Lecture 21
Advanced Perl Programming

In the previous lecture, we learned some basic constructs of perl programming including regex processing in Perl. Combining regex constructs with other high level programming capabilities of Perl is one of the main advantages of using Perl for tasks that require text processing. In this lecture we will cover few other miscellaneous topics of interests such as sub-routines, command line arguments, advanced data structures, reference variables in Perl 5 and system programming.

Let us first understand how to define sub routines in perl.

Subroutines

Subroutines are part of many of the modern programming languages. C has functions and Java has methods and Pascal has procedures and functions for defining subroutines. Subroutines allow us to break the code into manageable pieces. Perl subroutines, like many of the high level languages, can have input parameters, local variables, and can return the answers back to the calling routine (eg: main). Let us start with a simple sub routine definition.

```
sub name {
   statements;
}
```

Subroutines can be defined anywhere in your program. However, we recommend placing subroutines at the end of the file so main program can appear in the beginning. A subroutine can be called by simply using its name: **name( );**

Subroutines can be called by other subroutines and subroutines can return values that can be used by other subroutines.

```
sub sum {
  return $a + $b;
}
```

So we can call this as:

```
$a = 12; $b = 10;
$sum = sum();
print “the sum is $sum\n”;
```

Subroutines can also return a list of values such as

```
sub list_of_values {
   return ($a,$b,$b,$a);
}
```

So we can write
@arr = list_of_values( );

Passing Arguments

A perl subroutine can be called with a list in parenthesis. For example, we can write a generic add function as follows.

```
sub add {
    $tmp = 0;  # this defines a global variable
    foreach $_ (@_) {
      $tmp += $_;
  }
  return $tmp;
}
```

In the above code @_ refers to the list passed to the function when it is called. For example, if we call the function as: **add($a,$b,$c);** or add(3,4,5,6);
Then @_ **= ($a,$b,$c)** or @_ = (3,4,5,6)

So **$_[0] refers to $a, $_[1] refers to $b** etc.

Local variables

Perl subroutines can define local private variables. Here is an example with a local variable.

```
sub product {
  my ($x);  # defines the local variable x
  foreach $_  (@_) {  $x *= $_;}
  return $x;
}
```

You can have a list of local variables by simply expanding the list as:
my ($x, $y, @arr);

Command Line Arguments in Perl

A Perl program can take command line arguments. One or more command line arguments can be passed when calling a perl program. For example,

- **perl program.pl infile.txt outfile.txt**

takes two command line arguments, infile.txt and outfile.txt. The number of command line arguments is given by **$#ARGV + 1** and command line arguments are named **$ARGV[0], $ARGV[1],** etc. In the above example,

```
$ARGV[0] = infile.txt
$ARGV[1] = outfile.txt
```

Here is a Perl program that reads from a file and writes to another file.

```
#! /usr/local/bin/perl -w
$numargs = $#ARGV + 1;
if ($numargs >= 2){
  open(INFILE, $ARGV[0]);
  open(OUTFILE,">$ARGV[1]");
  foreach $line (<INFILE>) {
        print OUTFILE $line;
   }
  close(INFILE);
  close(OUTFILE);
}
```

- You can run this program with > **perl program.pl infile.txt outfile.txt**

LWP and Getopt

LWP is the Perl library for processing functions related to www activities. For example, we can get the source code for a web page by typing

```
use LWP::Simple;
$_ = get($url)
```

Where URL refers to a string such as http://www.cs.cmu.edu and $_ is the default system variable.

Command line arguments can be obtained as above or for command line options such as -n 100 -t dir_name

You can use:

```
use Getopt::Long;
$images_to_get = 20;
$directory = ".";
GetOptions("n=i" => \$images_to_get,
                "t=s" => \$directory);
```

Hashes

Hash table is a collection of **<key, value>** pairs. An array is a special hash table with key being the array index. Using the key, any value can be found. The relationship between key and value is given by a hash function.

H(key) = value

The advantage of a hash table (compared to other data structures such as linked lists and trees) is that finding something in a hash table is easy. We simply provide the key we are looking for to a hash function, and the function returns a value. For example think of a hash function that maps each string to the sum of its characters where characters are assigned values such as ‘a’=1, ‘b’=2, .. ‘z’=26 etc. So

H(“abc”) = 1+2+3 = 6

and the string “abc” can be stored in array[6]. Of course you probably immediately notice a problem with this approach. All permutations of the string “abc” maps to the same value, producing what we call a “collision”. Collisions cannot be fully avoided in hash tables, but can be minimized by selecting a “good” hash function. For example if we choose the hash function,

H(“abc”) = ‘a’ + 2*’b’ + 2^2 * ‘c’;

Then all permutations of the string “abc” will not have the same hash code.

Hashing is a popular technique for **storing and retrieving** data and hash tables are becoming popular data structures to be used in programming. Hash tables are easy to program and are very effective in applications where searching is a major operation (eg: a search engine). Hashing is an ideal way to store a dictionary. Major operations required by a dictionary are **updates and lookups** that can be achieved in constant time or **O(1)** using a hash table. Hash tables are not suitable applications that require data to be stored in some order. Finding max, min or median is inefficient in a hash table. These operations can be efficiently done using a sorted array.

Hashing in Perl

Perl provides facilities for hash tables. A hash variable name is preceded by a percent sign (%). For example we can define a hash as follows.

```
%hash = ( ); # initializes a hash to empty set. We can add elements later
$hash{‘guna’} =  “aa”;
$hash{‘neil’} = “ab”;
$hash{‘george’}=”ac”;
```

So our hash looks like

```
%hash = ('guna', `aa', 'neil', `ab', 'george', `ac' );
```

The key set of a hash table is unique. However, some keys may have the same value (based on hash function). For example, the keys 'abc' and 'bac' have the same value if the value is computed by a hash function that simply add characters of the key. Hash tables are typically implemented using arrays. When two keys maps to the same array index, then we call that a **collision**. There are techniques like separate chaining and linear and quadratic probing to deal with collisions. Hash Table is a map between a set of values and set of keys.

