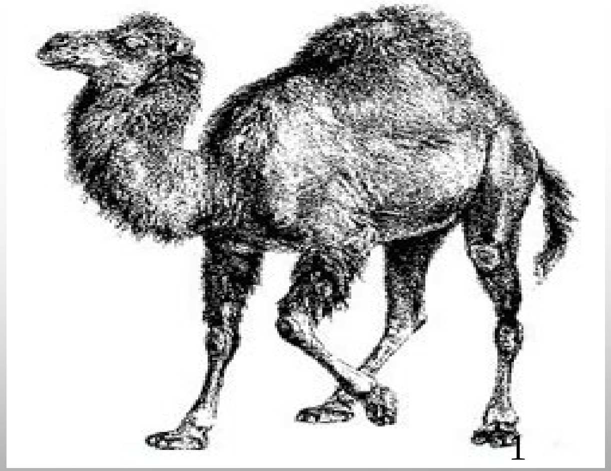
Introduction to Perl

Texas A&M High Performance Research Computing (HPRC)

Keith Jackson



Fall 2016

Acknowledgements

- Title page clip art taken from O’Reilly Publishers books *Programming Perl*.
- A few code examples taken from *Programming Perl* and *Beginning Perl for Bioinformatics*, as well as on-line references.
  (See hprc.tamu.edu)
- perlconsole was written by Alexis Sukrieh
  (See http://sukria.net/perlconsole.html)

Who Should Attend This Class?

HPRC users who need the power and simplicity of Perl to do:

- Text and pattern analysis
- File administration
- Database and network operations
- Quick, easy Unix tasks

Suggested Prerequisites

- **HPRC account**
  (See http://hprc.tamu.edu/)
- **Intro to Unix shortcourse** (http://hprc.tamu.edu/shortcourses/)
- **Experience with programming at least one language** (C, C++, FORTRAN, Java, or similar)

Agenda

- What kind of language is Perl?
- Executing your program
- Finding documentation
- Statement syntax
- Variables, constants, expressions

Agenda

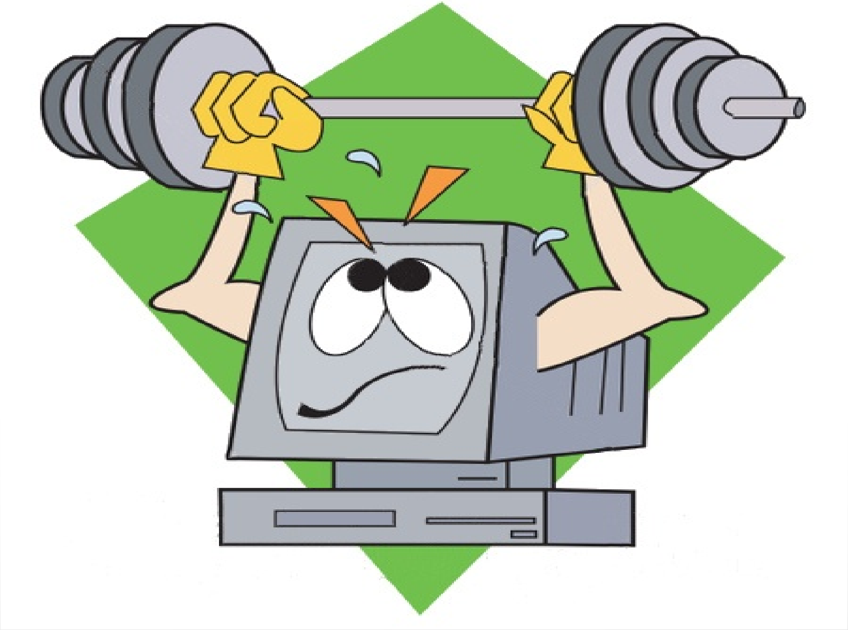
- Control flow
- Understanding error messages
- I/O
- Regular expressions
- Subroutines
- System calls
- Objects/modules

What is Perl?

1. Interpreted, dynamic programming language
2. High-level language
   - Functional
   - Procedural
   - Object-oriented
3. Extensible library modules

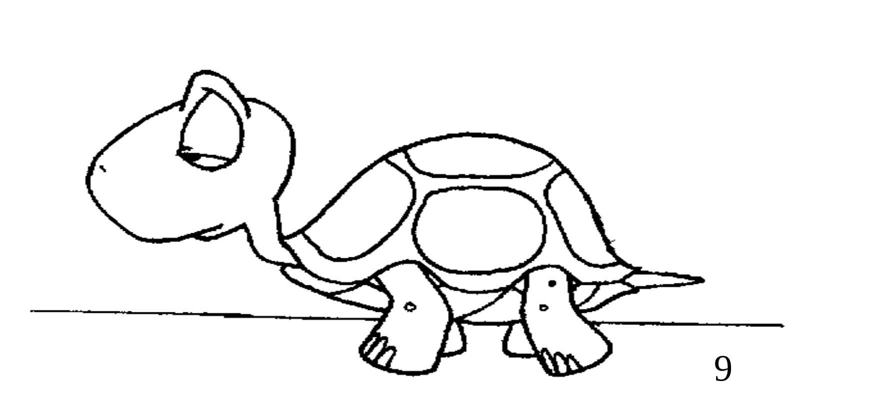
What Perl Does Well

1. Pattern matching with regular expressions
2. String processing
3. Hash tables
4. File I/O
5. Network and database access
6. CGI (website)



Limitations of Perl

1. Compiled on each run
2. Large FP calculations not as fast or as easy as FORTRAN or C/C++
3. No contiguous multi-dimensional arrays. Complex data structures have memory management overhead



How to Run a Perl Program

Usually, program is in a file with “**.pl**” suffix. You can run it from the command line with the **perl** command:

```
$ perl sum.pl
```

Run Perl with **–e**

You can run short programs with **`–e`** option:

```
$ perl –e 'printf("%f\n", exp(10 * log(2)))'
```

Be sure to put quotes around your statements so they are passed unchanged to **`perl`**.

Run with **eval perl**

You can use **eval perl** to enter statements without creating a file:

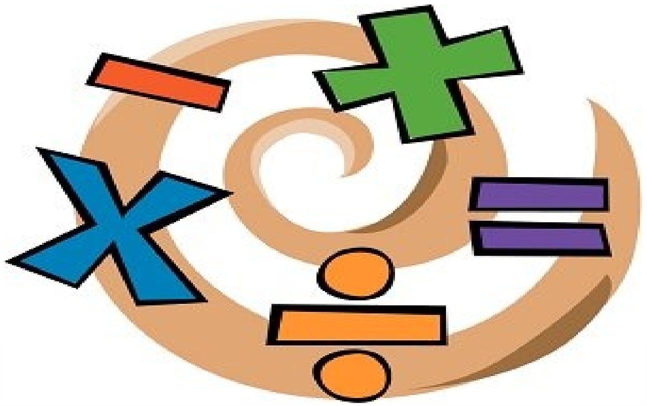
```
$ eval perl
$x = 3;
$y = 8;
printf("The sum is %d\n", $x + $y);
```

Press ctrl-d (^D) on a new line to complete input.

