {
        # Count downwards from $fred to $barney
        reverse $barney..$fred;
    }
}

$fred = 11;
$barney = 6;
@c = &list_from_fred_to_barney; # @c gets (11, 10, 9, 8, 7, 6)
```

Persistent, Private Variables

state - private variables scoped to subroutine, but Perl keeps values between calls.

```
use 5.010;

running_sum( 5, 6 );
running_sum( 1..3 );
running_sum( 4 );

sub running_sum {
  state $sum = 0;
  state @numbers;

  foreach my $number ( @_ ) {
    push @numbers, $number;
    $sum += $number;
  }

  say "The sum of (@numbers) is $sum";
  }
```

Can’t create state variables in list context.

```
state @array = qw(a b c); # Error!
```

Chapter 5 - Input and Output

Input from Standard Input

<STDIN> operator

```
while (<STDIN>) {
    print "I saw $_";
}
```

Shortcut only works if there's **ONLY** the line-input operator in the conditional.

Input from the Diamond Operator

<>

Useful for making programs that work like Unix utilities.
Instead of getting input from the keyboard, it's from the user's choice of input.

```
while (defined($line = <>)) {
    chomp($line);
    print "It was $line that I saw!\n";
}
```

Can use shortcut like before, to use **$_**.

```
while (<>) {
    chomp;
    print "It was $_ that I saw!\n";
}
```

The Invocation Arguments

@ARGV array holds invocation arguments.
Diamond operator looks in **@ARGV** to determine what filenames to use.

```
@ARGV = qw# larry moe curly #;  # force these three files to be read
while (<>) {
    chomp;
    print "It was $_ that I saw in some stooge-like file!\n";
}
```

Output to Standard Output

print operator takes list of values and sends each item to standard output as a string.
Has optional parentheses.

```
$name = "Larry Wall";
print "Hello there, $name, did you know that 3+4 is ", 3+4, "?\n";
```

Formatted Output with `printf`

Takes a format string followed by a list of things to print.
conversions - begins with **%** followed by a letter.

```
printf "Hello, %s; your password expires in %d days!\n",
    $user, $days_to_die;
```

%g automatically choses floating-point, integers, or even exponential notation

```
printf "%g %g %g\n", 5/2, 51/17, 51 ** 17;  # 2.5 3 1.0683e+29
```

%d is decimal integer

%#d adds number of spaces (insert to #) before printing integer.

```
printf "%6d\n", 42;  # output like ````42 (the ` symbol stands for a space)
printf "%2d\n", 2e3 + 1.95;  # 2001
```

%s is string

%#s sets field width (justification of string)

```
printf "%10s\n", "wilma";  # looks like `````wilma
```

`%f` is floating point, lets you determine number of digits after decimal point.

```
printf "%12f\n", 6 * 7 + 2/3;    # looks like ```42.666667
printf "%12.3f\n", 6 * 7 + 2/3;  # looks like ``````42.667
printf "%12.0f\n", 6 * 7 + 2/3;  # looks like ``````````43
```

Arrays and `printf`

- Format strings work only with fixed number of items, so it's impractical to use array.
- Is possible using a format string.

    -
      ```
      my @items = qw( wilma dino pebbles );
      my $format = "The items are:\n" . ("%10s\n" x @items);
      ## print "the format is >>$format<<\n"; # for debugging
      printf $format, @items;
      ```

Filehandles

Name in a Perl program for an I/O connection between your Perl process and the outside world.
Filename cannot start with a digit.
Recommendation to name filehandles with uppercase letters.

Opening a Filehandle

`open` operator tells Perl to ask operating system to open connection between your program and the outside world.

```
open CONFIG, 'dino';
open CONFIG, '<dino';
open BEDROCK, '>fred';
open LOG, '>>logfile';
```

Can use scalar expression in place of filename specifier.

```
my $selected_output = 'my_output';
open LOG, "> $selected_output";
```

3 argument open is available starting with Perl 5.6.

```
open CONFIG, '<', 'dino';
open BEDROCK, '>', $file_name;
open LOG, '>>', &logfile_name();
```

- Perl never confuses 2nd and 3rd arguments.
- Can specify encoding
- `open CONFIG, '<:encoding(UTF-8)', 'dino';`
- `open BEDROCK, '>:utf8', $file_name;  # probably not right` (shortcut for specifying encoding)

Specifying encoding to deal with DOS line endings.

```
open BEDROCK, '>:crlf', $file_name;
```

Binmoding Filehandles

- **`binmode`** turns off line ending processing

    -
      ```
      binmode STDOUT; # don't translate line endings
      binmode STDERR; # don't translate line endings
      ```
- Can specify layer as second argument.
    - `binmode STDOUT, ':encoding(UTF-8)';`

Bad Filehandles

- Trying to read from a bad filehandle (not properly opened or closed network connection) will end with end of file

Closing a Filehandle

- Use the **`close`** operator to close a filehandle.
    - `close BEDROCK;`

Fatal Errors with `die`

`die` function prints out message you give it to standard error stream and makes sure your program exits with a nonzero exit status.

```
if ( ! open LOG, '>>', 'logfile' ) {
    die "Cannot create logfile: $!";
}
```

`$!` - human readable complaint from system
`die` will automatically append the Perl program name and line numbers to the end of the message to easily identify where in the program it exited.
Add newline at the end of to leave off line number and file on usage errors

```
if (@ARGV < 2) {
    die "Not enough arguments\n";
}
```

Warning Messages with `warn`

- **`warn`** does the same as **`die`**, but doesn't exit program.

Automatically die-ing

- Starting with Perl 5.10, the autodie pragma is available so you don't have to use **`die`** every time
```
use autodie;
```
-
```
open LOG, '>>', 'logfile';
```

Using Filehandles

Can read lines from open filehandle just like STDIN.

```
if ( ! open PASSWD, "/etc/passwd") {
    die "How did you get logged in? ($!)";
}

while (<PASSWD>) {
    chomp;
    ...
}
```

You can use a filehandle open for writing or appending with **`print`** or **`printf`**, appearing immediately after the keyword but before the list of arguments.

```
print LOG "Captain's log, stardate 3.14159\n";  # output goes to LOG
printf STDERR "%d percent complete.\n", $done/$total * 100;
```

Changing the Default Output Filehandle

- By default, if no filehandle is given to **`print`** or **`printf`**, the output goes to STDOUT.
- Default is changed with **`select`** operator.
- **`$| = 1`** flushes the buffer so entries won't get stuck.
  -
```
select LOG;
$| = 1;  # don't keep LOG entries sitting in the buffer
select STDOUT;
# ... time passes, babies learn to walk, tectonic plates shift, and then...
print LOG "This gets written to the LOG at once!\n";
```

Reopening a Standard Filehandle

```
# Send errors to my private error log
if ( ! open STDERR, ">>/home/barney/.error_log") {
    die "Can't open error log for append: $!";
}
```

Reopening STDERR, error message from Perl go to new file.

If **`die`** is executed in statement above, the original standard filehandle (in this case, STDERR) picks up the error message.

Output with `say`

`say` is like print except it adds newline to the end.

```
use 5.010;

print "Hello!\n";
print "Hello!", "\n";
say "Hello!";
```

Filehandles in a Scalar

Using scalar variable without a value, your filehandle ends up in the variable.

```
my $rocks_fh;
open $rocks_fh, '<', 'rocks.txt'
    or die "Could not open rocks.txt: $!";
```

Surround anything that should be a filehandle in braces so Perl does the right thing.