Any value in a hash table can be changed by
$hash{'guna'} = 'ab';
$hash{'neil'} = 'aa';

The Key Set

The key word "keys" provides a list of all keys in a hash table. For example,

@keyset = keys(%hash);

provides the keyset for the hash. To print all keys we can do (or simply print the keyset as defined above)

```
foreach $key (keys %hash) {
  print "$key\t$hash{$key}\n";
}

or

print @keyset;
```

The value Set

Now that we know the key set, we can find the values for each key by

```
foreach $key (keys(%hash)){
    print $hash($key);
```

The values function

values(%hash) also returns the set of values of the current hash. So we can write

```
@valueset = values(%hash);
```

Each function

We can iterate over all the (key,value) pairs in the hash table using "each" function. For example

```
while (($key,$value) = each(%hash)){
    # do something with key and value
}
```

Each(%hash) function returns an empty list when the end of the hash is reached.

Ex 1: Write a Perl program to store a dictionary (you can use dictionary.txt from a lab) in a hash table. Use each word in the dictionary as the key. You may use any hash function to compute the value. Ask the user to input a word interactively and look for the word in the hash table. If exists print the value.

The delete function

To delete a **(key,value)** pair from the hash, we can use the "delete" function. To use delete,

delete $hash{'key'};

References in Perl

With the release of Perl 5, programmers now can represent complex data structures in Perl. For example, we may want to build an array of arrays, or an array of hash tables and many combinations of data structures. References to objects can be created using the "\" operator. A reference to an object is much like the address operator in C. Here are some examples.

Creating a Reference to a scalar
$num = 10;
$numref = \$num;

Creating a Reference to an array
@array = (guna, me, neil);
$arrayref = \@array;

Creating a Reference to a hashtable
%hash = {guna, aa, me, bb, him, cc};
$hashref = \%hash;

Dereference

Dereferencing a Reference to a scalar
$num = 10;
$numref = \$num;
print $$numref; # prints the value 10

Dereferencing a Reference to an array using -> operator
@array = (guna, me, neil);
$arrayref = \@array;
print **$arrayref->[0];** # prints 'guna'
Dereferencing a Reference to an array using -> operator
@arrayref = [guna, me, [blue, red]];
print **$arrayref->[2][1];** # prints 'red'

You can find many resources on the web to read more about references.
http://search.cpan.org/dist/perl/pod/perlref.pod

Systems Programming with Perl

Perl can interface with OS to provide you with many valuable tools. As we will see later, many of the UNIX commands can be directly called with Perl.
Let us assume that we need to list all the files in the current directory. This can be accomplished easily by the following Perl code

Listing Files in a Directory

```
opendir(DIR,".");
foreach $file (readdir(DIR)) {
  print "$file \n";
}
close(DIR);
```

Here opendir opens the current directory (".") and pass a reference to the directory DIR to readdir function. The code prints out all files in the current directory.

Ex 2: Write a perl program to output all files in a specific directory that contains the substring .c (use regex)

Changing Directories

In UNIX we can use the chdir command to change a directory. For example

- **chdir "somedir"**

changes the current directory to somedir. We can use similar code in perl programs. For example, you can ask the user to enter a name of a directory and list all the files in that directory as follows.

```
print "which directory to change to : ";
chomp($dir = <STDIN>);
if (chdir $dir){
 print "we are now in $dir \n";
} else {
 print "we could not change to $dir \n";
}

opendir(DIR,".");
foreach $file (readdir(DIR)) {
  print "$file \n";
}
```

```
close(DIR);
```

Every process has its own current directory. If you launch the program from the parent process, it won't affect the parent shell that launched the perl program.

Removing a File
This is really risky behavior. You should try not to run any program that removes files without making sure you really want to do that. If you delete a file accidentally, your system administrator can still find the file for you. But unless sys admin is your good buddy this is something you want to avoid. If you want to remove a file in UNIX, you would do

➢ **rm "guna.c"**

removes the file guna.c. Perl also has an unlink function, that removes the name for a file. Typically removing the name of a file, removes the file unless you have created hard links for the file using link function. Here is Perl code that removes a file.

```
print "what file do you want to remove :";
chomp($name = <STDIN>);
unlink ($name);
```

There are variations of unlink function. You can remove multiple files or all the object files in a directory as follows.

```
unlink("file1", "file2");

foreach $file (<*.o>) {
  unlink ($file) || warn "file $_ cannot be deleted\n";
```

Renaming a File
A file can be renamed as follows

```
rename ("guna.c", "temp.c") || die "cannot rename the file";
```

Modifying Permissions
The permission on a file or directory ca be modified using chmod. The following code modifies the file permission of two files given as a list. It can be generalized to any array of files.

```
foreach $file  ("guna.c", "temp.o") {
   unless chmod (O666, $file) {
      warn "could not chmod the file $file \n";
   }
}
```

Running a Perl script from another Perl Script
Suppose you are provided with a perl script that performs a task. Say it creates a bunch of directories using a file.txt that contains some ID's. Now you need to run this script inside another. You can do something like this.

```
#!/usr/local/bin/perl

system 'perl mkdir.pl file.txt';

# write the script for this perl program
```

Ex 3 : Write a perl program that reads a file of folder names and create sub directories in the current directory with permission for each folder set to rwx for all.

Ex 4 : Write a perl program that request the directory name and list all the files in that directory that have file permissions rwx for Group.