Testing with **`eval perl`**

Using **`eval perl`** allows you to quickly test Perl syntax without a program file.

```
$ eval perl
@a = ('red', 'green', 'blue');
print @a, "\n";
print @a . "\n";
print "@a\n";
```



Testing with **perlconsole**

perlconsole allows you to run Perl statements interactively for quick testing.

- Not a standard utility.

```
ada$ /scratch/training/Perl/bin/perlconsole
Perl Console 0.4
Perl> printf "sum is %d\n", 8 + 9;
```

Configuring **`perlconsole`**

- For informative output, create **`$HOME/.perlconsolerc`** :

```
$ echo ":set output=dumper" > ~/.perlconsolerc
```

On-Line Documentation

1. Unix man pages:

```
$ man perl
$ man perlfunc
```

2. Websites, such as:
   http://perldoc.perl.org

Perl Books

hprc.tamu.edu

The Perl Programming Language

http://www.perl.org

Variable Names

Names in Perl:

- Start with a letter
- Contain letters, numbers, and underscores "_"
- Case sensitive

Two major types:

$ Scalars (single value)

@ Lists

$x

Scalars

- Start with a dollar sign “$”
- Can be of type:
    - Integer
    - Floating point
    - String
    - Binary data
    - Reference (like a pointer)
- But Perl is not a strongly typed language

Lists

- Start with an at symbol “@”
- Also known as arrays
- Always one-dimensional
- Index starts at 0
- Can contain mixture of scalar types (not strongly typed)

Hash Tables

- Start with percent sign “%”
- Implemented as a list with special properties
- Key-Value pairs
- Keys are unique
- Keys can be any scalar value
- Values can be any scalar value

$a[3]

List Elements

- Individual array elements:
  - Scalar values
  - Start with “$”
  - Indexed in square brackets: “[ ]”

```
@a = (2, 3, 5, 7, 11);
print $a[3];
$a[5] = 13;
```

prints “7”

extends @a by one

$#a

List Size

- Assign array to scalar to get list length
- The top index is given by “**$#**”

```
@a = (2, 3, 5, 7, 11);
$len = @a;
$len = $#a + 1;
```

$len gets “5”
Same thing

$h{name}

Hash Elements

- Individual array elements:
  - Scalar values
  - Start with “$”
  - Indexed in curly brackets: “{ }”

```
%h = (name => "Sam", phone => "555-1212");
print $h{name};
$h{age} = 27;
```

prints “Sam”

extends %h by one

Same Name, Different Variables

- The same name can be reused for scalars, arrays, hashes, subroutine names, file handles, etc..
- Each of the following refer to a completely different thing:

```
$a    $A    @a    %a    &a
```

Perl Statement Syntax

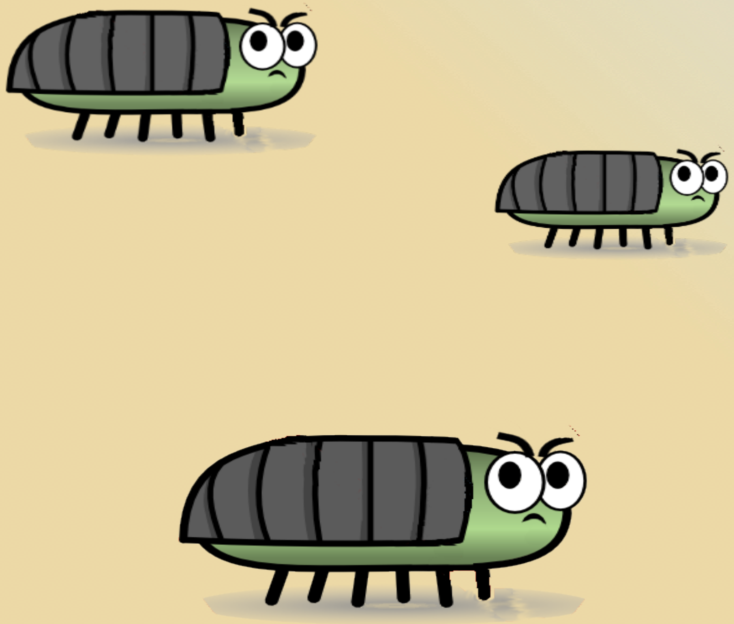
- Statements separated by semicolons: “;”
- Statement blocks surrounded by curly braces: “{ }”
- Comments preceded by pound sign: “#”

Sample Syntax

```
# see if we need to run work()
$remaining = queue_size();
if ($remaining > 0)
{
    work($x);       # does the main task
    print "did work\n";
}
```

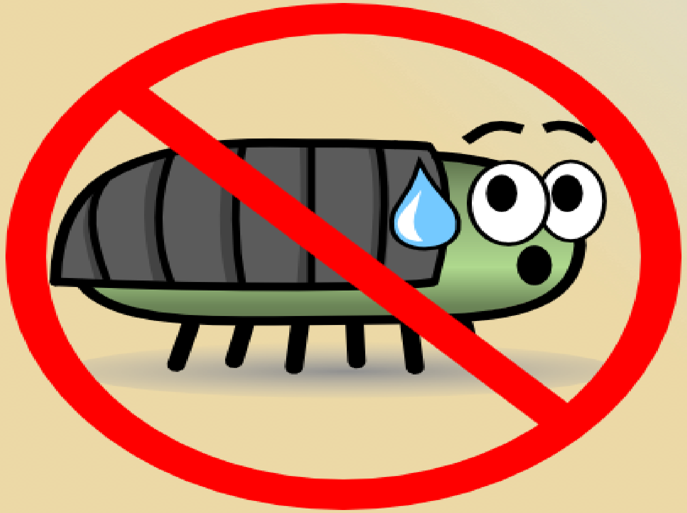
Common Syntax Errors

```
if ($remaining > 0)
    work($x);

for ($i = 1; $i < $n; $i++)
    print a[$i], "\n";

while ($c = 1)
{
    $c = do_thing($m, $q);
}
```



Fixing Errors

```
if ($remaining > 0)
{    work($x);    }

for ($i = 1; $i < $n; $i++)
{    print $a[$i], "\n";    }

while ($c == 1)
{
    $c = do_thing($m, $q);
}
```