```
print { $rock_fh }; # uses $_ by default
print { $rocks[0] } "sandstone\n";
```

Chapter 6 - Hashes

What is a Hash?

A hash is a data structure.
You look up hash values by name.
The indices are known as keys and are arbitrary, unique strings.
Keys are always converted to strings.

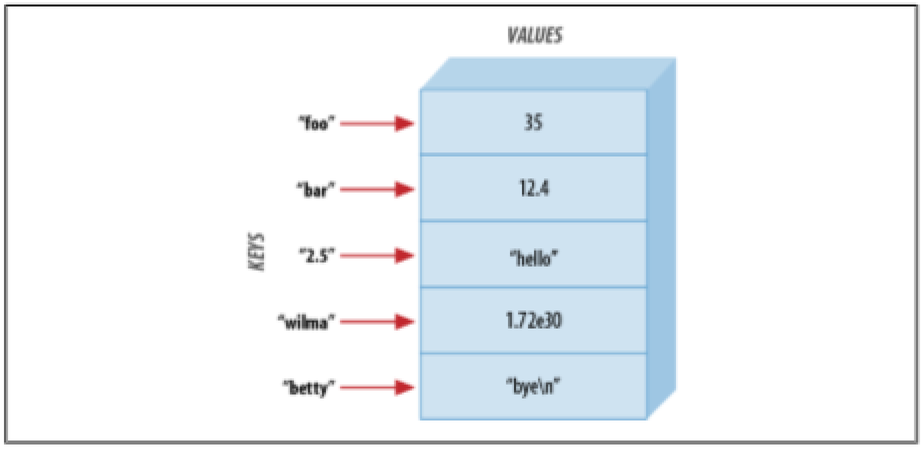


Figure 6-1. Hash keys and values

Why Use a Hash?

- For a set of data related to another set of data.
- Ex. Given name, family name
- Ex. Hostname, IP Address

Hash Element Access

```
$hash{$some_key}
```

The Hash As a Whole

- **%** as a prefix.
- Can turn a hash into a list.

```
%some_hash = ('foo', 35, 'bar', 12.4, 2.5, 'hello',
      'wilma', 1.72e30, 'betty', "bye\n");
```

The value of the hash (in a list context) is a simple list of key-value pairs:

```
@any_array = %some_hash;
```

Perl calls this *unwinding* the hash; turning it back into a list of key-value pairs. Of course, the pairs won't necessarily be in the same order as the original list:

```
print "@any_array\n";
  # might give something like this:
  # betty bye (and a newline) wilma 1.72e+30 foo 35 2.5 hello bar 12.4
```

-

Hash Assignment

- Assign one hash to another OR make inverse hash.

-
  ```
  my %new_hash = %old_hash;
  ```
  - Inverse hash good only if no duplicate values.

  -
    ```
    my %inverse_hash = reverse %any_hash;
    ```

The Big Arrow

- **=>** Easy way to identify key-value pairs when making a hash.

```
my %last_name = (  # a hash may be a lexical variable
  'fred'   => 'flintstone',
  'dino'   => undef,
  'barney' => 'rubble',
  'betty'  => 'rubble',
);
```

Hash Functions

The `keys` and `values` Functions

- keys - yields a list of all keys in a hash
- values - gives the corresponding values
- Empty list is returned if there are no elements in hash.

```
my %hash = ('a' => 1, 'b' => 2, 'c' => 3);
my @k = keys %hash;
my @v = values %hash;
```

- In scalar context, it will return the number of keys in the hash.

```
my $count = keys %hash;  # gets 3, meaning three key-value pairs
```

The `each` Function

- Used for iterating over an entire hash.
- Returns key-value pair as two-element list.

```
while ( ($key, $value) = each %hash ) {
  print "$key => $value\n";
}
```

Typical Use of a Hash

Ex. A library system that uses a hash to determine the number of books a person has checked out.

The `exists` Function

- Used to see if a key exists in a hash.

```
if (exists $books{"dino"}) {
  print "Hey, there's a library card for dino!\n";
}
```

The `delete` Function

- Removes given key and values from hash.
- NOT the same as storing undef into hash element.

```
my $person = "betty";
delete $books{$person};  # Revoke the library card for $person
```

Hash Element Interpolation

You can interpolate a single hash element into a double-quoted string just as you'd expect:

```
foreach $person (sort keys %books) {            # each patron, in order
  if ($books{$person}) {
    print "$person has $books{$person} items\n";   # fred has 3 items
  }
}
```

The `%ENV` hash

Perl stores information about your environment in the **`%ENV`** hash.

Ex. Getting your PATH from **`%ENV`** - `print "PATH is $ENV{PATH}\n";`

Chapter 7 - In the World of Regular Expressions

What Are Regular Expressions?

regular expression (pattern in Perl) - a template that either matches or doesn't match a given string

Using Simple Patterns

To match pattern against contents of $_, put pattern between forward slashes.

```
$_ = "yabba dabba doo";
if (/abba/) {
    print "It matched!\n";
}
```

Unicode Properties

- **\p{PROPERTY}** to match particular unicode property.
  ```
  if (/\p{Space}/) { # 26 different possible characters
      print "The string has some whitespace.\n";
  }
  ```
-
- Also **\p{Digit}** and **\p{Hex}** available.
- To negative property, use **\P{PROPERTY}**.

About Metacharacters

- Metacharacters have special meanings in regular expressions.
- Dot (**.**) is wildcard character. It matches any single character except a newline.

Simple Quantifiers

- ***** - to match preceding item zero or more times
  - `/fred\t*barney/`
- **+** - to match preceding item one or more times
  - `/fred +barney/`
- **?** - preceding item is optional
  - `/bamm-?bamm/`

Grouping in Patterns

- Parentheses to group parts of a pattern.
- Parentheses give a way to reuser part of the string directly in the match.
  - Use back references to refer to text that you matched in parentheses, called capture group.
  - Denote back reference as backslash followed by a number.
  - Number denotes capture group.
    ```
    $_ = "abba";
    if (/(.)\1/) {  # matches 'bb'
        print "It matched same character next to itself!\n";
    }
    ```
  - In Perl 5.10 a new way to denote back references is **\g{N}**.
    - N is the number of the back reference.
    - Can use negative numbers (relative back reference.

Alternatives

- **|** - match either left or right side
  - `/fred|barney|betty/`

Character Classes

character class - list of possible characters in square brackets (**[]**) matches any single character from within the class.

```
[abcwxyz]
```

^ negates character class

```
[^def]
```

Character Class Shortcuts

- Abbreviate character class for digits with **\d**.
- **/a** at the end of the match operator tells Perl to use old ASCII interpretation (feature in Perl 5.14).
- **\s** for matching any whitespace (similar to **\p{Space}**).
- **\h** for horizontal whitespace.

    -
      ```
      if (/\h/) {
          say 'The string matched some horizontal whitespace.';
      }
      ```
- **\v** for vertical whitespace.

    -
      ```
      if (/\v/) {
          say 'The string matched some vertical whitespace.';
      }
      ```
- **\R** matches linebreak.
- **\w** matches set of characters **[a-zA-z0-9_]**.

Negating the Shortcuts

- You can use **^** or just uppercase the shortcuts (**\D \W \S**).

Chapter 8 - Matching with Regular Expressions

Matches with `m//`

`/fred/` is shortcut for **`m//`** operator.
Like **`qw//`**, can look like **`m(fred)`**, **`m<fred>`**, with different paired delimiters.
Shortcut is only valid with the forward slash delimiter.

Match Modifiers

Can append as a group right after ending delimiter to change behavior from the default.

Case-Insensitive Matching with `/i`

-
  ```
  print "Would you like to play a game? ";
  chomp($_ = <STDIN>);
  if (/yes/i) {  # case-insensitive match
      print "In that case, I recommend that you go bowling.\n";
  }
  ```

Matching Any Character with `/s`

- Used for matching strings that have newlines in them.
    - Applies to **`.`** in the pattern.

    -
      ```
      $_ = "I saw Barney\ndown at the bowling alley\nwith Fred\nlast night.\n";
      if (/Barney.*Fred/s) {
          print "That string mentions Fred after Barney!\n";
      }
      ```
- **`\N`** shortcut complements **`\n`** for matching newline.

