Programming in Perl

Week Six

Working with Lists

More I/O

Homework 5.1, 5.2, 5.3

- Regex to match target if it contains only digits:

```
m/^\d+$/
```

- Regex to match if target is positive integer or decimal number:

```
m/^\d+(\.\d+)?$/
```

- Regex to match if target is positive or negative number including scientific format (from perlfaq4):

```
m/^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/
```

Homework 5.4

- Function to print out summary of vowels in a string:

```
#!perl -w

my $quote = "This is the winter of our discontent";

print_vowels($quote);

sub print_vowels {
    my $string = shift;
    print "a ", scalar (@tmp = $string =~ /(a)/g),"\n";
    print "e ", scalar (@tmp = $string =~ /(e)/g),"\n";
    print "i ", scalar (@tmp = $string =~ /(i)/g),"\n";
    print "o ", scalar (@tmp = $string =~ /(o)/g),"\n";
    print "u ", scalar (@tmp = $string =~ /(u)/g),"\n";
}
```

Processing LISTS

- The standard way to process a list is with a foreach loop

```
foreach my $item (  @list ) {
    # do something
}
```

- There are other ways to process a list
  - `map()` applies a function to each element of a LIST returning a new LIST
  - `grep()` filters a LIST returning what was filtered out
  - `sort()` will order a LIST returning the ordered list
  - `reverse()` return an inverted LIST

`map()` operation

- The `map()` operation iterates over a LIST, applying an expression to each element in the LIST
- Using a `foreach`, it would look like:

```
foreach(@list) {
    push @new_list, $_  * 2;    # this is the expression to apply
}
```

- Using a `map()` allows you to do a LIST assignment

```
$new_list = map $_ * 2, @list;
```

- `map()` has two forms

```
map EXPR, LIST
map { BLOCK }, LIST
```

- As each element in the LIST is processed, the value is placed in the `$_` variable

Filtering a list with `grep()`

- The `grep()` operation filters a LIST using an expression, returning what was filtered

```
my @list = qw/one two three four/;
my @new_list = grep /^t/, @list;
print “@new_list\n”;    # prints “two three”
```

- Like `map()`, `$_` is used to hold the value of the current LIST element
- The current element will be returned if the expression returns true
- The syntax for grep() is

```
grep EXPR, LIST
grep { BLOCK }, LIST
```

Sorting a list with `sort()`

- Like map() , sort() returns all of the elements in the LIST, but will return them in sorted order

```
my @list = qw/one two three four/;
@list = sort @list;
print “@new_list\n”;    # prints “five four one three two”
```

- sort() can also do more complex sorts by creating a “sort function”

```
@list = sort { $a cmp $b } @list;
```

- The variables `$a` and `$b` are localized to the function, and the sort function must return `-1` if `$a < $b`, `0` if `$a == $b`, and `1` if `$a > $b`
- The operators `cmp` and `<=>` return the correct values

Writing a sort function

- You can write a subroutine if you want a more complex sort function

```
@list = sort stringwise @list

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

- Note that the localization of `$a` and `$b` happens in the `sort()`, not in the subroutine

The `reverse()` operation

- The `reverse()` operation returns a reversed LIST
- This operator is a “two trick pony”
  - It will reverse a string if the argument is a scalar

```
print reverse ‘kram’;    # prints “mark”
```

  - It will reverse a LIST if the argument is a LIST

```
my @list = reverse qw/ three two one /;
print “@list\n”;   # prints ‘one two three’
```

The -X File (Tests)

- File tests work like the test(1) UNIX command to return true/false on a file handle or string filename.