Perl Control Statements

- Conditionals:
  - **`if/elsif/else`**
  - **`unless`** *(inverse of “**if**”)*
  - no “**`switch/case`**” in Perl
- Loops:

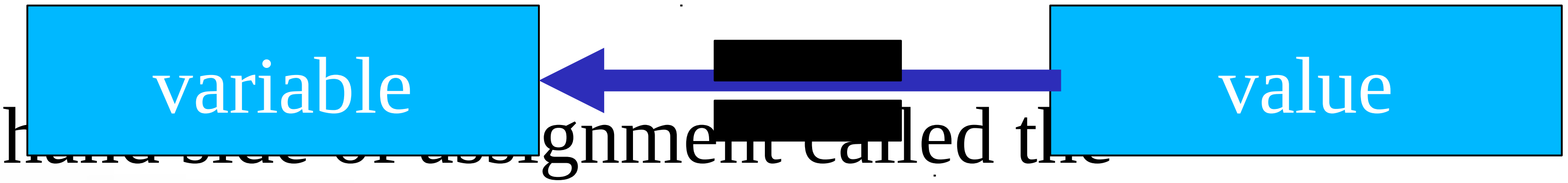
```
for(;;)            foreach ()          while()

until()            do()
```

Common Perl Statements

- Assignment (use equal sign: “=”)
- Jumping
  - **`next/last/redo/goto`**
- Subroutine calls
  - Functional
  - Procedural
  - Object method
- Print statements

Assignments

- Use a single equal sign: “=”
- Put the value on the right into the place specified on the left



- Left-hand side of assignment called the “L-value”, most often a variable

Assignment Examples

```
$a = 2.75;          # scalar, floating point

$color = 'yellow';  # scalar, string

# array of four strings
@ary = ( "Perl", "C++", "Java", "FORTRAN" );

# hash, two keys (strings), numeric values
%ht = ( "AAPL" => 282.52, "MSFT" => 24.38 );
```

Assigning List Elements

```
@ary = ( "Perl", "C", "Java", "FORTRAN" );

$ary[1] = "C++";   # overwrite "C"

%ht = ( "AAPL" => 282.52, "MSFT" => 24.38 );

$ht{IBM} = 135.64; # add 1 item to hash
```

Operator-Assignment

```
$a += 10;          # add 10 to $a
$b *= 2.5;         # multiply $b by 2.5

$name .= ', Jr.';  # append to $name

$mode &= 0722; # apply bitwise mask to $mode
```

Scalar Values

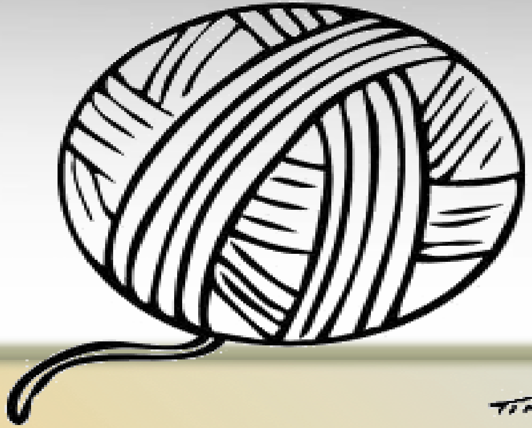
Scalar expression can be numeric or string, made from any of:

Variable	`$a`	`$ht{MSFT}`
Constant	`2.6`	`'blue'`
Function	`sin($angle)`	`length($name)`
Operators	`$a/sin($ang)+ 2.6`	`$col . "_" . $sz`

Numeric Constants

```
10       # decimal integer
0722     # octal integer
0xF3E9   # hexadecimal integer
-2.532   # floating point
6.022E23 # floating point (scientific)
```

String Constants



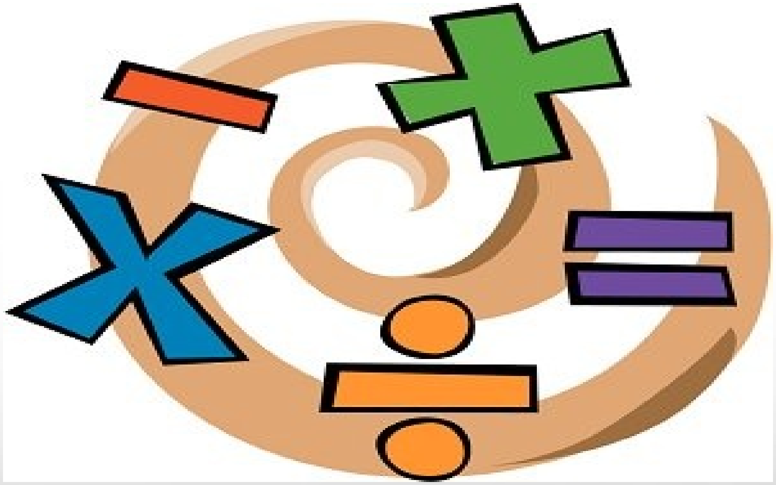
```
'simple'       # single quotes

"one\ttwo"     # double quotes

<<"END_TEXT";  # here document (dbl quotes)
1\t15kg\t3:25
2\t9kg\t0:22
END_TEXT
```

List Forms

```
("one", "two", 3) # list of mixed constants

qw(one two 3)     # same thing

# hash table
(
    "Mustang" => "Ford",
    "Civic"   => "Honda",
)
```

Operators

- Numeric operators are mostly the same as C/C++, also the "`**`" (exponent) operator
- Also has string and regular expression operators
- Documentation:
  - http://perldoc.perl.org/perlop.html
  - "**`man perlop`**"

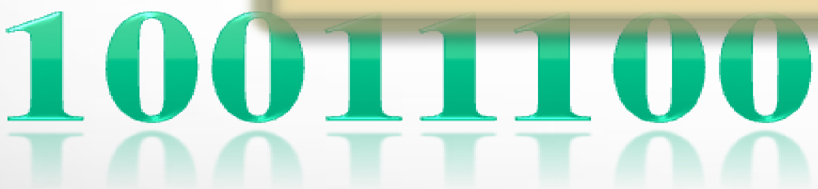


Numeric Operators

```
$x + 3      -4.3 / $z       2 ** 10

$i++        --$j            $f % $mod
```

Bitwise Operators

```
$mode << 8        $t | 0x3F

$v ^ $mask        ~$q
```



Comparison Operators

```
# numeric
$x == 3      $a >= -4.3       $m != $n

# stringwise
$y eq "AT"   $title lt 'm'    $q ne 'B'
```

