Intermediate Perl

Boston University
Information Services & Technology

Course Coordinator: Timothy Kohl

Last Modified: 09/19/13

Outline

- explore further the data types introduced before.
- introduce more advanced types
  - special variables
  - references (i.e. pointers)
  - multidimensional arrays
  - arrays of hashes
- introduce functions and local variables
- dig deeper into regular expressions
- show how to interact with Unix, including how to process files and conduct other I/O operations

Data Types

Scalars revisited

As we saw, scalars consist of either string or number values.
and for strings, the usage of " versus ' makes a difference.

Ex:

```
$name="Fred";
$wrong_greeting='Hello $name!';
$right_greeting="Hello $name!";
#
print "$wrong_greeting\n";
print "$right_greeting\n";
```

yields

```
Hello $name!
Hello Fred!
```

If one wishes to include characters like **$** , **%** , \ , **"**, **'** (called meta-characters) in a double quoted string they need to be preceded with a **** to be printed correctly

Ex:

```
print "The coffee costs \$1.20\n";
```

which yields

```
The coffee costs $1.20
```

The rule of thumb for this is that if the character has some usage in the language, to print this character literally, escape it with a ****

Sometimes we need to insert variable names in such a way that there might be some ambiguity in how they get interpreted.

Suppose

`$x="day"` **or** **`$x="night"`**

and we wish to say "It is daytime" or "It is nighttime" using this variable.

incorrect

```
$x="day";
print "It is $xtime\n";
```

This is interpreted as a variable called **`$xtime`**

correct

```
$x="day";
print "It is ${x}time\n";
```

putting **{ }** around the name will insert **`$x`** properly

Arrays revisited

For any array **`@X`**, there is a related <u>scalar</u> variable **`$#X`** which gives the index of the **last defined** element of the array.

Ex:

```
@X=(3,9,0,6);
print "$#X\n";
```

yields

3

Similarly, arrays can be viewed in what is known as **'scalar context'**

Ex:

```
@blah=(5,-3,2,1);
$a = @blah;
```

Here, `$a` equals **4** which is the current **length** of `@blah`

(i.e. `$#X` = `@X-1` if you want to remember which is which.)

We can print whole arrays as follows.

```
@X=(4,5,6);
print "@X";
```

yields

```
4 5 6
```

Note, if you drop the `"` then the array still prints, but without the spaces between each element.

stacks and queues

There are built in functions that can manipulate arrays in such a way that **any** array can be treated as a stack or queue!

Ex:

```
@X=(2,5,-8,7);

push(@X,10); # now @X=(2,5,-8,7,10);

$a=pop(@X);  # now @X=(2,5,-8,7) and $a=10
```

- **pop removes the last element** of an array
- **push adds an element** to the end of an array

Likewise,

```
@X=(2,5,-8,7);

unshift(@X,10); # now @X=(10,2,5,-8,7);

$a=shift(@X);  # now @X=(2,5,-8,7) and $a=10
```

- **shift removes the first element** of an array
- **unshift adds an element** to the beginning of an array

miscellaneous array operations (neat tricks)

If **`$a`** and **`$b`** are two scalars, then **`($a,$b)`** is implicitly an array, and so the following works.

Given

```
$a=1; $b=2;
($a,$b)=($b,$a);
print "$a $b\n";
```

we get

```
2 1
```

(i.e. We can swap two values without needing a third temporary variable!)

Using the **`foreach()`** function, one can loop over the elements of an array and <u>modify</u> each element along the way.

Ex:

```
@a=(1,2,3);
foreach $element (@a){
      $element = $element*4;
}
# now @a=(4,8,12)
```

Associative arrays revisited

Last time we introduced the `keys()` function which returns (as an array) the keys in a given associative array.

Similarly, there is a `values()` function which returns (also as an array) the values of an associative array.

Ex:

```
%Grades=("Tom"=>"A","Dick"=>"B","Harry"=>"C");

@People=keys(%Grades);
# @People=("Tom","Dick","Harry");

@letters=values(%Grades);
# @letters=("A","B","C");
```

There is also a way of looping over **all** the key-value pairs in an associative array using the `each()` function.

Ex:

```
%Grades=("Tom"=>"A","Dick"=>"B","Harry"=>"C");
while(($person,$grade)=each(%Grades)){
      print "$person received a $grade\n";
}
```

yields:

```
Tom received a A
Dick received a B
Harry received a C
```

There is also a function for removing elements from an associative array.

