Programming in Perl

Week Three

Simple I/O and text processing

Functions

Exercise 2.1

```
#!/usr/bin/perl -w
use strict;

print "Enter a weight in pounds: ";
my $pounds <STDIN>;
chomp($pounds);
my $kilos = $pounds / 2.2;
print "$pounds pounds equals $kilos kilograms\n";
```

Exercise 2.2

```
#!/usr/bin/perl -w
use strict;

print "Enter your hourly pay rate: ";
my $rate = <STDIN>
chomp($rate);

print "Enter the number of regular hours: ";
my $reg_hours = <STDIN>;
chomp($reg_hours);

print "Enter the number of overtime hours: ";
my $over_hours = <STDIN>;
chomp($over_hours);

my $reg_pay   = $rate * $reg_hours;
my $over_pay  = $rate * $over_hours * 1.5;
my $gross_pay = $reg_pay + $over_pay;
print "Gross pay is: $gross_pay\n";
```

Exercise 2.3

```
#!/usr/bin/perl -w
use strict;

my @values;
foreach my $count ( 1 .. 20 ) {
   print "Enter value $count [or 'q' to stop]: ";
   chomp(my $value = <STDIN>);
   last if $value eq 'q';
   unshift(@values, $value);
}

print "Reversed values are:\n";
foreach my $value (@values) {
   print "$value \n";
}
```

Exercise 2.4

```
#!/usr/bin/perl -w
use strict;

my @values;
my %seen;
foreach my $count ( 1 .. 20 ) {
   print "Enter value $count [or 'q' to stop]: ";
   chomp(my $value = <STDIN>);
   last if $value eq 'q';
   next if $seen{$value}++;
   unshift(@values, $value);
}

print "Reversed values are:\n";
foreach my $value (@values) {
   print "$value : used $seen{$value} time(s)\n";
}
```

What is a file handle?

- A File Handle is what you use to pick up a file!
- It's name assigned to a Input/Output channel
- Some predefined names: `STDIN`, `STDOUT`, `STDERR`, `DATA`, `ARGV`, `ARGVOUT`
- Unlike other “data types” in Perl, file handles do not have a prefix character
  - They can be confused with operators and functions
  - It’s recommended that you name them all in uppercase

Opening and Closing a File Handle

- You associate a file handle with an external I/O channel (i.e.: “open” a file handle to a file) with `open()`

```
open SOMEHANDLE, "foo";    # opens the file "foo" as SOMEHANDLE
open(LOG, ">>bar");        # opens "bar" as LOG, appending to the
                           # end of the file
open NEW, ">fred";         # opens "fred" as NEW, writing to it as
                           # if it was a new file (overwriting)
open SOMEHANDLE, "<foo";   # Same as the first example
```

- You can remove the association with `close()`

```
close SOMEHANDLE;
```

- Perl will automatically close any open file handles when the script exits
  - You rarely see `close()` operators in scripts because of this

A Slight Diversion: `die()` and `warn()`

- Both `open()` and `close()` return a true/false value- true if the operation was successful
- It's not very wise to continue the script if a file that is needed for the algorithm is not found or can't be created
- `die()` prints a message (to the standard error channel) and exits the script without completing the reset

```
unless (open(LOG, ">>logfile")) {
   # We can't open the log file, let's get outta here
   die "Cannot open or create logfile";
}
```

A Slight Diversion: `die()` and `warn()`

- There is a more idiomatically "correct" way to express this in Perl (and easier to type):

```
open LOG, ">>logfile" || die "Cannot open or create logfile";
open LOG, ">>logfile" or die "Cannot open or create logfile";
```

- Read this as "open the logfile for append… or die!"
- If the message doesn't end in a newline ("`\n`"), the script name and line number of the `die()` are append to the message automatically
- `warn()` is like `die()` but doesn't exit the script

Using File Handles

- Any file handle can be read using the diamond operator, like `<STDIN>`

```
open FILE, "filename" or die "Can't open file 'filename'\n";
while (<FILE>) {
  print;
}
```