Sort Comparison Operators

-1	Left is less than right
0	Left equals right
+1	Left is greater than right

```
# numeric          stringwise
$x <=> $y          $s cmp $t
```

Logical Operators

Logical Operators

```
# C-style
$ready && ($y > 2)    !$done     $e || $r

# lower precedence
$ready and $y > 2    not $done    $e or $r

# ternary conditional
($d != 0) ? ($n / $d) : "Inf"
```

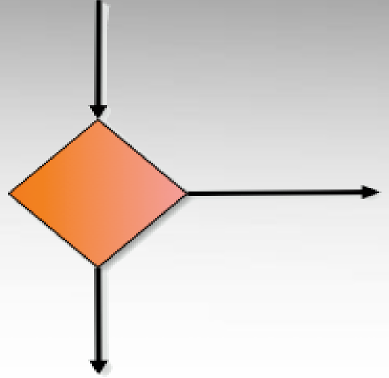
Other Operators

```
# List separators
2, 4, 6     "R" => 255, "G" => 0, "B" => 127

# string concatenation
"First " . $item

# range operator
1..10        0..$#ary
```

Comparison Operators

```
# numeric
$x == 3      $a >= -4.3       $m != $n

# stringwise
$y eq "AT"   $title lt 'm'    $q ne 'B'
```

Logical Operators



Logical Operators

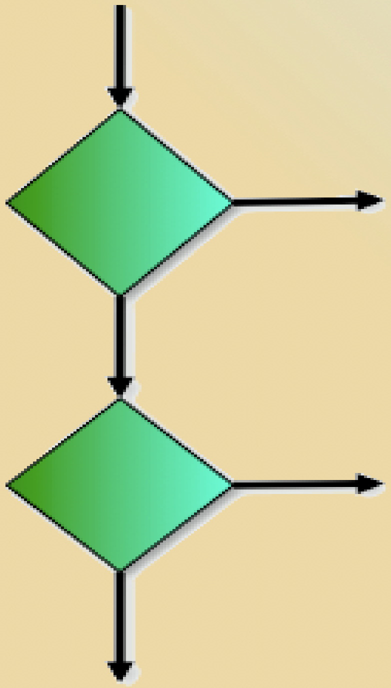
```
# C-style
$ready && ($y > 2)    !$done     $e || $r

# lower precedence
$ready and $y > 2    not $done    $e or $r

# ternary conditional
($d != 0) ? ($n / $d) : "Inf"
```

Other Operators

```
# List separators
2, 4, 6     "R" => 255, "G" => 0, "B" => 127

# string concatenation
"First " . $item

# string repetition
"AB" x 10

# range operator
1..10        0..$#ary
```

Conditional Branches

```
# choose the size of x
if ($x > 5) {
    print "big x\n";
} elsif ($x > 3) {
    print "medium x\n";
} else {
    print "small x\n";
}
```



Conditional After Statement

```
print "f is even\n" if ($f % 2 == 0);

print "not capitalized\n"
    unless ($name =~ /^[A-Z]/);
```

- Special feature of Perl
- Avoids need for braces
- Can be confusing and no “else” branch

Logical Operator as Conditional

```
($y != 0) &&           # C-style
    $ratio = $x / $y;

(-f $myfile) or        # word style
    die "File ``$myfile'' does not exist!";
```

- Avoids need for braces
- Can be confusing and no “else”
- Word style better rather than C-style

While Loops

```
$x = 1;               # initialize
while ($x != 7) {     # test
    print "x = $x\n";
    $x += 2;
    last if ($x > 12); # escape clause
}
```

For Loops

```
for ($x = 1; $x != 7; $x += 2) {
    print "x = $x\n";
    last if ($x > 12); # escape clause
}
```

Foreach Loops

```
foreach $item qw(PC Mac Linux) {
    printf "cost of %s is \$%1.2f\n",
        $item, $price{$item};
    print "TOO MUCH!\n"
        if ($price{$item} > 1200);
}
```

Loops and Labels

```
OUTER: while ($x != 8) {
    INNER: foreach $y (4, 3, 2, 1) {
        print "x = $x   y = $y\n";
        next INNER if ($y == 2);
        $x++ if ($x + $y == 5);
    }
    $x += 2;
    last OUTER if ($x > 12);
}
```

Errors and Warnings

- Warning is “non-fatal”, can still keep going
- Error can be at:
  - Compile time, e.g., syntax errors
  - Run time:
    - Numeric, e.g., division by zero
    - Reference type, e.g., hash vs. List
    - Object method

Warnings

Using -w option turns on warning messages

```
$ perl -w bounds.pl
Use of uninitialized value in addition (+) at bounds.pl line
8.
b = 3
```

Warnings Pragma

Put “**`use warnings`**” pragma at top to turn on warning messages.

```
use warnings;

my @a = (1, 2);
my $b = $a[0] + $a[1] + $a[2];

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

Uninitialized Value

```
my @a = (1, 2);
my $b = $a[0] + $a[1] + $a[2];
```

```
$ ./bounds.pl
Use of uninitialized value in addition (+) at ./bounds.pl
line 8.
```

“should be ==”

```
print "c is big\n" if ($c = 100);
```

```
$ ./twoeq.pl
Found = in conditional, should be == at ./twoeq.pl line 8.
```

Syntax Errors

```
while a < 10 {
```

```
$ ./synerr.pl
syntax error at ./synerr.pl line 4, near "while a "
syntax error at ./synerr.pl line 8, near "}"
Execution of ./synerr.pl aborted due to compilation errors.
```

Runaway Strings

```
$ ./closeq.pl
Scalar found where operator expected at ./closeq.pl line 8, near "print "$a"
  (Might be a runaway multi-line "" string starting on line 3)
        (Do you need to predeclare print?)
Backslash found where operator expected at ./closeq.pl line 8, near "$a\"
        (Missing operator before \?)
String found where operator expected at ./closeq.pl line 8, at end of line
        (Missing semicolon on previous line?)
syntax error at ./closeq.pl line 8, near "print "$a"
Can't find string terminator '"' anywhere before EOF at ./closeq.pl line 8.
```

Checking for Errors

Do your own error checking, to get diagnostics:

```
die "denominator zero " if ($d == 0);
$r = $n / $d;
```

- The “die” and “warn” functions output to stderr and can show line number
- The “Carp” module is more detailed

System Error String

If a system call fails, look at “**$!**” variable

```
open FH, $myfile, "r" or
    die "open $myfile: $! ";
```

```
$ ./nofile.pl
open nofile: No such file or directory  at
./nofile.pl line 4.
```

Perl Debugger

You can use **`perl -d`** to debug your program:

```
$ perl –d debugme.pl
$Loading DB routines from perl5db.pl version 1.28

Enter h or `h h' for help, or `man perldebug' for more help.

main::(debugme.pl:7):   my @a = qw(TAGC CGTA ATTT GGCA);
DB<1>
```

Variable Scope

- Using the “**`my`**” declaration makes a variable local to the statement block or file.
- Don’t use “**`local`**” declaration unless you understand it—the “**`my`**” declaration is almost always what you want.
- The “**`our`**” declaration is used for declaring global variables within packages (modules).

Examples of Local Variables

- Surround multiple declared variables with parentheses.