Adding Whitespace with `/x`

- Allows arbitrary whitespace to pattern for readability.

-
  ```
  /-?[0-9]+\.?[0-9]*/         # what is this doing?
  / -? [0-9]+ \.? [0-9]* /x   # a little better
  ```

Combining Option Modifiers

- Put modifiers at the end. Order is not important.

-
  ```
  if (/barney.*fred/is) {  # both /i and /s
      print "That string mentions Fred after Barney!\n";
  }
  ```

Choosing a Character Interpretation

- **`/a`** - ASCII
- **`/u`** - Unicode
- **`/l`** - locale

-
  ```
  use 5.014;

  /\w+/a       # A-Z, a-z, 0-9, _
  /\w+/u       # any Unicode word charcter
  /\w+/l       # The ASCII version, and word chars from the locale,
               # perhaps characters like Œ from Latin-9
  ```

Anchors

`\A` anchor matches at the absolute beginning of a string, so pattern doesn't float down string.

```
m{\Ahttps?://}i
```

Use **`\z`** to anchor at the end of a string.

```
m{\.png\z}i
```

`\Z` allows option newline after it.

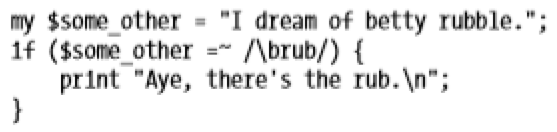
```
while (<STDIN>) {
    print if /\.png\Z/;
}
```

Word Anchors

- Use **`\b`** to match at either end of a word.
- **`/\bfred\b/`** matches fred ONLY.
- **`\B`** matches where **`\b`** would not match.
- **`/\bsearch\B/`** matches “searches”, “searching”, but NOT “search” or “researching”.

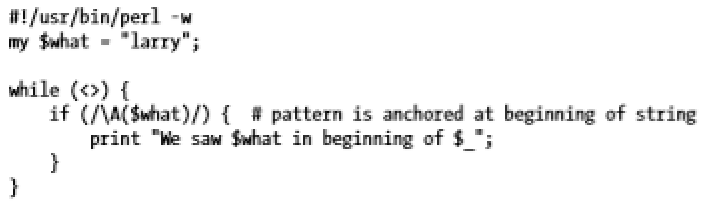
The Binding Operator =~

Tells Perl to match pattern on the right to string on left instead of matching to **`$_`**.



```
my $some_other = "I dream of betty rubble.";
if ($some_other =~ /\brub/) {
    print "Aye, there's the rub.\n";
}
```

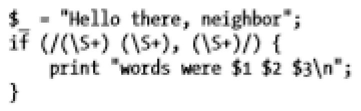
Interpolating into Patterns



```
#!/usr/bin/perl -w
my $what = "larry";

while (<>) {
    if (/\A($what)/) {  # pattern is anchored at beginning of string
        print "We saw $what in beginning of $_";
    }
}
```

The Match Variables

Made by using parentheses to make capture groups.
Match variables become **`$1`**, **`$2`**, … based on order of capture groups.



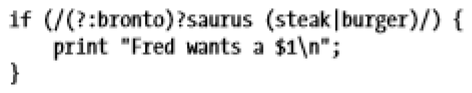
```
$_ = "Hello there, neighbor";
if (/(\S+) (\S+), (\S+)/) {
    print "words were $1 $2 $3\n";
}
```

The Persistence of Captures

- Capture variables stay until next successful pattern match.
- This is the reason why pattern match is mostly found in conditionals

Noncapturing Parentheses

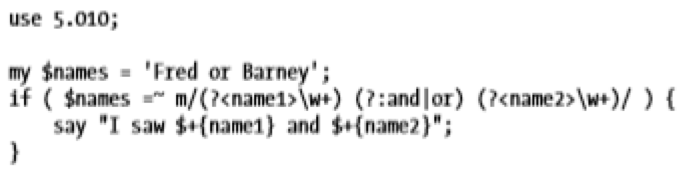
- Add **`?:`** to parentheses to tell Perl you don’t want to use a capture group.



```
if (/(?:bronto)?saurus (steak|burger)/) {
    print "Fred wants a $1\n";
}
```

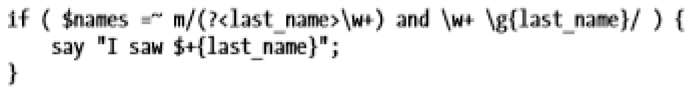
Named Captures

- 5.10+ lets you name match variable (**`?<LABEL>PATTERN`**).



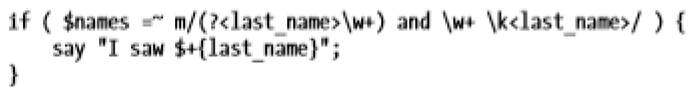
```
use 5.010;

my $names = 'Fred or Barney';
if ( $names =~ m/(?<name1>\w+) (?:and|or) (?<name2>\w+)/ ) {
    say "I saw $+{name1} and $+{name2}";
}
```

- Use **`\g{label}`** to refer to them for back reference.



```
if ( $names =~ m/(?<last_name>\w+) and \w+ \g{last_name}/ ) {
    say "I saw $+{last_name}";
}
```

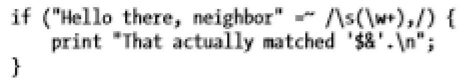
- **`\k<label>`** has same effect.



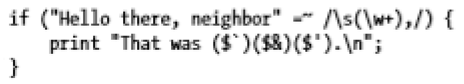
```
if ( $names =~ m/(?<last_name>\w+) and \w+ \k<last_name>/ ) {
    say "I saw $+{last_name}";
}
```

The Automatic Match Variables

- **$&** - stores part of string actually matched
- **$`** - stores before matched section
- **$’** - stores after matched section

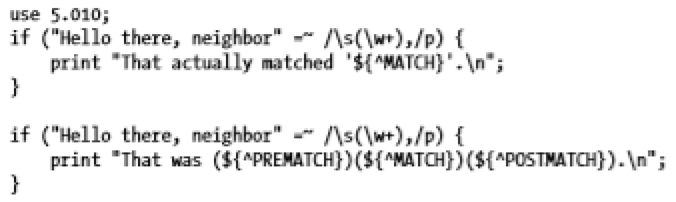


```
if ("Hello there, neighbor" =~ /\s(\w+),/) {
    print "That actually matched '$&'.\n";
}
```



```
if ("Hello there, neighbor" =~ /\s(\w+),/) {
    print "That was ($`)($&)($').\n";
}
```

- 5.10+ allows use of /p modifier to grant **${^PREMATCH}**, **${^MATCH}**, and **${^POSTMATCH}** with same effect as above.



```
use 5.010;
if ("Hello there, neighbor" =~ /\s(\w+),/p) {
    print "That actually matched '${^MATCH}'.\n";
}

if ("Hello there, neighbor" =~ /\s(\w+),/p) {
    print "That was (${^PREMATCH})(${^MATCH})(${^POSTMATCH}).\n";
}
```

General Quantifiers

quantifier - to repeat the preceding item a certain number of times in a pattern.
Ex. *****, **+**. **?**
Use comma separated numbers in curly braces to specify number of repetitions.
Ex. **/a{5,15}** - matches repetition of a from 5 to 15 times.
/a{5,} - matches repetition of a 5 or more times.
/a{,15} - matches repetition of a up to 15 times.

Precedence

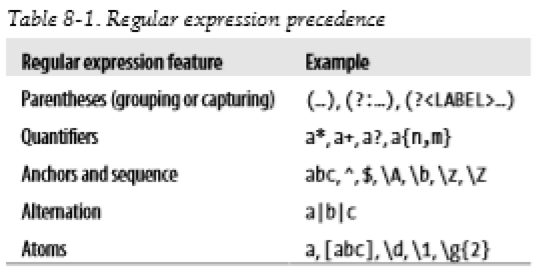
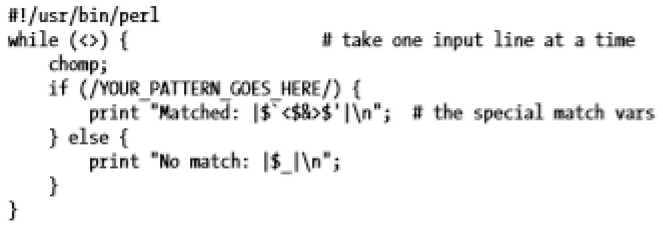


Table 8-1. Regular expression precedence

Regular expression feature	Example
Parentheses (grouping or capturing)	(...), (?:...), (?<LABEL>...)
Quantifiers	a*, a+, a?, a{n,m}
Anchors and sequence	abc, ^, $, \A, \b, \z, \Z
Alternation	a|b|c
Atoms	a, [abc], \d, \1, \g{2}

A Pattern Test Program



```
#!/usr/bin/perl
while (<>) {                    # take one input line at a time
    chomp;
    if (/YOUR_PATTERN_GOES_HERE/) {
        print "Matched: |$`<$&>$'|\n";  # the special match vars
    } else {
        print "No match: |$_|\n";
    }
}
```

A generic pattern match program to test any combination of patterns you want.