- The code above uses some of Perl's shortcuts

```
open FILE, "filename" or die "Can't open file 'filename'\n";
while (defined($_ = <FILE>)) {
  print $_;
}
```

- `$_` is a special, predefined, variable. Perl has lots of them, see `perlvar` man page for details

The @ARGV Array

- Another predefined variable is the @ARGV array. It holds any arguments to that were given on the command line
- The Diamond operator, <>, treats each argument as a file name, opens each in turn and reads the contents

```
#!/usr/bin/perl -w
use strict;
while (<>) {
  print;
}
```

- If no files are given on the command line, <> acts the same as <STDIN>

Pattern Matching

- One of the most powerful features of Perl is its Pattern Matching and Substation operations
- Both use a Regular Expression "pattern". The pattern is compared to a string to see if it matches the string
- The pattern match operator has the syntax

```
m/PATTERN/
```

- Other delimiters (than "`//`") may be use, as long as they are the same, `m?PATTERN?`, or paired, `m{PATTERN}`
- If the default delimiters ("`//`") are use, you can drop the "m"

```
/PATTERN/
```

Regular Expressions

- Regular Expressions (RE's or Regex's) are a template that defines a collection of possible strings
- A string either matches, and returns true, or doesn’t match a regular expression

```
#!/usr/bin/perl -w
use strict;
while (<>) {
  if (/foo/) {
   print;
  }
}
```

Simple uses of regular expressions

- By default, the match operator (`m//`) matches a regular expression against the contents of `$_`, returning true or false
- To match a scalar variable use the `=~` operator

```
#!/usr/bin/perl -w
use strict;
my $line;
foreach my $file (@ARGV) {
  open(FILE, $file) || die "can't open file: $!";
  while (defined($line = <FILE>) {
   if ($line =~ m/foo/) {
          print $line;
   }
  }
  close FILE;
}
```

Substitutions

- Allow the replacement of whatever part of a string matches a regular expression with another string
- Simplest substitution: `s/old/new/`

```
$_ = 'hello world';
s/hello/goodbye, cruel/;  # $_ = 'goodbye, cruel world'
s/cruel/not so $&/;       # $_ = 'goodbye, not so cruel world'
s/not.*(e)(l)/P$1r$2/;    # $_ = 'goodbye, Perl world'
```

- To make all possible non-overlapping replacements, append a “`g`” modifier after the last delimiter

```
$_ = 'He smiled at the miles!';
s/mile/corn/g;            # $_ is now 'He scorned at the corns!'
```

The `split()` operator

- The `split()` operator sakes a single string and breaks it up into a list according to a regular expression

```
@some_list = split(/regexp/, $string);
```

- Anything that matches is a delimiter— everything between delimiters is a field that will be returned

```
$some_input = "This is a test.\n";
@args = split(/\s+/, $some_input);
# @args is now ('This','is','a','test.')
```

- Trailing empty fields are discarded— leading empty fields are kept

```
@a = split(/:/,':::a:b:c:::'); # @a is ('','','','a','b','c')
Defaults to the common case (splitting $_ on whitespace)
@some_input = split; # same as split(/\s+/,$_);
```

The `join()` operator

- The `join()` operator does the inverse of a split— makes a single string out of a bunch of pieces by joining them together with some "glue"

```
$some_string = join($glue, @some_values)
```

- The glue can be any string; even the empty string
- The glue is inserted between elements, not after elements

```
$x = join(':;,4,6,8,10,12); # $x = '4:6:8:10:12'
$y = join('foo','bar'); # $y = 'bar'
@values = split(/:/,$x); # @values = (4,6,8,10,12)
$z = join('+',@values); # $z = '4+6+8+10+12'
```

The `DATA` file handle

- The `DATA` file handle allows you to get at data in the script file
- Reads from the `DATA` file handle will contain any text that comes after a `__END__` or `__DATA__` line in your script

```
#!/usr/bin/perl -w
use strict;
while (<DATA>) {
  print;
}
__END__
foo
bar
baz
```

User Functions (Subroutines)

- Subroutines allow code to be reused in multiple places in the program
- Subroutines are named like everything else— a subroutine name can be prefixed by ampersand (&). It is not necessary to use the prefix in Perl5.000 and later
- Subroutine names are in yet another namespace (`$foo` has nothing to do with `&foo`.)