```
print “/etc/motd shouldn’t be writeable\n” if -w “/etc/motd”;
foreach $file (“fred”,”barney”,”betty”,”wilma”) {
  if (-r $file) { $goodfile = $file; last; }
}
die “no good file” unless $goodfile;
```

- File tests default to $_ (as a filename) if not given a parameter.
  - The read/write/execute/owned-by flags: -r, -w, -x, -o
  - Exists, exists with zero size, exists with non-zero size: -e, -z, -s (-s also returns the size in bytes rather than just a true/false indication)
  - Plain file, directory, symbolic link: -f, -d, -l

The `-X` File (Tests)

- Is this a terminal (The user is connected): `-t`

```
if (-t STDIN) { # are we connected to a human?
  print “Delete the log file? “;
  if (<STDIN> =~ /^y/i) {
    $delete_log = 1;
  }
}
```

- (Note: `-t` defaults to `STDIN`, not `$_`)
- is text, is binary: `-T`, `-B`
- creation time, modification time, age time (age in days): `-C`, `-M`, `-A`

```
die “Input is too old!” if -M STDIN > 28;
```

The `stat()` and `lstat()` operators

- `-X` file tests tell a lot but not everything.
- The `stat()` and `lstat()` operators give the rest.

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,
  $size,$atime,$mtime,$ctime,$blksize,$blocks) = stat(...)
```

- The fields are just like the stat(2) system call.
- The lstat() operator returns information about a symbolic link, rather than what it points at.
- Sometimes it's useful to grab part of the data with an array slice:

```
($nlink) = (stat("thisfile"))[3]; # number of links to thisfile
```

Moving around the directory tree

- The `chdir()` operator changes the working directory.
- Just like the shell's `cd` command

```
chdir("/etc") || die "Cannot chdir to /etc";
```

- Omitting the parameter takes you to your home directory (not `$_`!).

Globbing

- Globbing is the act of taking a shell filename pattern and generating a list of matching filenames.
- Perl can do this all from within the program.

```
@allfiles = <*>; @c_source_files = <*.c>;
```

- A glob returns a list of names in an array context, or the next name in a scalar.

```
while ($somename = <*.o>) {
  print “a writeable file is $somename\n” if -w $somename;
}
@etcfiles = </etc/* /usr/etc/*>;
```

- The glob is double-quote interpolated.

```
$dir = “/etc”; @dirfiles = <$dir/* $dir/.*>;
```

Directory handles

- Another way to get a list of filenames is with a directory handle (dirhandle.)
- Dirhandles come from yet another namespace, and look like filehandles.

```
opendir(ETC,”/etc”) || die “Cannot open /etc”;
foreach $file (readdir(ETC)) {
  print “one file in /etc is $file\n”;
}
closedir(ETC);
```

- The names returned by a dirhandle are in no particular order, and include the dot-files (especially dot and dotdot.)
- The names do not have any directory part.

Removing a file

- Remove a file (or a list of files) with the `unlink()` operator.

```
unlink(“this”,”that”,”theother”);
unlink(<*.o>); # removes all of the object files
```

- The return value is the number of files successfully unlinked.
  - No way to tell which files were unlinked if `unlink()` returned less than all, but you can always unlink them one at a time.

```
foreach $filename (<*.o>) {
  unless (unlink($filename)) { # successful?
    print STDERR “Cannot unlink $filename\n”;
  }
}
```

- Can’t remove directories this way—see `rmdir()` later.

Renaming a file

- The `rename()` operator is like the `mv` command.

```
rename(“old”,”new”); # like: mv old new
## change all foo.old to foo.new in the current directory
foreach $old (<*.old>) {
  ($new = $old) =~ s/\.old$/.new/;
  rename($old,$new) unless -e $new;
}
```

Making and removing directories

- The `mkdir()` and `rmdir()` operators work like their same-named command counterparts.

```
mkdir(“dir1”,0755); # like: mkdir dir1; chmod 755 dir1
rmdir(<dir*>); # like: rmdir dir*
```

- `mkdir()` returns true if successful.
- `rmdir()` return the number of directories successfully removed (like `unlink()`)
- You cannot remove a non-empty directory—use `unlink()` on the contents firs

```
$dir = “/var/tmp/scratchdir”;
unlink(<$dir/* $dir/.*>);
rmdir($dir);
```

Homework Week Six

1. Write a function that, when given a hash, returns the list of hash values sorted by the hash keys. Write the same thing in one line using a `map()` function.
2. Modify the one-line version above to also filter out any value less than 25