Ex:

```
%Appointment=("Monday"=>"1PM",
              "Wednesday"=>"10AM",
              "Friday"=>"4PM");
```

Suppose now that our Wednesday appointment is cancelled.
We can then do:

```
delete($Appointment{"Wednesday"});
```

and now **`%Appointment`** consists of just two key and value pairs.

Special variables

In Perl there are a number of variables (scalars, arrays and hashes) which have special meanings within Perl, but which you can use as well.

scalars

`$_` default input variable

As we have seen, one can take standard input from the keyboard (or from a Unix pipe) as follows.

```
while($line=<STDIN>){
      chomp($line);
      print "$line\n";
}
```

One could rewrite this very compactly as follows

```
while(<STDIN>){
      chomp();
      print "$_\n";
}
```

Here, the line of input was not explicitly assigned to a user specified variable, but rather, Perl assigned it to the special variable **`$_`** instead.

Likewise **`chomp()`** operates on **`$_`** by default!

Actually, one could rewrite this even more compactly as follows

```
while(<>){
      chomp();
      print "$_\n";
}
```

as **`<>`** is synonymous with **`<STDIN>`**

We can also use **`$_`** for regular expression matching.

Ex:

```
while($line=<STDIN>){
      chomp($line);
      if($line =~/blah/){
            # do something
      }
}
```

can be rewritten as

```
while(<>){
      chomp();
      if(/blah/){
            # do something
      }
}
```

There are **many** other default scalars

Ex:

`$0`	- name of the Perl script currently running
`$]`	- version of Perl that you are using
`$.`	- number of lines you have currently read in from a given file (e.g. **STDIN**);

Additionally, there are default **arrays** and **associative arrays**

`@ARGV` - **program arguments** passed to the script you are running

ex: If your script is called 'myscript' and if you invoke it as follows

```
>myscript Tom Dick Harry
```

then

```
$ARGV[0]="Tom"
$ARGV[1]="Dick"
$ARGV[2]="Harry"
```

An important associative array that Perl keeps track of is `%ENV`
which contains information about your current environment

Ex:

```
$ENV{HOME}          # your home directory
$ENV{LOGNAME}       # your login name
$ENV{PWD}           # the current directory
```

An easy way to see all of `%ENV` is as follows:

```
#!/usr/bin/perl
foreach $key (keys(%ENV)){
      print "$key => $ENV{$key}\n";
}
```

Advanced Data Types

References

In languages like C one has the notion of a pointer to a given data type, which can be used later on to read and manipulate the contents of the variable pointed to by the pointer.

In Perl, these are called (hard) **references** and they are the key to building all manner of complex data structures, including

- multidimensional arrays
- hashes of arrays
- hashes of hashes

and more...

Ex:

```
$x="5";
$rx=\$x;
```

$rx is a reference to the scalar $x

To manipulate **`$x`** using **`$rx`** we need to 'de-reference' the reference.

```
$$rx="6";    # now $x=6;
```

The extra leading **`$`** gives us the value of what's pointed to by the reference.

One can copy references just as one would any other scalar value since a reference to anything is always a scalar.

So if we do

```
$x="5";
$rx=\$x;
$rx2=$rx;
```

then

`$$rx2` also equals "5"

One can also have references to other data types like arrays and associative arrays.

Ex:

```
@Names=("Tom","Dick","Harry");
$n=\@Names;
```

How do we use this?

Ex:

```
push(@{$n},"John");
```

So now

```
@Names=("Tom","Dick","Harry","John")
```

That is, if **`$n`** is a reference to an array, then **`@{$n}`** is the array itself.

Also, we could have done this

```
@Names=("Tom","Dick","Harry");
$n=\@Names;

$$n[3]="John";
```

and again

```
@Names=("Tom","Dick","Harry","John");
```

```
i.e. $Names[3] = $$n[3]
```

Similarly, if we have

```
%CourseGrade=("Tom"=>"A",”Dick"=>"B","Harry"=>"C");
```

and if we set

```
$h=\%CourseGrade;
```

then we can access and modify **`%CourseGrade`** via the reference, **`$h`**

Ex:

```
$$h{"John"}="D";
```

yields

```
%CourseGrade=("Tom"=>"A",
              "Dick"=>"B",
              "Harry"=>"C",
              "John"=>"D");
```

Multidimensional arrays

A multidimensional array can be created and accessed in a number of ways.