Subroutines

- You can defined a subroutine anywhere in script (although you will typically find them at end)— no forward declaration necessary

```
sub marine {
  print "hello, sea-goer number ",++$n,"!\n";
}
```

- You evoke a subroutine by using it's name in an expression

```
marine;    # invokes marine, 'hello, sea-goer number 1!\n'
marine;    # invokes marine, 'hello, sea-goer number 2!\n'
marine;    # invokes marine, 'hello, sea-goer number 3!\n'
&marine;   # invokes marine, 'hello, sea-goer number 4!\n'
```

Return values

- The last expression evaluated is the return value of the subroutine

```
sub add_a_and_b {
   print "Hey, I was invoked!\n”;
   $a + $b;
}
$a = 3; $b = 4;
$c = &add_a_and_b;          # $c gets 7, and also prints message
$d = &add_a_and_b + 3;      # $d gets 10
```

- The last expression is not always textually last!

```
sub bigger_of_a_or_b {
   if ($a > $b) {
     $a;
   } else {
     $b;
   }
}
```

Arguments

- What if we wanted the max of any two numbers? We can do that by creating an argument list
- The arguments come in via the `@_` array— elements are `$_[0]`, `$_[1]`, `$_[2]` and so on
- The @_ array is local to the subroutine— invoking another subroutine creates a new @_.

```
max(10,15); # invocation: notice a list in parens
sub max {
   if ($_[0] > $_[1]) {
     $_[0];
   } else {
     $_[1];
   }
}
```

Local variables in subroutines

- By default, all variables in Perl are global
- You can create local variables with `my()`

```
sub max {
   my $a, $b;
   ($a,$b) = @_; # give names to the parameters
   if ($a > $b) { $a; } else { $b; }
}
```

- The `my()` operator is creating an assignable list, so you can combine the first two operations

```
my ($a, $b) = @_;
```

- The `local()` operator does dynamic localization rather than lexical localization

```
local($a, $b) = @_;
```

A slight digression: Global Variables

- Global Variables are variables that are visible throughout the program
- By default, all variables in Perl are global, but…
  - `use strict;` will not allow you to have global variables unless you fully qualify them:

    ```
    $main::global = 'fully qualified global';
    ```
  - Note that we did not use the `my()` operator to make `use strict;` shut up
- You can use the `use vars` pragma to declare global variables so that you don't need to use the fully qualified name

  ```
  use vars qw/global1 global2/;
  ```
- The `qw()`, the "quote word" operator , let's you use bare words and returns a LIST

Scope

- The `my()` operator allows you to create lexical variables which are local to the scope that they are defined in
- That simply means that a variable that is declared with my() is private within the block that it is defined in and also in any blocks contained within that block
- Remember, subroutines are made up of a block

Scope

- Example:

```
my $bar = 'outside bar';
{
    my $foo = 'inside foo';
    print "1: $foo : $bar\n";
    {
        my $bar = 'inner inside bar';
        print "2: $foo : $bar\n";
    }
    print "3: $foo : $bar\n";
    blat();
}
print "5: $foo : $bar\n";

sub blat {
    my $foo = 'blat foo';
    print "4: $foo : $bar\n";
}
```

Week Three Homework

- Read `perlreftut`, `perldsc`, `perllol`, `perlpod`, and `perldebtut` handouts
- Write a program that creates a report of the mount of time a user spent logged into a computer system.

```
bjones, 27 min.
asmith, 102 min.
asmith, 12 min.
jdoe,  311 min.
bjones, 45 min.
```

- Write a subroutine that will return the sum of all of it's arguments
- Write two subroutines, `max_num()` and `min_num()`, that return the maximum and minimum of there argument lists
- Modify the two subroutines above so that they come strings instead of numbers