

# Perl Scalar Data

Beginning Perl Programming

# Keeping Up



- <http://dynamicacademy.net/perlclass>
- Readings
- Assignments

# Programs

- A program is a text file
- A series of statements
- Starts at the top, reads to the end
- Mostly whitespace independent, except...
- # comments go to the end of line



# Running a Program

- You can run a program by telling perl what file it's in:

```
% perl helloworld.pl  
Hello world!  
%
```

# Statements vs Expressions

- Valid bits of perl are either *statements* or *expressions*
- Expressions are just values, like nouns.

```
5  
"hello"  
3.14159  
2 + 2  
8 == 2 * 4  
6 > 3  
"abc" lt "def"  
'Count Dracula' . ' is a friendly guy'  
60 * 60 * 24
```

# Statements vs Expressions

- Statements are ended by a semicolon ;
- They DO something! The verbs of Perl.

```
print "Hello world";  
  
$a = 2 + 2;  
  
$area = 3.14159 * $radius * $radius  
  
if (2 + 2 == 4) {  
    print "All is right with the world";  
}
```

# Perl Types



- Going back to expressions...
- Every expression in Perl has a type
- Every language takes a different approach to types
- Some say how the language treats types is all that separates them

# Types

- Perl is known as a “typed” language, but not a “strongly typed” language.
- Types help! It helps Perl figure out what you meant and when you may have made a mistake!
- Built-in types give you useful tools



# Types help!

```
$fish1 = "red";  
$fish2 = "blue";  
$fish3 = "green";  
$fish4 = "yellow";  
  
print "All my fish:"  
print $fish1;  
print $fish2;  
print $fish3;  
print $fish4;
```

# Types help!

```
@fish = ("red", "blue", "green", "yellow");
```

```
print "All my fish:";  
print @fish;
```

```
# The built-in array type helps keep track of  
# ordered lists of things.
```

# Scalar Type

- Perl's Scalar Type is powerful and flexible at keeping track of single values.
- For perl a *string of text* is a scalar too!

# What's a Scalar?

- A single value.
- The value itself could take a lot of different forms
- Perl does its best to translate it automatically for you.
- For now, just numbers and text



# Scalars in Action

```
my $name = "Doug Treder";  
my $email = 'dtreder@gmail.com';  
my $age = 33;  
my $GPA = 2.1;  
  
if ( $GPA > 3 ) {  
    print "Good job $name!\n";  
    print "I'll send email to $email.\n";  
} else {  
    print "You need to study more $name!\n";  
}
```

# Expressions and Statements

- A single value is a scalar expressions
- Assignment takes an expression and puts it into a variable
- That's a statement!

```
5      # expression  
$x     # expression  
$x = 5; # statement!
```

# Variables

- A variable is storage for a value
- Variables start out empty (“undef”)
- Use assignment (“ = “) to put values in



# Scalar Variables (\$)

- Variable names can contain letters, numbers, and underscore, but can't start with a number.

```
$foo  
$Foo  
$foo_bar  
$fooBar  
$foo123  
$f_o_o_1_2_3  
$_not_recommended  
$__super_secret
```

# Scalar Variables (\$)

- I recommended you stick with long, descriptive names in lowercase, separated by underscores.

\$foo

\$Foo

\$foo\_bar

\$fooBar

\$foo123

\$f\_o\_o\_1\_2\_3

\$\_not\_recommended

\$\_super\_secret

# Numbers

- Perl numbers can be integers, floating point, scientific notation, very large or very small.

```
$x = 5;  
$x = 5.0000;  
$x = 5.00000000000001;  
$x = 3.14159;  
$x = -9000000000000000;  
$x = 1e24;  
$x = 1e-24;
```

# What's really going on?

- Behind the scenes:
- Perl keeps track of an integer as long as it can. Perl can be compiled with either 32 or 64bit integer support.
- If the value gets a fractional portion or you overflow the integer, Perl tracks it as a double-precision floating point value.
- So Perl is easy to write but nearly as fast at math as compiled C.