```
my $a;
my @f;
my $x = "initial value";
my ($i, $j, $k);
foreach my $item (@ilist) {
    $sum += $item;
}
```

Example of Scope

```
my @numlist = (3, 4, 5);

foreach my $item (@numlist) {
    print "item = $item\n";
}
print "item = $item\n”;
```

```
item = 3
item = 4
item = 5
item =
```

Strict Pragma

Put “**`use strict`**” pragma at top to require use of “**`my`**” declarations.

```
use strict;

my $x = 15;
$y = 19;

print ”y = $y\n";
```

Input/Output

Input
Program
Output

File Handles

- **`STDIN`**, **`STDOUT`**, and **`STDERR`** correspond to the **`stdin`**, **`stdout`**, and **`stderr`** of C/C++.
- A simple Perl filehandle is a name by itself, typically all caps.
- Filehandles can be scalar variables, too.
- Objects from **`IO::File`** and similar library modules have advantages.

print
printf
syswrite

Printing

- **`print`**
  - Prints a list of strings
- **`printf`**
  - C-style formatting
- **`syswrite`**
  - Low-level **`write(2)`** system call
  - Unbuffered and unformatted.

print
printf
syswrite

File Handle in Printing

- **print** ***FH LIST***
- **printf** ***FH FORMAT, LIST***
- **syswrite** ***FH, DATA,*** **...**

For **print** and **printf** there is no comma separating file handle from arguments. Without a file handle, the output goes to **STDOUT**, by default.

Print Examples

```
print "Hello, world!\n";
print STDOUT "Hello, world!\n"; # same

print STDERR "File not found:", $fname,
   "\n";

printf MYRPT "%d items processed\n",
    $count;
printf MYRPT ("%d items processed\n",
    $count);                 # same
```

< >

sysread

Reading

- **`< >`**
  - Input operator for buffered input.
  - Uses **`STDIN`**, by default.
- **`<` *`FH`* `>`**
  - Input from filehandle ***`FH`***.
- **`sysread`**
  - Low-level **`read(2)`** system call
  - Unbuffered and unformatted.

Input Examples

```
print ”Enter name:";
$name = <>;

@listing = <STDIN>; # read all lines

# one line at a time
while ($line = <$myinfo>) {
    myprocess($line);
}
```

Using __DATA__

```
while ($line = <DATA>) {
    print $line;
}

__DATA__
anything here read from DATA filehandle
another line
last line
```

Opening a File

- The **open** function opens a file for reading, writing, appending, or more.
- Can specify a bareword file handle name or a scalar variable.

```
open(MYINFO, "<info.dat") or
    die("open info.dat: $! ");

open $fh, ">logfile" or die $!;
```

File Mode

Read	<	+<
Create/truncate	>	+>
Append	>>	+>>

```
open(MYINFO, ”+<info.dat") or
    die("open info.dat: $! ");

open $fh, $fname, ">>" or die $!;
```

Putting it Together

```
open RAW, $rfile, "<";
open $ofh, ">>results";

while ($line = <RAW>) {
    @useful = myprocess($line);
    printf $ofh "%d,%6.2f,%s\n", @useful;
}
```

The **$_** Variable

- **$_** acts as the default scalar for input, output, patterns, and many Perl functions
- The read operator ("**< >**") puts input into **$_** if no variable (L-value) specified
- Pattern matching defaults to using **$_**
- Various functions use **$_** by default if no argument specified.

File Input Using **$_**

```
open $motd, "</etc/motd" or die $!;

while (<$motd>) {
    next unless /ada/i; # only matching lines
    tr/A-Z/a-z/;        # replace uppercase
    print;
}
```

File Input Using **$_**

```
open $motd, "</etc/motd" or die $!;

while (defined($_ = <$motd>)) {
    next unless $_ =~ /ada/i;
    $_ =~ tr/A-Z/a-z/;
    print $_;
}
```

Redirecting Filehandles

- Can reassign input, output, and error filehandles to files or command pipes
- Following statements which use default filehandles will then use these files

Redirected

```
open STDERR, "error.report" or die $!;
# sent to error.report
warn "Something went wrong!\n";

open STDIN, "<input.data" or die $!;
$line = < >; # reads from input.data
```

Command Pipelining

- Use “**open**” to run a command and redirect input or output to a filehandle
- Use the “|” symbol before a command if you want to provide its input through the filehandle
- Use the “|” symbol after a command if you want to capture output from it

Pipelining

```
open SORTER, "|sort –t: -k3 –k1" or die $!;
print SORTER @lines;

open PROCS, "ps aux | cut –c1-15,65-80 |"
    or die $!;
while (<PROCS>) {
    print if (/^kjacks/);
}
```

Command Substitution

- A shorthand way to run a command an capture the output is with the backwards single quotes, or “**qx**”:

```
$hostname = `/bin/hostname`;
$hostname = qx{/bin/hostname};
```

Same thing

Binary Files

- Always use “**`sysopen`**” with “**`sysread`**” and “**`syswrite`**”
- Don’t mix “**`sys*`**” with buffered I/O functions (“**`< >`**”, “**`print`**”, etc.)
- Use “**`pack`**” and “**`unpack`**” to convert byte data to scalar variables

Read the “**`pack`**” Tutorial

- http://perldoc.perl.org/perlpacktut.html
- Or: “**`man packtut`**”
- Example: “Packing and Unpacking C Structures”
- Need to copy the “**`#define Pt()`**” macro to C stub file for determining Perl format string

C struct

```
typedef struct {
    char       fc1; // pos 0 (& 1 byte pad)
    short      fs;  // pos 2
    char       fc2; // pos 4 (& 3 byte pad)
    long       fl;  // pos 8
    float      ff;  // pos 16
} gappy_t;
gappy_t      info; // ...
write(fh, (char *) &info, sizeof(info));
```

Print Format String

```
#define Pt ...
Pt(gappy_t, fc1, c  );
Pt(gappy_t, fs, s!  );
Pt(gappy_t, fc2, c  );
Pt(gappy_t, fl, l!  );
Pt(gappy_t, ff, f   );
printf("total = %d\n", sizeof(gappy_t));
```