As a whole

```
@A=(
    ["a","b"],
    ["c","d"]
    );
```

or, entry by entry

```
$A[0][0]="a";  $A[0][1]="b";
$A[1][0]="c";  $A[1][1]="d";
```

One can also create more exotic structures.

associative array of (ordinary) arrays

```
%Food=(
        "fruits"        => ["apples","oranges","pears"],
        "vegetables"    => ["carrots","lettuce"],
        "grains"        => ["rye","oats","barley"]
       );
```

i.e. `$Food{"vegetables"}[1]="lettuce";`

associative array of associative arrays (a hash of hashes)

```
%StateInfo=(
            "Massachusetts" => { "Postal Code" => "MA",
                                 "Capital" =>"Boston"
                               },

            "New York"      => { "Postal Code" => "NY",
                                 "Capital" => "Albany"
                               }
          );
```

i.e.

```
$StateInfo{"New York"}{"Postal Code"}="NY";
```

Note the usage of the **( )** and **{ }** above.

Behind the scenes, all these structures are managed using references.

We’ll explore the inner details in the next tutorial.

With Perl, the syntax is such that you can create very flexible structures.

Most of the time, what seems reasonable on paper actually works syntactically!

Functions

In order to write more modular Perl scripts, one uses functions.

The general syntax is

```
sub function_name {

      # do something

}
```

Invoking the function is done using either

`&function_name()` or **`function_name()`**

The **&** before the name is optional if one is simply invoking the function.

One can put functions anywhere within a script but it's customary
to put them at the end. (the reverse of the custom in C)

Parameters (by value)

When one passes parameters to a function, they arrive in the function
in the array `@_`

Ex:

```
sub converse{
      my ($first,$second) = @_;
      print "$first spoke to $second\n";
}

converse(”Holmes","Watson");
```

yields

```
Holmes spoke to Watson
```

The individual elements of `@_` are accessible as **`$_[0]`** , **`$_[1]`** , ... etc.

So we could have also written this as

```
sub converse{
      my $first  = $_[0];
      my $second = $_[1];
      print "$first talked to $second\n";
}
```

The **`my`** directive is used to make the variables **`$first`** and **`$second`** local to the subroutine. (what's known as lexical scoping)

That is, it is defined ***only for the duration of the given code block*** between **{** and **}** which is usually the body of the function anyway.

With this, one can have the same variable name(s) used in various functions without any potential conflicts.

Another option for obtaining the parameters passed to a function is to use the **`shift`** function we saw earlier.

```
sub converse{
      my $first  = shift;
      my $second = shift;
      print "$first talked to $second\n";
}
```

Recall that **`shift(@X)`** extracts the leftmost element of **`@X`** and removes it from **`@X`** and that subsequent calls remove the remaining elements of **`@X`** in the same fashion.

Here, calling **`shift`** with no arguments implies that we wish to extract the elements of **`@_.`**

Parameters (by reference)

One may pass to a sub, a **reference** to a given variable, and thereby allow the sub to modify this passed variable.

Ex:

```
 sub myfunction{
      my $x=shift;
      $$x=$$x+10;
}

$a=3;
myfunction(\$a);
print "$a\n";
```

Here we modify the value of the underlying variable by dereferencing it with the extra leading $.

Here we pass a **reference** to **$a** which allows the sub to modify **$a** itself.

yields

```
13
```

Return values

To receive values from a function, one can use the **return** command.

Ex:

```
sub add_array{
      my @numbers=@_;
      my $sum=0;
      my $n;
      foreach $n (@numbers){
            $sum += $n;
      }
      return $sum;
}

$s = add_array(3,5,10,6,-1);
```

or by invoking the return value by itself on the last line of the function.

Ex:

```
sub add_array{
      my @numbers=@_;
      my $sum=0; # local variable
      foreach $n (@numbers){
            $sum += $n;
      }
      $sum;
}
```

Note, one can return scalars, arrays or associative arrays from a function.

Regular expressions

Recall that to match a variable against a regular expression, the syntax is:

```
if($x =~ /pattern/){
      # do something
}
```

or

```
if($x !~ /pattern/){
      # do something
}
```

!~ means not match

where **`pattern`** is some regular expression.

We saw last time that one may memorize parts of a regular expression and also so substitutions (i.e. rewrites) based upon the results of a pattern match.

We can do more than this by taking the results of the match and make the replacement be based upon a *expression* involving the matched components.

Ex:

```
   $x=“Fred: 70 70 100”;
   $x =~ s/(\d+) (\d+) (\d+)/$1+$2+$3/e;
   print “$x\n”;
```

returns

```
  Fred: 240
```

We can even use functions of **$1, $2,...** as well.

Ex:

```
$x=“Fred: 70 70 100”;
$x=~s/(\d+) (\d+) (\d+)/avg($1,$2,$3)/e;
print “$x\n”

sub avg{
      my @list=@_;
      my $n=0,$sum=0;
      foreach (@list){
            $sum+=$_;
            $n++;
      }
      return($sum/$n);
}
```

yields

```
Fred: 80
```

split() and join()

Two string operations related to regular expressions are **`split()`** and **`join()`** .