Chapter 9 - Processing Text with Regular Expressions

Substitutions with `s///`

Search and Replace

```
$_ = "He's out bowling with Barney tonight.";
s/Barney/Fred/;  # Replace Barney with Fred
print "$_\n";
```

Global Replacements with `/g`

- Calls `s///` to make all possible non-overlapping replacements.
-
  ```
  $_ = "home, sweet home!";
  s/home/cave/g;
  print "$_\n";  # "cave, sweet cave!"
  ```
- Most commonly used to collapse whitespace.
-
  ```
  $_ = "Input  data\t may have    extra whitespace.";
  s/\s+/ /g;  # Now it says "Input data may have extra whitespace."
  ```

Different Delimiters

- Can use nonpaired delimiters as normal.
- Delimiters with open/close, use 2 pairs.

  -
    ```
    s{fred}{barney};
    s[fred](barney);
    s<fred>#barney#;
    ```

Substitution Modifiers

- Can use `/i`, `/x`, and `/s` in addition to `/g`.
-
  ```
  s#wilma#Wilma#gi;  # replace every WiLmA or WILMA with Wilma
  s{__END__.*}{}s;   # chop off the end marker and all following lines
  ```

The Binding Operator

-
  ```
  $file_name =~ s#^.*/##s;  # In $file_name, remove any Unix-style path
  ```

Nondestructive Substitutions

-
  ```
  my $original = 'Fred ate 1 rib';
  my $copy = $original;
  $copy =~ s/\d+ ribs?/10 ribs/;
  ```
- `/r` modifier allows copy, keeping original intact.

  -
    ```
    use 5.014;

    my $copy = $original =~ s/\d+ ribs?/10 ribs/r;
    ```

Case Shifting

- `\U` forces uppercase.

  -
    ```
    $_ = "I saw Barney with Fred.";
    s/(fred|barney)/\U$1/gi;  # $_ is now "I saw BARNEY with FRED."
    ```
- `\L` forces lowercase.

  -
    ```
    s/(fred|barney)/\L$1/gi;  # $_ is now "I saw barney with fred."
    ```
- `\E` turns off case shifting.

  -
    ```
    s/(\w+) with (\w+)/\U$2\E with $1/i;  # $_ is now "I saw FRED with barney."
    ```
- `\l` and `\u` affect only next character.

  -
    ```
    s/(fred|barney)/\u$1/ig;  # $_ is now "I saw FRED with Barney."
    ```
- Stacking is allowed (i.e using \u with \L to make all lowercase except first character).

-
  ```
  s/(fred|barney)/\u\L$1/ig;  # $_ is now "I saw Fred with Barney."
  ```

The `split` Operator

Splits string based on separator.

```
my @fields = split /separator/, $string;
```

```
my @fields = split /:/, "abc:def:g:h";  # gives ("abc", "def", "g", "h")
```

Default is to break up string based on whitespace.

```
my @fields = split;  # like split /\s+/, $_;
```

The `join` Function

Uses no patterns, but glues piece of strings into one.

```
my $result = join $glue, @pieces;
```

```
my $x = join ":", 4, 6, 8, 10, 12;  # $x is "4:6:8:10:12"
```

`m//` in List Context

Return value is a list of capture variables created in the match or empty list if the match failed.

```
$_ = "Hello there, neighbor!";
my($first, $second, $third) = /(\S+) (\S+), (\S+)/;
print "$second is my $third\n";
```

More Powerful Regular Expressions

Nongreedy Quantifiers

- **`+`** - greedy
- **`+?`** - nongreedy, prefers to match as few times as possible.

Matching Multiple-Line Text

- **`/m`** regular expression option lets string match at internal newlines.

-
  ```
  print "Found 'wilma' at start of line\n" if /^wilma\b/im;
  ```

Updating Many Files

- **`<>`** helps with editing files.
- Example Program

  -
    ```
    #!/usr/bin/perl -w

    use strict;

    chomp(my $date = `date`);
    $^I = ".bak";

    while (<>) {
        s/^Author:.*/Author: Randal L. Schwartz/;
        s/^Phone:.*\n//;
        s/^Date:.*/Date: $date/;
        print;
    }
    ```
  - **`$^I`** saves original file while edits are saved in new file.

In-Place Editing from the Command Line

-
  ```
  $ perl -p -i.bak -w -e 's/Randall/Randal/g' fred*.dat
  ```
- Works similar to example program above.

Chapter 10 - More Control Structures

The `unless` Control Structure

Executes block of code when conditional is false (opposite of if).

```
unless ($fred =~ /\A[A-Z_]\w*\z/i) {
    print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

The `else` Clause with `unless`

```
unless ($mon =~ /\AFeb/) {
    print "This month has at least thirty days.\n";
} else {
    print "Do you see what's going on here?\n";
}
```

The `until` Control Structure

Reverse condition of while loop, repeats as long as conditional is false.

```
until ($j > $i) {
    $j *= 2;
}
```

Expression Modifiers

An expression may be followed by a modifier that controls it.

```
print "$n is a negative number.\n" if $n < 0;
```

The conditional is still evaluated first, even though it's at the end.

```
&error("Invalid input") unless &valid($input);
$i *= 2 until $i > $j;
print " ", ($n += 2) while $n < 10;
&greet($_) foreach @person;
```

The Naked Block Control Structure

A block without a keyword or conditional. The block of code is executed only once.

```
{
    body;
    body;
    body;
}
```

The `elsif` Clause

Used for checking a number of conditional expressions.

```
if ( ! defined $dino) {
    print "The value is undef.\n";
} elsif ($dino =~ /^-?\d+\.?$/) {
    print "The value is an integer.\n";
} elsif ($dino =~ /^-?\d*\.\d+$/) {
    print "The value is a _simple_ floating-point number.\n";
} elsif ($dino eq '') {
    print "The value is the empty string.\n";
} else {
    print "The value is the string '$dino'.\n";
}
```

Autoincrement and Autodecrement

`++` adds one to scalar variable
`--` substracts one to scalar variable

```
my $bedrock = 42;
$bedrock++;  # add one to $bedrock; it's now 43
$bedrock--;  # subtract one from $bedrock; it's 42 again
```

The Value of Autoincrement

- Preincrement/Predecrement

  -
    ```
    my $m = 5;
    my $n = ++$m;  # increment $m to 6, and put that value into $n
    my $c = --$m;  # decrement $m to 5, and put that value into $c
    ```
- Postincrement/Postdecrement

  -
    ```
    my $d = $m++;  # $d gets the old value (5), then increment $m to 6
    my $e = $m--;  # $e gets the old value (6), then decrement $m to 5
    ```

The `for` Control Structure

```
for (initialization; test; increment) {
    body;
    body;
}

for ($i = 1; $i <= 10; $i++) {  # count from 1 to 10
    print "I can count to $i!\n";
}
```

The Secret Connection Between `foreach` and `for`

- **`foreach`** is equivalent to **`for`** in Perl parser.