```
@0c @2s! @4c @8l! @16f
total = 24
```

Read Data in Perl

```
my $packf = '@0c @2s! @4c @8l! @16f';
my $sz = 24; # C struct is 24 bytes
sysopen my $fh, $fname, O_RDONLY or die $!;
my ($data, $count, @fields);
while ($count = sysread($fh, $data, 24)) {
    @fields = unpack($packf, $data);
    ...
}
```

Regular Expressions

- Regular expressions are patterns designed to concisely match a set of strings which follow the rules of the given pattern
- Regular expressions have a long history in Unix (**`ed`**, **`grep`**, **`vi`**, **`awk`**)
- Perl extends the traditional regular expressions, adding new rules

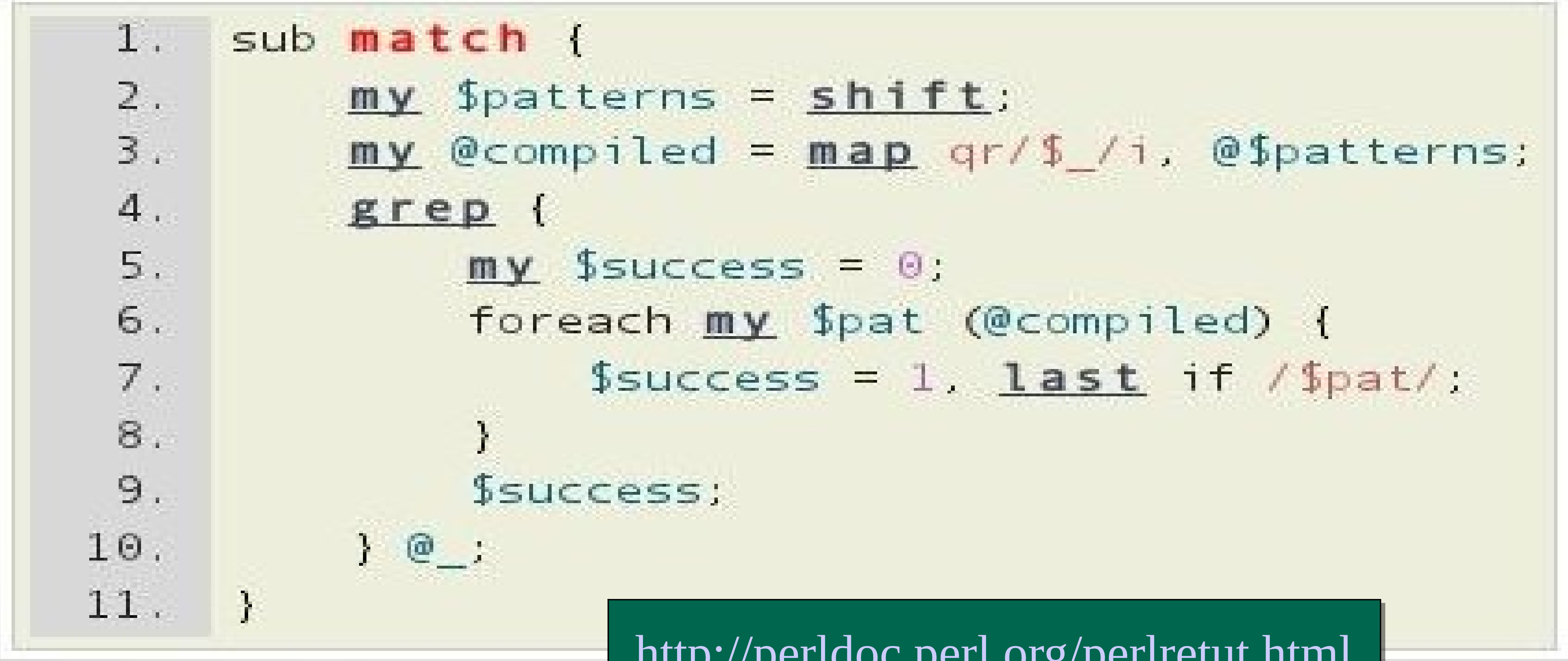
Quick Examples

```
$name =~ /Mich/; # Michael, Michelle, ...

$shell =~ /^[abck]sh/; # csh, ksh (not bash)

$fname !~ /.*\.[ch] $/; # not a source

$command =~ s?^?/usr/bin?; # prepend dir

$dosfile =~ tr/A-Z/a-z/;   # case
```

Main Regexp Operators

Operator	**Use (return)**
qr/*pattern*/	precompile pattern (regexp)
/*pattern*/ m{*pattern*}	match a pattern (success status)
s/*pattern*/*replacement*/ s{*pat*}{*repl*}	substitute (count of replacements)
tr/*set1*/*set2*/ y|*set1*|*set2*|	transliterate (count of replaced characters)

Precompiled Pattern



```
sub match {
    my $patterns = shift;
    my @compiled = map qr/$_/i, @$patterns;
    grep {
        my $success = 0;
        foreach my $pat (@compiled) {
            $success = 1, last if /$pat/;
        }
        $success;
    } @_;
}
```

http://perldoc.perl.org/perlretut.html

split and **grep**

- **split** function divides a string using regexp to indicate separator pattern

```
split(/[:,]/, 'a:fg:x:::2,2:3 KB');
```

- **grep** can use a regexp to test against a list

```
grep /^A.*s$/, qw(Adams Aaron Avons arts);
```

Metacharacters

<table>
<tr><td>\</td><td>Quote the next metacharacter</td></tr>
<tr><td>^</td><td>Match start of line</td></tr>
<tr><td>.</td><td>Match any one character</td></tr>
<tr><td>$</td><td>Match end of line</td></tr>
<tr><td>|</td><td>Alternation</td></tr>
<tr><td>( )</td><td>Grouping</td></tr>
<tr><td>[ ]</td><td>Character class</td></tr>
</table>

http://perldoc.perl.org/perlre.html

Quantifiers

`*`	0 or more times
`+`	1 or more times
`?`	1 or 0 times
`{n}`	Exactly n times
`{n,}`	At least n times
`{n,m}`	At least n but not more than m times

- Add a “**?**” after quantifier to make it not “greedy”, a “**+**” to force “greediness”

http://perldoc.perl.org/perlre.html

Escape Sequences

`\t \n \033`	C-style control characters
`\l`	lowercase next character
`\u`	uppercase next character
`\L`	lowercase until **`\E`**
`\U`	uppercase until **`\E`**
`\E`	end case modification
`\Q`	quote (disable) metacharacters until **`\E`**

http://perldoc.perl.org/perlre.html

Character Classes

`\w`	“Word” character: `[a-zA-Z0-9_]`
`\W`	Non-“word” character: `[^a-zA-Z0-9_]`
`\s`	Whitespace character
`\S`	Non-whitespace character
`\d`	Digit character: `[0-9]`
`\D`	Non-digit character: `[^0-9]`
`\1 \2 \3`	Back references to groupings with “( )”

http://perldoc.perl.org/perlre.html

Capture Buffers

- **( )** grouping is saved in buffers
  \1, **\2**, …, or **$1**, **$2**, …

