Lecture #19 – Perl Part III

- Subroutines

Defined with keyword “sub”, and referenced like function calls in other languages.

Example:

```
sub razzle
{
        print “You have been razzled.\n”;
}

razzle();
```

Parameters (Arguments) to subroutines are treated as a flat list of scalars, and are stored in the @_ array.

In other words, the first argument to the subroutine could be referenced as $_[0], and the second as $_[1], etc.

Example:

```
sub myenv
{
        my ($key, $value) = @_;

        $ENV{$key} = $value unless $ENV{$key};
}
```

Example:

```
sub max
{
        my $max = shift(@_);

        for my $item (@_)
        {
                $max = $item if $max < $item;
        }

        return $max;
}

$best = max($mon, $tue, $wed, $thu, $fri, $sat, $sun);
$best_weekend = max($sat, $sun);
```

Note: All arguments to a subroutine are passed as one long list of scalars. When passing multiple arrays, it maybe useful to pass references instead.

Example:

```
@c = (1, 2, 3);
@d = (1, 2, 3, 4);

($aref, $bref) = func(\@c, \@d);
print "@$aref has more than @$bref\n";

sub func
{
        my ($cref, $dref) = @_;

        if (@$cref > @$dref)
        {
                return ($cref, $dref);
        }
        else
        {
                return ($dref, $cref);
        }
}
```

- Packages and Modules

Modules are written to accomplish tasks not implemented by Perl's built-in functions.

The module is the fundamental unit of code reuse in Perl. Under the hood, it's just a package defined in a file of the same name (with .pm on the end).

Perl comes with many modules on installation. Many more can be found at the Comprehensive Perl Archive Network (CPAN). See http://www.cpan.org.

Modules come in two flavors – traditional and object oriented. Traditional modules define subroutines and variables for the caller to import and use. Object-oriented modules function as class definitions and are accessed through method calls.

Modules are included in your program with the "use" or "require" statement:

```
use MODULE;                (Example:  use CGI;)
require MODULE;
```

"use" works at compile time, and "require" at run-time.

- The CGI Module

  CGI (Common Gateway Interface) programs provide one mechanism for running code from a web page.

  Specifically, CGI programs obtain input parameters via the web browser interface (via GET or POST methods), and these programs return HTML code as their output.

  Perl is one of the more popular ways to implement CGI programs primarily because of the CGI module available for Perl.

  Example (HTML):

```
<HTML>
<HEAD>
<TITLE>Are you old enough to vote?</TITLE>
</HEAD>
<BODY>
<H1>Are you old enough to vote?</H1>
<P>
<FORM ACTION=”voter.cgi” METHOD=GET>
Age: <INPUT TYPE=”text” NAME=”age”>
</FORM>
</P>
</BODY>
</HTML>
```

  Example (CGI):

```
use CGI;
$query = new CGI;

#  Obtain the “age” data from the web interface

if ($query->param(“age”) >= 18)
{
        $voter = “yes”;
}
else
{
        $voter = “no”;
}
```

```
# Print output in HTML

print $query->header;
print "<HTML><HEAD>\n";
print "<TITLE>Are you old enough to vote?</TITLE>\n";
print "</HEAD><BODY>\n";
print "<H1>Are you old enough to vote?</H1>\n";

if ($voter eq 'yes')
{
        print "<P>You are old enough!</P>\n";
}
else
{
        print "<P> You are not old enough yet!\n";
}

print "</BODY></HTML>\n";
```

- Misc. Examples

Example: (wc)

```
while (<STDIN>)
{
    $line++;
    $char += length;
    $word += split;
}

print "$line $word $char\n";
```

Example: (wc_improved)

```
sub count
{
    my ($fh, $name) = @_;

    my $line = 0;
    my $char = 0;
    my $word = 0;

    while (<$fh>)
    {
        $line++;
        $char += length;
```

```
                $word += split;
        }

        print "$line $word $char\t$name\n";
}

if ($#ARGV < 0)
{
        count(*STDIN, "STDIN");
}
else
{
        foreach (@ARGV)
        {
                open(F, $_) || die "cannot open $_\n";
                count(*F, $_);
                close(F);
        }
}
```

Example: (lower)

```
$count = 0;
$total = 0;

foreach $file (`ls`)
{
        #  convert filename to lowercase
        chomp($file);
        $lower = $file;
        $lower =~ tr/A-Z/a-z/;

        #  skip this file if its already lowercase
        next if $file eq $lower;

        #  make sure not to overwrite another file by accident
        next if -e $lower && print "cannot rename $file: already exists\n";

        #  rename the file, and track the count
        $status = system("mv $file $lower");
        $count++;
}
continue
{
        $total++
}
```

```
# Report the stats
print "$count of $total files successfully renamed\n";
```

Example: (upper)

```
use strict;

my $fname = "input.txt";

open(F, $fname) || die "Cannot open file $fname\n";
open(OUT, ">upcase.txt") || die "Cannot open output file.\n";

while (<F>)
{
    tr/a-z/A-Z/;

    print OUT $_;
    print $_;
}

close(F);
close(OUT);
```