Loop Controls

The `last` Operator

- Immediately ends execution of the loop.

```
# Print all input lines mentioning fred, until the __END__ marker
while (<STDIN>) {
    if (/__END__/) {
        # No more input on or after this marker line
        last;
    } elsif (/fred/) {
        print;
    }
}
## last comes here ##
```

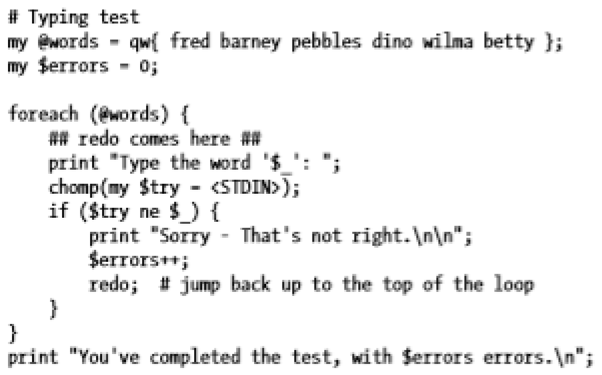
The `next` Operator

- Jumps to the inside of the bottom of the current loop block.

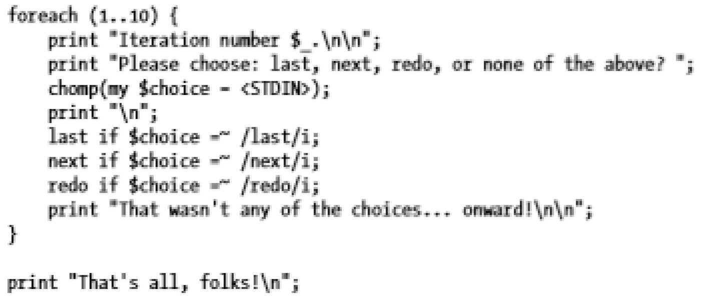
```
# Analyze words in the input file or files
while (<>) {
    foreach (split) {  # break $_ into words, assign each to $_ in turn
        $total++;
        next if /\W/;    # strange words skip the remainder of the loop
        $valid++;
        $count{$_}++;    # count each separate word
        ## next comes here ##
    }
}
```

The `redo` Operator

- Goes back to the top of the current loop block, without testing any conditional expression or advancing to the next iteration.

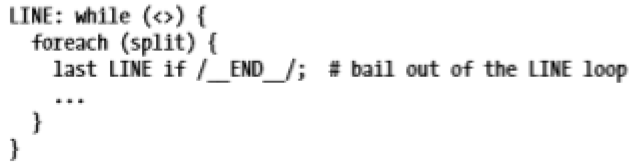
- 

```
# Typing test
my @words = qw{ fred barney pebbles dino wilma betty };
my $errors = 0;

foreach (@words) {
    ## redo comes here ##
    print "Type the word '$_': ";
    chomp(my $try = <STDIN>);
    if ($try ne $_) {
        print "Sorry - That's not right.\n\n";
        $errors++;
        redo;  # jump back up to the top of the loop
    }
}
print "You've completed the test, with $errors errors.\n";
```

- Example program to test 3 operators.
    - 

```
foreach (1..10) {
    print "Iteration number $_.\n\n";
    print "Please choose: last, next, redo, or none of the above? ";
    chomp(my $choice = <STDIN>);
    print "\n";
    last if $choice =~ /last/i;
    next if $choice =~ /next/i;
    redo if $choice =~ /redo/i;
    print "That wasn't any of the choices... onward!\n\n";
}

print "That's all, folks!\n";
```

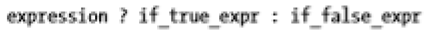
Labeled Blocks

- Use labeled blocks to work with a loop block that's not the innermost one.
- Labels are made of letters, digits, and underscores.
    - Can't start with a digit.
    - No prefix character.
    - Recommended to be all uppercase.
- Put label and colon in front of loop to specify loop block.
    - 

```
LINE: while (<>) {
  foreach (split) {
    last LINE if /__END__/;  # bail out of the LINE loop
    ...
  }
}
```

The Conditional Operator ?:

Shorthand if-then-else statement



```
expression ? if_true_expr : if_false_expr
```

Logical Operators

AND (&&)
OR (||)
short circuit operator - evaluates left side ONLY if it meets logical operator criteria.

The Values of a Short Circuit Operator

- Value is the last part evaluated.
    - True if the whole thing is true, false if the whole thing is false.

The defined-or Operator

- `//` - short circuits when it finds a defined value, no matter if the value of the left hand side is true or false.

Control Structures Using Partial-Evaluation Operators

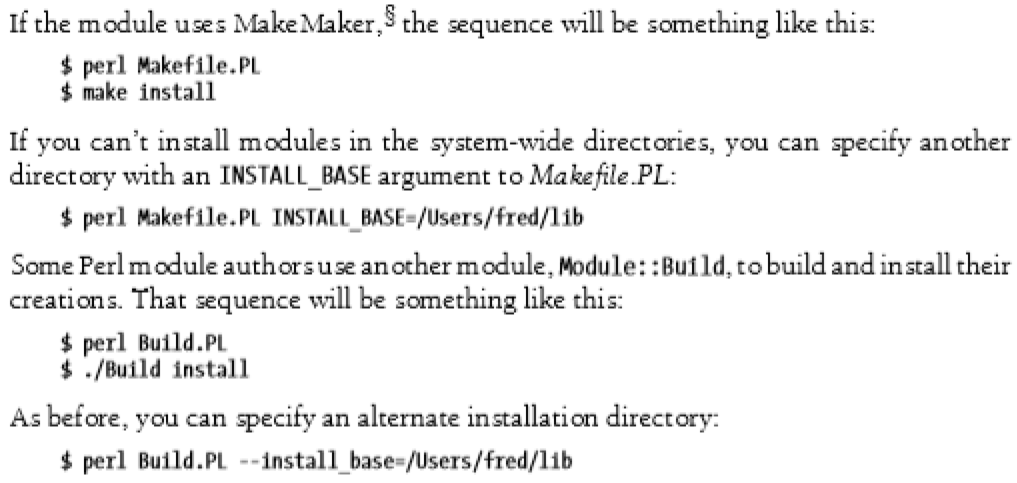
- **&&**, **||**, **//**, **?:** may or may not evaluate an expression.
- `($m < $n) && ($m = $n);`
    - Logical AND isn't being assigned anywhere. Only if the left side is true will the right side be evaluated. Equivalent to this:
        - `if ($m < $n) { $m = $n }`

Chapter 11 - Perl Modules

Finding Modules

Two types: one that come with Perl or those from CPAN you install yourself.
`perldoc (module)` is used to search documentation of a Perl module.
`cpan -a` creates a list of installed modules with version numbers.

Installing Modules



If the module uses MakeMaker,§ the sequence will be something like this:

```
$ perl Makefile.PL
$ make install
```

If you can't install modules in the system-wide directories, you can specify another directory with an INSTALL_BASE argument to *Makefile.PL*:

```
$ perl Makefile.PL INSTALL_BASE=/Users/fred/lib
```

Some Perl module authors use another module, Module::Build, to build and install their creations. That sequence will be something like this:

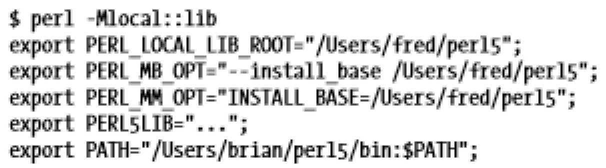
```
$ perl Build.PL
$ ./Build install
```

As before, you can specify an alternate installation directory:

```
$ perl Build.PL --install_base=/Users/fred/lib
```

Using Your Own Directories

- **`local::lib`** (provided by CPAN) is to keep new modules in their own directories, rather than placed where Perl is.