```
$line =~ /^(\w+) (\d+)\s*(\w+)?$/;
$name = $1; $count = $2; $optlabel = $3;

$fullname =~ s/^(\w+), (\w+)$/\2 \1/?;

($fn, $ln) = ($N =~ /^(\w+) (?:\w+) (\w+)$/);
```

Process Counting Part 1

```
my $pwtbl = new IO::File "</etc/passwd";
open my $pscom, "ps auxww | tail -n +2 |”;

my %realname = ();
while (<$pwtbl>) {
    @fields = split(/:/);
    next unless ($fields[2] > 1000);
    $realname{$fields[0]} = $fields[4];
}
```

Process Counting Part 2

```
my %numprocs = ();
while (<$pscom>) {
   my ($login) = (m/^(\w+)/);
   next unless (exists $realname{$login});
   $numprocs{$login} = 0
        unless (exists $numprocs{$login});
   $numprocs{$login}++;
}
```

Process Counting Part 3

```
foreach my $login (sort keys %numprocs) {
    printf("%4d procs for %9s (%s)\n”,
        $numprocs{$login}, $login,
        $realname{$login});
}
```

References

Fall 2016

References

- Perl references are scalar values which contain a pointer to:
  - another scalar
  - an array
  - a hash table
  - a subroutine
  - typeglobs

Examples of an Array Reference

```
@a = (9, 8, 3, 6);
$aref = \@a;

@r = reverse @{$aref};
@s = sort @$aref;

$third = ${$aref}[2];
$num3  = $aref->[2];
```

same as:

```
@r = reverse @a;
@s = sort @a;

$third = $a[2];
$num3  = $a[2];
```

Making a Reference

- Perl references are created by:
  1. a backslash (“\”) before a variable or subroutine, or
  2. an assignment to an “anonymous” list, hash, or code block.

References to Variables

```
$sc_ref = \$number;   # scalar
$ar_ref = \@namelist; # array
$hs_ref = \%lookup;   # hash
$sb_ref = \&mysub;    # subroutine
```

References to Anonymous

```
$ar_ref = [ 4, 3, 3, 7 ];        # array

$hs_ref = { m => 6, n => 9 };    # hash

$sb_ref = sub { return(shift(@_) + 1) };
# subroutine
```

Using a Reference

- Dereference by:
  1. using type symbol (“**$**”, “**@**”, “**%**”, “**&**”) then the reference variable in curly braces (“**{ }**”), or
  2. access an element by inserting “**->**” between reference variable and the element specifier, i.e., “**[ ]**” for arrays and “**{ }**” for hashes. For subroutines, the argument list in parentheses follows the “**->**” .

Bracing References

```
$sc_ref = \$number;   # scalar

printf("%d\n", ${$sc_ref});
printf("%d\n", $number);
```

Same thing

```
$ar_ref = \@namelist; # array

push(@{$ar_ref}, "Harvey");
push(@namelist, "Harvey");
```

Same thing

Bracing References

```
$hs_ref = \%lookup;   # hash

@logins = keys %{$hs_ref};
@logins = keys %lookup;

$sb_ref = \&mysub;    # subroutine

$rc = &{$sb_ref}($arg1, $arg2);
$rc = mysub($arg1, $arg2);
```

Same thing

Same thing

Leaving Off the Braces

- You don’t always have to surround the reference variable with braces, as long as doing so doesn’t create ambiguity.

```
@{$ar_ref}        @$ar_ref

%{$hs_ref}        %$hs_ref
```

Bracing for Subelements

```
$ar_ref = \@namelist; # array

$fourth = ${$ar_ref}[3];
$fourth = $namelist[3];                          Same thing

foreach $i (0..$#{$ar_ref}) …
foreach $i (0..$#namelist) …                     Same thing
```

Bracing for Subelements

```
$hs_ref = \%lookup; # hash

$myid = ${$hs_ref}{$login};
$myid = $lookup{$login};
```

Same thing

Arrow Shorthand

```
$ar_ref = \@namelist; # array

$fourth = ${$ar_ref}[3];
$fourth = $ar_ref->[3];               Same thing
$fourth = $namelist[3];               Same thing

$oops   = $ar_ref[3];                 DIFFERENT!
```

Arrow Shorthand

```
$hs_ref = \%lookup; # hash

$myid = ${$hs_ref}{$login};
$myid = $hs_ref->{$login};
$myid = $lookup{$login};

$wrong = $hs_ref{$login};
```

Same thing

Same thing

DIFFERENT!

Arrow Shorthand

```
$sb_ref = \&mysub; # subroutine

$rc = ${$sb_ref}($arg1, $arg2);
$rc = $sb_ref->($arg1, $arg2);              Same thing
$rc = mysub($arg1, $arg2);                  Same thing

$wrong = $sb_ref($arg1, $arg2);             SYNTAX ERROR
$wrong = sb_ref($arg1, $arg2);              different
```

Breaking the 1-Dimension Barrier

- For a 2-dimensional array, create a list of references to individual lists, one per row:

```
@table =
(
   [  2, -1,  3 ],
   [  0, 10, -9 ],
   [ 18,  3,  4 ],
);
```

```
$x = ${$table[1]}[2];
$x = $table[1]->[2];
$x = $table[1][2];
```

All are -9

Complex Data Structures

- You can nest arrays and hashes to accomplish multiple dimensions:

```
@info = (
  [ 3, "red", { Bldg => 'CSA', Floor => 1 } ],
  [ 7, ”blue", { Bldg => 'Bright', Hrs => [ 8, 5 ] } ],
);

$start = $info[1][2]{Hrs}[0];
$flr = $info[0][2]{Floor};

$intermed = $info[1][2];     # use extra vars to simplify
$start = $intermed->{Hrs}[0];
```

When the Arrow is Required

- Leaving out the arrow only works between indices/keys:

```
@info = (
  [ 3, "red", { Bldg => 'CSA', Floor => 1 } ],
  [ 7, ”blue", { Bldg => 'Bright', Hrs => [ 8, 5 ] } ],
);