# Operators

- An operator connects expressions
- An operator makes another expression

2 + 3

5.1 - 2

3 \* 12

10.2 / 3

2 \*\* 3

10 % 3

\$a < \$b

\$e > \$f

\$g >= 67

\$t <= 2.3

\$r == \$y

\$w != 3.4

# Assignment

- Let's write a program!
- A statement is a program too.

```
$height = 100;  
$width = 150;  
$rectangle_area = $height * $width;
```

# Assignment

```
$one = 1;  
$two = 2.0;  
  
$first_name = "Joel";  
$last_name = 'Grow';  
$another_one = $one;  
$copy_last_name = $last_name;
```

# Assignment

```
$a = 4;
```

```
$b = $a + 3;
```

```
$b = $b * 2;
```

```
$d = ($c = 4);           # assign 4 to $c and $d
```

```
$d = $c = 4;           # same thing
```

```
$b = 4 + ($a = 3);     # what is $a?  $b?
```

# A note on legibility

- Just because you can do something in Perl doesn't always mean you should.
- This line is not very obvious to the code reader!
- Think of the future coders (including yourself) that will have to read and understand your code.

```
$b = 4 + ($a = 3);
```

# ShortCut Operators

```
$a += 3;    # same as $a = $a + 3;
```

```
$b -= 1;    # $b = $b - 1;
```

```
$d /= 3;    # $d = $d / 3;
```

```
$r **= 4;   # $r = $r ** 4;
```

```
$b = ($a += 4); # valid, but weird.  
                # Please don't do this!
```

```
$str .= 'foobar';
```

```
# same as $str = $str . 'foobar';
```

# Increment and Decrement

- ++ means 'add one'
- -- means 'subtract one'

```
$a = 2;  
$a++;    #same as $a = $a + 1;  
  
# $a now equals 3
```

# Pre and Post Increment

```
$a = 3;
```

```
print $a++; # prints 3, changes $a to 4
```

```
print ++$a; # prints 5, $a now 5
```

```
print --$a; # prints 4, $a now 4
```

```
print $a--; # prints 4, $a now 3
```

# Booleans

- A mathematician named George Boole did a lot of work with discrete math
- "true" and "false" often called Booleans
- Perl false is 0 or undef or "empty string" : "" or "
- Perl true is anything else

```
1;      # true
0;      # false
3.14159; # true
0.00001; # true !
```

# Strings

- Sequences of 0 or more characters
  - ""
  - 'Joel'
  - 'ugly string with !\*#@%}{ characters'
- **Any** character (8 bit, 0 thru 256)
  - Not just ASCII 32 thru ASCII 126
- Written single quoted or double quoted

# Single-Quoted Strings

- What you see is what you get :
- ‘ treats the string as *literal*

```
'camel' # five chars, 'c', ..., 'l'  
'don\'t' # includes single quote char  
' ' # null string, no chars  
'a\\b' # 'a', '\\', 'b'
```

# Double Quoted Literal Strings

- Interpolates and evaluates variables and special chars
- backslashes are special

```
"big camel\n"    # 2 words and a newline  
"James Bond"   # nothing to interpolate  
"coke\tsprite" # tab between words  
"a\\b"         # 'a', '\', 'b'  
"c:\temp"     # what's this?  
"hello $foo\n"; # replaces $foo with its value
```

# Scalar Interpolation

```
$a = "joel";  
$b = "hi $a!\n";  
print $b;
```

```
hi joel!
```

```
$c = "no such variable $foo!\n";  
print $c
```

```
no such variable !
```

# No interpolation?

```
$x = "Joel";
```

```
# if you don't want interpolation:
```

```
$z = "hey \"$x\"";
```

```
# $z is now what?
```

# Interpolation

```
$fred = 'Jones';
```

```
$x = '$fred';    # what is $x ?
```

```
$y = "hey $x";  # what is $y ?
```

# Interpolation

- We want to assign "It's payday" to \$msg

```
$fred      = "pay";  
$fredday  = "Monday";  
  
$msg       = "It's $fredday!";  
  
# what is $msg?
```

# Interpolation

```
$fred    = "pay";  
$freeday = "Monday";
```

```
# Separate the variable name from  
# possible suffixes:
```

```
$msg     = "It's $fred"."day!";
```

```
$msg     = "It's " . $fred . "day!";
```