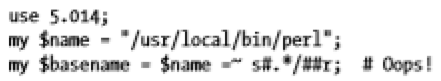


```
$ perl -Mlocal::lib
export PERL_LOCAL_LIB_ROOT="/Users/fred/perl5";
export PERL_MB_OPT="--install_base /Users/fred/perl5";
export PERL_MM_OPT="INSTALL_BASE=/Users/fred/perl5";
export PERL5LIB="...";
export PATH="/Users/brian/perl5/bin:$PATH";
```

-

Using Simple Modules

Example to get basename from a directory

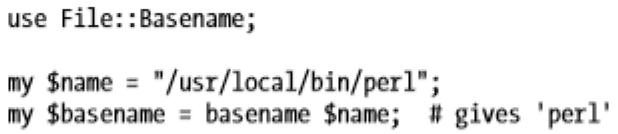


```
use 5.014;
my $name = "/usr/local/bin/perl";
my $basename = $name =~ s#.*/##r;  # Oops!
```

The problem?
1. The **.** regular expression can't detect a newline and with a UNIX type directory name, this is possible.
2. It's UNIX-specific, meaning it's assuming that all directories are going to have forward slash separators.
3. We are trying to solve a problem that has already been solved…

The `File::Basename` Module

- Extracts the basename of a file without the need of the example above.



```
use File::Basename;

my $name = "/usr/local/bin/perl";
my $basename = basename $name;  # gives 'perl'
```

-

Using Only Some Functions from a Module

- File::Basename allows import list of functions to use.

- Why? In case your script and the module have the same subroutine name.
    - If you need to use your subroutine, then to invoke the modules subroutine, you must use the full name.

    -
      ```
      use File::Basename qw/ /;      # import no function names

      my $betty = &dirname($wilma);  # uses your own subroutine &dirname
                                     # (not shown)

      my $name = "/usr/local/bin/perl";
      my $dirname = File::Basename::dirname $name;  # dirname from the module
      ```

The `File::Spec` Module

- Manipulating file specifications (files, directories, etc.)

-
  ```
  use File::Spec;

  print "Please enter a filename: ";
  chomp(my $old_name = <STDIN>);

  my $dirname  = dirname $old_name;
  my $basename = basename $old_name;

  $basename =~ s/^/not/;  # Add a prefix to the basename
  my $new_name = File::Spec->catfile($dirname, $basename);

  rename($old_name, $new_name)
    or warn "Can't rename '$old_name' to '$new_name': $!";
  ```

`Path::Class`

- Doesn't come with Perl, but has a more pleasant interface than **`File::Spec`**.

-
  ```
  my $dir     = dir( qw(Users fred lib) );
  my $subdir  = $dir->subdir( 'perl5' );     # Users/fred/lib/perl5
  my $parent  = $dir->parent;                # Users/fred

  my $windir  = $dir->as_foreign( 'Win32' ); # Users\fred\lib
  ```

CGI.pm

- Used for creating CGI program.
- Example: Creating HTML tag.

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

      use CGI qw(:all);

      print header(),
          start_html("This is the page title"),
          h1( "Input parameters" );

      my $list_items;
      foreach my $param ( param() ) {
          $list_items .= li( "$param: " . param($param) );
      }
      ```

Database and DBI

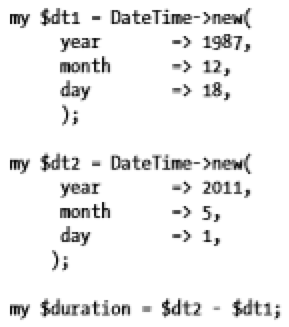
- Database Interface module doesn't come with Perl.
- One installed, needs a Database Driver (DBD).

Dates and Times

DateTime module by Dave Rolshy

```
print $dt->ymd;           # 2011-04-23
print $dt->ymd('/');      # 2011/04/23
print $dt->ymd('');       # 20010423
```

Date/Time Arithmetic



```
my $dt1 = DateTime->new(
     year       => 1987,
     month      => 12,
     day        => 18,
     );

my $dt2 = DateTime->new(
     year       => 2011,
     month      => 5,
     day        => 1,
    );

my $duration = $dt2 - $dt1;
```

Chapter 13 - Directory Operations

Moving Around the Directory Tree

working directory - starting point for relative pathnames
`chdir` - changes working directory
Tilde prefix with chdir will not work. That is a function of the shell, not the operating system, which Perl is using.

Globbing

Shell expands any filename patterns on each command line into matching filenames (globbing).

```
$ echo *.pm
barney.pm dino.pm fred.pm wilma.pm
$
```

`glob` operator is used to match filenames as well.

```
my @all_files = glob '*';
my @pm_files = glob '*.pm';
```

An Alternate Syntax for Globbing

Angle-bracket syntax was the old way of globbing.

```
my @all_files = <*>;    # exactly the same as my @all_files = glob "*";
```

```
my @files = <FRED/*>;   # a glob
my @lines = <FRED>;     # a filehandle read
my @lines = <$fred>;    # a filehandle read
my $name = 'FRED';
my @files = <$name/*>;  # a glob
```

`readline` operator used to get operation of an indirect filehandle read.

```
my $name = 'FRED';
my @lines = readline FRED;  # read from FRED
my @lines = readline $name; # read from FRED
```

Directory Handles

Looks and acts like a filehandle.
Use **`opendir`** to open, **`readdir`** to read, and **`closedir`** to close.

```
opendir DIR, $dir_to_process
    or die "Cannot open $dir_to_process: $!";
foreach $file (readdir DIR) {
  print "one file in $dir_to_process is $file\n";
}
closedir DIR;
```

Instead of reading contents of a file, you read the names of files.
Directory handles are automatically closed at the end of the program or if the directory handle is is reopened into another directory.

Recursive Directory Listing

`File::Find` library for recursive directory processing.
Can convert Unix **`find`** to Perl **`find`** using **`find2perl`**. Uses the same arguments as find.

Manipulating Files and Directories

Perl is very Unix-centric, but works the same way on non-Unix systems.

Removing Files

Perl uses **unlink** operator with a list of the file you want to remove.

```
unlink 'slate', 'bedrock', 'lava';

unlink qw(slate bedrock lava);
```

Can combine unlink and glob since they both take lists.

```
unlink glob '*.o';
```

Renaming Files

rename function

```
rename 'old', 'new';
```

Ex. Renaming files with .old to .new.

```
foreach my $file (glob "*.old") {
  my $newfile = $file;
  $newfile =~ s/\.old$/.new/;
  if (-e $newfile) {
    warn "can't rename $file to $newfile: $newfile exists\n";
  } elsif (rename $file => $newfile) {
    # success, do nothing
  } else {
    warn "rename $file to $newfile failed: $!\n";
  }
}
```

Links and Files

mounted volume - hard disk drive
inode - disk real estate, a number assigned to a file or directory
- holds a number called a link count
  - How many times it's listed in a directory.

link function creates a new link.

```
link 'chicken', 'egg'
  or warn "can't link chicken to egg: $!";
```

Can't add links to directories, it would break the hierarchy and commands like **find** and **pwd** would get lost.
Can use **symlink** as a workaround.

```
symlink 'dodgson', 'carroll'
  or warn "can't symlink dodgson to carroll: $!";
```

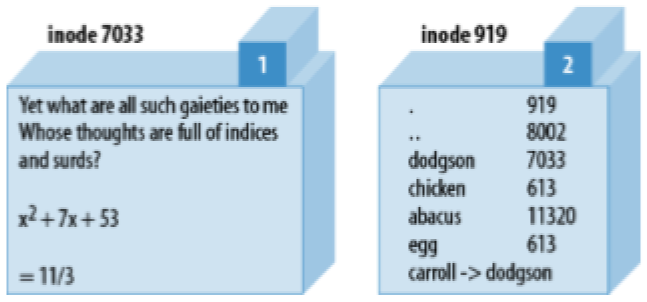


Figure 13-3. A symlink to inode 7033

readline tells you where symlink leads.

```
my $where = readlink 'carroll';            # Gives "dodgson"
```

unlink will remove association, decrement link count, and possibly free inode.

Making and Removing Directories

`mkdir` (requires octal number for setting permissions)

```
mkdir 'fred', 0755 or warn "Cannot make fred directory: $!";
```

`oct()` forces octal interpretation of string.

```
mkdir $name, oct($permissions);
```

`rmdir`

```
foreach my $dir (qw(fred barney betty)) {
  rmdir $dir or warn "cannot rmdir $dir: $!\n";
}
```

`rmdir` fails on non-empty directories. Use **`unlink`** to remove directory contents, then use **`rmdir`**.

```
my $temp_dir = "/tmp/scratch_$$";       # based on process ID; see the text
mkdir $temp_dir, 0700 or die "cannot create $temp_dir: $!";
...
# use $temp_dir as location of all temporary files
...
unlink glob "$temp_dir/* $temp_dir/.*"; # delete contents of $temp_dir
rmdir $temp_dir;                        # delete now-empty directory
```

Modifying Permissions

`chmod`

```
chmod 0755, 'fred', 'barney';
```

Symbolic permissions (i.e **`+x`**, **`go=u-w`**) do not work in Perl.