$inf_ptr = \@info;

$start = $info[1][2]{Hrs}[0];
$end   = $inf_ptr->[1][2]{Hrs}[1];  # arrow required
```

Perl Subroutines

- Perl subroutines declared with “**`sub`**”
- The subroutine name follows rules of variable names
- Can leave off name to make an “anonymous” routine
- Can prototype, calls are type checked
- Sub returns a scalar or a list

Subroutine Arguments

- Arguments are a scalar list
- Arguments are not named in declaration (formal parameters) or prototype, but are put in the “`@_`” variable
- Contents of `@_` are call by reference
- Use **`shift(@_)`** and **`my`** to make local copies

A Sample Subroutine

```
sub add2array {
    my $val = shift @_;
    my @newary = @_;
    foreach my $idx (0..$#newary) {
        $newary[$idx] += $val;
    }
    $val = -1;
    return @newary;
}

$x = 5;
@orig = (1, 2, 4, 7);
@incr = add2array($x, @orig);
print "x = $x\norig = (@orig)\nincr = (@incr)\n";
```

```
x = 5
@orig = (1 2 4 7)
@incr = (6 7 9 12)
```

Modifying Original

- What not to do—a cautionary tale:

```
sub add2array {
    my $val = $_[0];
    foreach my $idx (1..$#_) {
        $_[$idx] += $val;
    }
    $val = -1;
    $_[0] = 999;
    return @_[1..$#_];
}

$x = 5;
@orig = (1, 2, 4, 7);
@incr = add2array($x, @orig);
print "x = $x\norig = (@orig)\nincr = (@incr)\n";
```

```
x = 999
@orig = (6 7 9 12)
@incr = (6 7 9 12)
```

Prototypes

- After **`sub`** ***`sname`***, put list of type characters in parentheses
- Backslash before symbol turns parameter into reference
- Semicolon separates mandatory from optional parameters
- Except references, only one argument (last one) can be list or hash

Sample Prototypes

```
sub myindex($$;$)

sub myjoin($@)
sub mypop(\@)
sub mysplice(\@$$@)

sub mygrep(&@)
```

```
myindex $c, $str
myindex $c, $str, $pos

myjoin ':', @items
mypop @stack
mysplice @ary, 0, 3
mysplice @ary, 0, 3, (2, 4)
mygrep { /pat/ } @lines
```

Prototype References

- When backslash used to indicate reference parameter, actual parameter in the call is the original type, but argument in the “`@_`” list is reference to the original
- You can always pass an actual reference, designated as scalar (“**$**”) in the prototype

Reference Parameters

```
sub mypop1(\@) {
    my $ary_ref = shift;
    my $retval;
    if ($#{$ary_ref} >= 0) {
        $retval = $ary_ref->[$#{$ary_ref}];
        $#{$ary_ref}--
    } else {
        $retval = undef;
    }
    return $retval;
}

$x = mypop1(@items);
```

Reference Parameters, Take 2

```
sub mypop2($) {
    my $ary_ref = shift;
    my $retval;
    if ($#{$ary_ref} >= 0) {
        $retval = $ary_ref->[$#{$ary_ref}];
        $#{$ary_ref}--
    } else {
        $retval = undef;
    }
    return $retval;
}

$x = mypop1(\@items);
```

Perl Functions

Perl Functions

- Perl has dozens of built-in functions
- Read descriptions at perlfunc man page, or on-line at:

http://perldoc.perl.org/index-functions.html

Queues and Stacks

- Use **push** and **shift** to implement FIFO queue
- Use **push** and **pop** to implement LIFO stack

```
while ($item = get_request()) {
    push(@mylist, $item);
}
# get first item from front of queue
while ($req = shift(@list)) {
    answer_request($req);
}
```

```
# get most recently added item from top of stack
while ($req = pop(@list)) {
    answer_request($req);
}
```

Sorting

- Use **`sort`** to order a list
- Specify your own code block to customize

```
@stlist = sort @namelist;

@ilist = sort { $a <=> $b } @numberlist;

@loginbyuid = sort
  { $userlist{$a}{UnixId} <=> $userlist{$b}{UnixId} }
  keys %userlist;
```

Sorting With a Subroutine

- Define a sub with **$a** and **$b**

```
sub bydatesizename {
   $mtime{$a} <=> $mtime{$b}
 or $filesize{$a} <=> $filesize{$b}
 or $filename{$a} cmp $filename{$b}
}

@sortfiles = sort bydatesizename @flist;

@mysorted = sort bydatesizename
  grep { $owner{$_} eq $USER } @flist;
```

Splitting and Joining

- Use a regexp for **`split`**
- Use a string for **`join`**

```
$line = "one:two:three:four";
@parts = split /:/, $line;

foreach (@parts) { s/^(.)/\u\1/ }

$newline = join(", ", @parts);

print "newline = `$newline'\n”;
```

```
newline = `One, Two, Three, Four'
```

Map

```
@surround = map { '"' . $_ . '"' } @words;

foreach $idx (0..$#words) {
    $surround[$idx] = '"' . $words[$idx] . '"';
}

######

map { send_email($_) } @recipients;
```

Same thing

Chomping the Input

- **chomp** removes newlines from end of input line

```
while (chomp(my $line = <STDIN>)){
   dosomething($line);
}

chomp(@lines = <$fh>);
myprocess(@lines);

$cwd = chomp(`pwd`);
```

Check Files

- shell test flags can check info on file

```
dosomething($myfile) if (-f $myfile);

print "cannot exec" unless (-x $myfile);
```

Perl Objects

Fall 2016

Modules

- Perl modules are external files containing packages and symbol tables (different namespace, e.g., variable scope)
- Modules effectively implement libraries and are often done in an object-oriented fashion
- To use a given module, read its man page first for instructions

IO::File and Fcntl

```perl
use IO::File;

my $fh = new IO::File $fname, "<" or die $!;
$input = <$fh>;
$fh->close;

use Fcntl; # get O_ constants
my $ofh = IO::File->new($outname,
         O_CREAT|O_WRONLY|O_EXCL);
print $ofh @data;
$ofh->close;
```

File::stat

```
use File::stat;
use Fcntl qw(:mode); # get S_I macros

$st = stat($myfile) or die $!;
next if (S_ISLNK($st->mode)); # skip if symbolic link

print "can read\n" if ($st->cando(S_IRUSR, 1));
```

Getopt::Std

```
use Getopt::Std;

my %opts = ();
getopts('of:v', \%opts) or die("invalid options");

$fname = $opts{f} or $fname = 'default';

print "verbosity!\n" if ($opts{v});
```

```
$ ./myprog.pl –v –f altfile
```