# Alternative Quotes

- Use `q{}` like single quotes, `qq{}` like double quotes
- You can actually use any matchable delimiter: `( ){} <> []`

```
q{hello 'world'!}
```

```
qq{hello $foobar!}
```

```
q*A 'starring' \*role\**
```

```
qq|Don't "quote" me on that!\n|
```

```
q<Don't "quote" me!>
```

Saturday, September 27, 2008

37

Quote and Quote-like Operators from **perlop** man page

Customary	Generic	Meaning	Interpolates?
''	q{}	Literal	no
""	qq{}	Literal	yes
``	qx{}	Command	yes
	qw{}	Word List	no
//	m{}	Pattern Match	yes
s///	s{}	Substitution	yes
tr///	tr{}	Translation	no

If interpolating, variables beginning with `$` or `@` are interpolated, as are `\t` (tab), `\n` (newline), etc.

# Here Documents

- Can print multi-line string easily

```
print <<END_OF_TEXT;
```

```
He said  
"Hello world!"
```

```
Nifty.
```

```
END_OF_TEXT
```

# Here doc with HTML

```
print <<End_of_HTML;  
  
<HTML>  
<BODY>  
<TABLE><TR><TD>Hello world!</TD></TR>  
</TABLE>  
</BODY>  
</HTML>  
  
End_of_HTML
```

# Controlling Case

```
$big      = uc('hello');      # HELLO
$little   = lc('World');      # world
$big      = "\Uhello";        # HELLO
$big      = "\uHELLO";        # HELLo
$little   = "\LWorld";        # world
$little   = "\lwORLD";        # wORLD

$beast    = "dromedaRY";

$capitalize = ucfirst($beast); #DromedaRY
$capitalize = "\u\L$beast"     # Dromedary
```

# Operators for Strings

## - Concatenation

```
"tall" . " turtle";    # concatenation  
'what "color" is that?' . "\n";  
"a" . " " . "b";
```

# Operators for Strings

## - Repeating

`(3+2) x 4`      # is "5555"

`"-" x 50`      # is what?

# Operators for Strings

## - Comparison

```
if ($a eq 'bcd') {}  
if ($f ne $g) {}  
if ($g gt "mcgd") {}  
if ($jake lt $fred) {}  
if ($a le $b) {}  
if ($c ge $d) {}
```

# Numeric vs String Comparisons

- Context is very important in comparisons!

```
$a = 5;  
$b = 16;  
$c = 'joelg';  
$d = 'joel';
```

```
if ( $a < $b ) # numeric, true  
if ( $a lt $b ) # string, false  
if ( $c gt $d ) # string, true  
if ( $c > $d ) # numeric, false
```

# Numeric vs String Comparisons

- Very common mistake: using eq instead of ==

```
$a = 5;  
$b = 16;  
$c = 'joelg';  
$d = 'joel';
```

```
if ( $c == $d ) # numeric, true!  
if ( $c eq $d ) # string, false
```

# Chop

- chop removes (and returns) the last character in a string.

```
$x = "hello world";  
  
chop($x);           # $x is now "hello worl"  
$last_char = chop($x); # what is $last_char?
```

# Chomp

- chomp only removes the "record separator character", usually a newline.
- It's a safe chop.
- chomp is much more common than chop.
- I use chomp a lot and chop very little, if ever!

```
$var = "abc\n";  
chomp($var); # $var becomes "abc"
```

# Scalar <STDIN>

- Reads input from the user until they hit ENTER
- The newline is included too! (chomp is handy)

```
$x = <STDIN>; # wait for user to type  
# something...  
chomp ($x); # get rid of newline  
  
print "You just typed '$x'\n";
```

# Scalars in Action

```
my $name = "Doug Treder";  
my $email = 'dtreder@gmail.com';  
my $age = 33;  
my $GPA = 2.1;  
  
if ( $GPA > 3 ) {  
    print "Good job $name!\n";  
    print "I'll send email to $email.\n";  
} else {  
    print "You need to study more $name!\n";  
}
```