Changing Ownership

`chown` - need numeric user and group ID values

```
my $user  = 1004;
my $group = 100;
chown $user, $group, glob '*.o';
```

Changing Timestamps

`utime` (access time, modification time)

```
my $now = time;
my $ago = $now - 24 * 60 * 60;  # seconds per day
utime $now, $ago, glob '*';     # set access to now, mod to a day ago
```

Chapter 14 - Strings and Sorting

Finding a Substring with `index`

`index` gives you the integer location of the first character of the substring you're looking for.

```
$where = index($big, $small);
```

A third parameter is available to tell **`index`** where to start.

```
my $stuff  = "Howdy world!";
my $where1 = index($stuff, "w");               # $where1 gets 2
my $where2 = index($stuff, "w", $where1 + 1);  # $where2 gets 6
my $where3 = index($stuff, "w", $where2 + 1);  # $where3 gets -1 (not found)
```

`rindex` gives integer location of the last character.
Third parameter will give the maximum permitted return value.

```
my $fred = "Yabba dabba doo!";

my $where1 = rindex($fred, "abba");  # $where1 gets 7
my $where2 = rindex($fred, "abba", $where1 - 1);  # $where2 gets 1
my $where3 = rindex($fred, "abba", $where2 - 1);  # $where3 gets -1
```

Manipulating a Substring with `substr`

Works with part of a larger string.

```
my $part = substr($string, $initial_position, $length);
```

Formatting Data with `sprintf`

Takes the same arguments as printf (except for optional filehandle), but returns string instead of printing it.

```
my $date_tag = sprintf
    "%4d/%02d/%02d %2d:%02d:%02d",
    $yr, $mo, $da, $h, $m, $s;
```

Using `sprintf` with "Money Numbers"

- **`%.2f`** formats numbers with a certain number of places after decimal point.
    - `my $money = sprintf "%.2f", 2.49997;`
- For "Money Numbers" that may need commas.

    -
      ```
      sub big_money {
          my $number = sprintf "%.2f", shift @_;
          # Add one comma each time through the do-nothing loop
          1 while $number =~ s/^(-?\d+)(\d\d\d)/$1,$2/;
          # Put the dollar sign in the right place
          $number =~ s/^(-?)/$1\$/;
          $number;
      }
      ```

Interpreting Non-Decimal Numerals

-
  ```
  hex('DEADBEEF')     # 3_735_928_559 decimal
  hex('0xDEADBEEF')   # 3_735_928_559 decimal

  oct('0377')         # 255 decimal
  oct('377')          # 255 decimal
  oct('0xDEADBEEF')   # 3_735_928_559 decimal, saw leading 0x
  oct('0b1101')       # 13 decimal, saw leading 0b
  oct("0b$bits")      # convert $bits from binary
  ```

Advanced Sorting

Numeric sort

```
sub by_number {
    # a sort subroutine, expect $a and $b
    if ($a < $b) { -1 } elsif ($a > $b) { 1 } else { 0 }
}
```

Numeric sort shortcut using spaceship operator (**<=>**)

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

`cmp` is like the spaceship operator, but applies to strings.

```
sub by_code_point { $a cmp $b }

my @strings = sort by_code_point @any_strings;
```

Case Insensitive Sort

```
sub case_insensitive { "\L$a" cmp "\L$b" }
```

Reverse sorting

```
my @descending = reverse sort { $a <=> $b } @some_numbers;
```

Reverse sorting just by switching the variables!

```
my @descending = sort { $b <=> $a } @some_numbers;
```

Sorting a Hash by Value

```
my %score = ("barney" => 195, "fred" => 205, "dino" => 30);
my @winners = sort by_score keys %score;
```

- You are comparing the numeric values, rather than key values.

```
sub by_score { $score{$b} <=> $score{$a} }
```

Sorting by Multiple Keys

```
my @winners = sort by_score_and_name keys %score;

sub by_score_and_name {
    $score{$b} <=> $score{$a}  # by descending numeric score
        or
    $a cmp $b                  # code point order by name
    }
```

Chapter 15 - Smart Matching and given-when

The Smart Match Operator

smart match operator (**~~**) looks at both operands and decides how to compare them.
Applies to any Perl version starting with 5.10.1 and beyond.

```
say "I found Fred in the name!" if $name ~~ /Fred/;
```

Smart Match Precedence

Table 15-1. Smart match operations for pairs of operands

Example	Type of match
`%a ~~ %b`	hash keys identical
`%a ~~ @b` or `@a ~~ %b`	at least one key in `%a` is in `@b`
`%a ~~ /Fred/` or `/Fred/ ~~ %b`	at least one key matches pattern
`'Fred' ~~ %a`	`exists $a{Fred}`
`@a ~~ @b`	arrays are the same
`@a ~~ /Fred/`	at least one element in `@a` matches pattern
`$name ~~ undef`	`$name` is not defined
`$name ~~ /Fred/`	pattern match
`123 ~~ '123.0'`	numeric equality with "numish" string
`'Fred' ~~ 'Fred'`	string equality
`123 ~~ 456`	numeric equality

Match operator is not always commutative.

```
say "match number ~~ string" if 4 ~~ '4abc';
say "match string ~~ number" if '4abc' ~~ 4;
```

The second one is the only one that prints anything.

The `given` Statement

Allows you to run a block of code when the argument to given satisfies a condition.
This is the Perl equivalent to C's **`switch`** statement.

```
given ( $ARGV[0] ) {
    when ( 'Fred'  )  { say 'Name is Fred' }
    when ( /fred/i )  { say 'Name has fred in it' }
    when ( /\AFred/ ) { say 'Name starts with Fred' }
    default           { say "I don't see a Fred" }
}
```

Dumb Matching

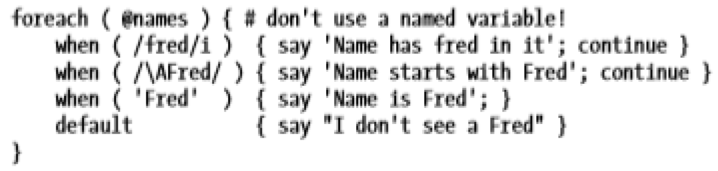
- Using explicit comparison operators rather than using the default smart matching operator.

```
given ( $ARGV[0] ) {
    when ( $_ eq 'Fred'  )  { say 'Name is Fred'; continue }
    when ( $_ =~ /\AFred/ ) { say 'Name starts with Fred'; continue }
    when ( $_ =~ /fred/i )  { say 'Name has fred in it'; }
    default                 { say "I don't see a Fred" }
}
```

-

Using when with Many Items

You can use **`foreach`** in the form of **`given`** to use when with many items such as an array.



```
foreach ( @names ) { # don't use a named variable!
    when ( /fred/i )  { say 'Name has fred in it'; continue }
    when ( /\AFred/ ) { say 'Name starts with Fred'; continue }
    when ( 'Fred'  )  { say 'Name is Fred'; }
    default           { say "I don't see a Fred" }
}
```

Chapter 16 - Process Management

The `system` Function

This creates a copy of your Perl program, called the child process.
Parameter is whatever you normally type in a shell.

```
system 'date';
```

Avoiding the Shell

- Invoking the system operator with more than one argument doesn't get the shell involved.

    -
      ```
      my $tarfile = 'something*wicked.tar';
      my @dirs = qw(fred|flintstone <barney&rubble> betty );
      system 'tar', 'cvf', $tarfile, @dirs;
      ```

The Environment Variables

System environment variables are stored in a Hash called **`%ENV`**.

```
$ENV{'PATH'} = "/home/rootbeer/bin:$ENV{'PATH'}";
delete $ENV{'IFS'};
my $make_result = system 'make';
```

The exec Function

Causes Perl process itself to perform request action rather than creating a child process.

```
chdir '/tmp' or die "Cannot chdir /tmp: $!";
exec 'bedrock', '-o', 'args1', @ARGV;
```

Using Backquotes to Capture Output

Capture output of command as string value rather than processing it using backquotes.

```
my $now = `date`;             # grab the output of date
print "The time is now $now"; # newline already present
```

qx() quoted operator does the same thing.

```
foreach (@functions) {
  $about{$_} = qx(perldoc -t -f $_);
}
```

Using Backquotes in a List Context

- Backquoted string in list context yields a list containing one line of output per element.

    -
      ```
      my $who_text = `who`;
      my @who_lines = split /\n/, $who_text;
      ```

External Processes with `IPC::System::Simple`

Available through CPAN.
Provides simpler interface that hides the complexity of operating system specific stuff when running or capturing output from external commands.

```
use IPC::System::Simple qw(system);

my $tarfile = 'something*wicked.tar';
my @dirs = qw(fred|flintstone <barney&rubble> betty );
system 'tar', 'cvf', $tarfile, @dirs;
```

Gives a more robust **`system`** command, **`systemx`** (avoiding invoking shell), `capture` (backquoting), and **`capturex`**.

Processes As Filehandles

Launching child process that stays alive, put command as filename in **open** call, preceding or following it with a pipe (**|**). AKA “piped open”.

```
open DATE, 'date|' or die "cannot pipe from date: $!";
open MAIL, '|mail merlyn' or die "cannot pipe to mail: $!";
```

Getting Down and Dirty with Fork

Low-level **system** call.
Allows for full control over creating pipes, rearranging filehandles, and knowing Process ID of parent process.

```
defined(my $pid = fork) or die "Cannot fork: $!";
unless ($pid) {
  # Child process is here
  exec 'date';
  die "cannot exec date: $!";
}
# Parent process is here
waitpid($pid, 0);
```

equivalent to `system 'date';`

Sending and Receiving Signals

Use **kill** to send **SIGINT** to a process. Must know process’ ID to do so.

```
kill 2, 4201 or die "Cannot signal 4201 with SIGINT: $!";
```

Chapter 17 - Some Advanced Perl Techniques

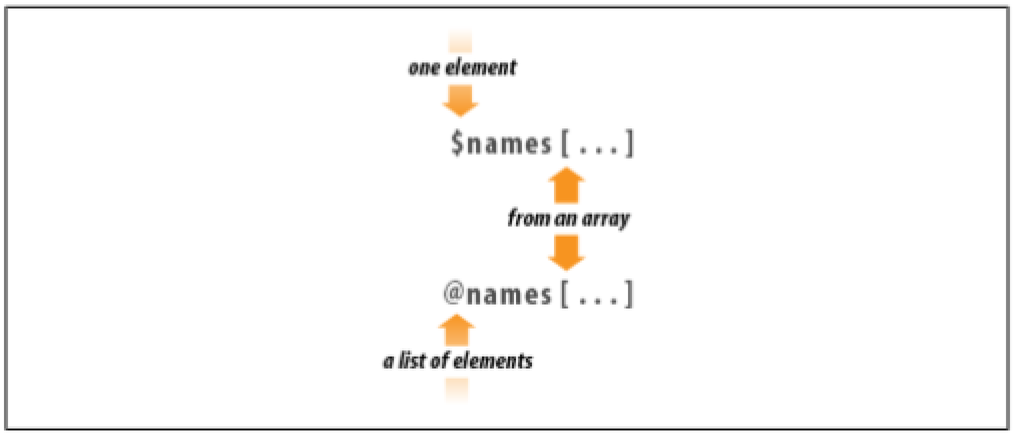
Slices

Simplest way to pull items from a list.
"A list slice has to have a subscript expression in square brackets after a list in parentheses."

```
my $card_num = (split /:/)[1];
my $count = (split /:/)[5];
```

Array Slice

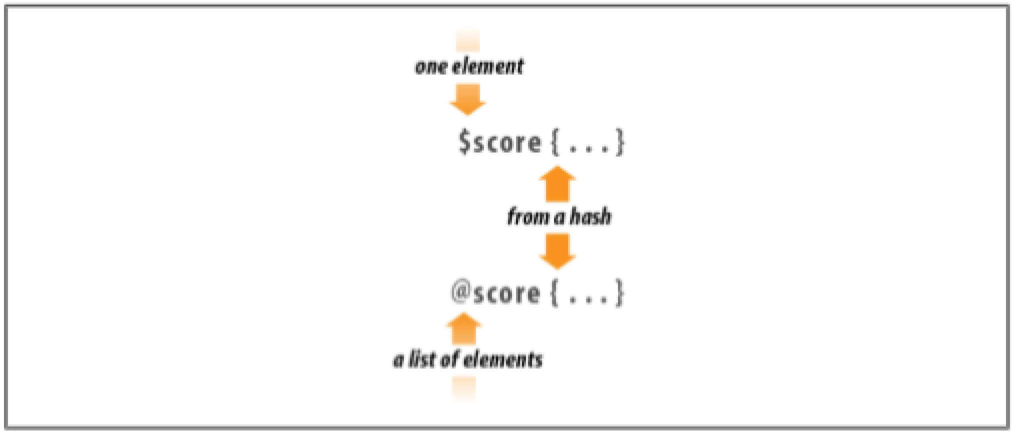
- `my @numbers = @names[ 9, 0, 2, 1, 0 ];`
- **`$name[]`** represents getting one element.
- **`@name[]`** represents getting a list of elements.



- *Figure 17-1. Array slices versus single elements*

Hash Slice

-
  ```
  my @three_scores = ($score{"barney"}, $score{"fred"}, $score{"dino"});
  my @three_scores = @score{ qw/ barney fred dino/ };
  ```



- *Figure 17-2. Hash slices versus single elements*

Trapping Errors

Using eval

- Wrap potential crashing code in eval block to allow normal program flow.
  - `eval { $barney = $fred / $dino };`
- Returns undef if fatal error is found.

- Use defined-or operator to set own default value.

    -
      ```
      my $barney = eval { $fred / $dino } // 'NaN';
      ```
- 4 problems **`eval`** can't trap.
    - Syntax errors in literal sense (mismatched quotes, missing semicolons, missing operands, invalid literal regular expressions)
    - Serious errors that crash Perl (out of memory)
    - Warnings (user generated or Perl's internal ones)
    - **`exit`** operator (can't stop its intended job!)

<u>More Advanced Error Handling</u>

- Throw an exception with **`die`** and catch with **`eval`**.

```
eval {
  ...;
  die "An unexpected exception message" if $unexpected;
  die "Bad denominator" if $dino == 0;
  $barney = $fred / $dino;
  }
if ( $@ =~ /unexpected/ ) {
  ...;
  }
elsif( $@ =~ /denominator/ ) {
  ...;
  }
```

<u>`autodie`</u>

- Pragma that gives you more control over how you handle errors in your program.
- Applies error message to operators by default.
    - You are used to this:

        -
          ```
          open my $fh, '>', $filename or
            die "Couldn't open $filename for writing: $!";
          ```
        - With autodie:

            -
              ```
              use autodie;

              open my $fh, '>', $filename; # still dies on error
              ```

<u>Picking Items from a List with `grep`</u>

```
my @odd_numbers = grep { $_ % 2 } 1..1000;
```

<u>Transforming Items from a List with `map`</u>

Makes a copy from a list, changes the format, and returns the newly formatted list.

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);
print "The money numbers are:\n",
  map { sprintf("%25s\n", &big_money($_) ) } @data;
```

<u>Fancier List Utilities</u>

`List::Util` module to perform high level performance versions of common list processing utilities at the C level.
Ex. Using **`sum`** subroutine in module to add numbers from 1 to 1000.

```
use List::Util qw(sum);
my $total = sum( 1..1000 ); # 500500
```