Ex:

```
$sentence="The quick brown fox jumped over the lazy dog";
@words=split(/\s/,$sentence);

# @words=("The","quick","brown","fox",...,"dog");
```

`split(/pattern/,$x)`

splits **`$x`** at every occurrence of **`/pattern/`** in **`$x`**
and returns the components in an array.

(note, any valid regexp can be used)

This is extremely useful if we wish to process collimated data.

Ex:

```
#!/usr/bin/perl
while($line=<STDIN>){
      chomp($line);
      @C=split(/\s+/,$line);
      print "$C[0] $C[1]\n";
}
```

← call this **twocol**

will take the output of a command such as '**who**' and print the first two columns.

```
>who | twocol
```

Note, the **`/\s+/`** indicated as a separator allows for <u>irregular</u> column spacing as well as allowing for real spaces **`" "`** or tabs **`"\t"`** etc.

Likewise one can easily join elements of an array into a string.

```
@words=("The","quick","brown",...,"lazy","dog");
$sentence=join(" ",@words);

#$sentence="The quick brown fox jumped over the lazy dog";
```

`join($separator,@stuff)`

joins the elements of **`@stuff`** with the string **`$separator`** in between each 'word'

one liners

logical short circuiting

```
(something) || (something else)
```

If **`(something)`** returned true then **`(something else)`** is **not** executed.
If **`(something)`** returned false then **`(something else)`** **is** executed.

```
(something) && (something else)
```

If **`(something)`** returned true then **`(something else)`** **is** executed.
If **`(something)`** returned false then **`(something else)`** **is not** executed.

i.e. Any command inside parentheses returns a logical value.

Ex:

```
chomp($x=<STDIN>);
($x eq “thanks”) && (print “yer welcome\n”);
```

Here, if the input `$x` was `“thanks”` then the output should be `“yer welcome”` but *only* if the input was `“thanks”`

```
chomp($N=<STDIN>);
($N % 2) || (print “$N is even\n”);
```

Here, if `$N` is divisible by two then `($N % 2)` is `0`, and therefore, logically, false so the right hand side (the print statement) must be evaluated, otherwise the remainder is 1 and therefore true (i.e. `$N` is odd) so the print statement is *not* executed.

I/O and Interaction with the Operating System

As we saw in our interactive script:

```
#!/usr/bin/perl
print "What is your name? ";
$name=<STDIN>;
chomp($name);
print "Hello there $name.\n";
```

we can take input from the keyboard and assign it to a variable name.

This is done using what's known as the STDIN filehandle which is how Perl deals with external files in general.

In this case, we asked for one line of standard input from the user.

However, if we wish to process a sequence of lines of standard input we can use, for example, a **`while()`** loop.

Ex: Suppose we want to take each line of input and surround it with brackets **[ ]**

```
while($line=<STDIN>){
      chomp($line);
      print "[$line]\n";
}
```

The lines of input can come from different sources than just the keyboard.

Ex: (Let's make that chunk of code into a script called 'bracket')

```
#!/usr/bin/perl
while($line=<STDIN>){
      chomp($line);
      print "[$line]\n";
}
```

Don’t forget, if you’re on a Unix system, you need to make this executable with the chmod command as before.

Now try this,

```
>echo 'Hello' | bracket
```

```
[Hello]
```

Or how about this:

```
>ls -al | bracket
```

(The output will depend on what directory you are in, but you'll get a listing of the current directory with each entry wrapped in brackets.)

Standard input is not the only way to read in data to a Perl script.

One can open specific files with the **`open()`** and **`close()`** commands.

To open a file for **reading**:

```
open(MYFILE,"/home/me/somefile");
while($line=<MYFILE>){
      # do something
}
close(MYFILE);
```

If we wish to open a file for **writing**:

note the **>**

```
open(MYFILE,">/home/me/somefile");
print MYFILE "Hi there!\n";
close(MYFILE);
```

If one wants to **append** to a file, the syntax is similar. (and very Unix like)

note the **>>**

```
open(MYFILE,">>/home/me/somefile");
print MYFILE "Here is some more stuff!\n";
close(MYFILE);
```

Note, when doing any kind of I/O like this, one should check that the operation of opening the file actually succeeded.

Ex: (terminate program if unable to open file)

```
(open(MYFILE,"/home/me/somefile")) || (die "Sorry!\n”);
```

If the **`open()`** operation fails (i.e. returns false) then the program **`die`** 's with the error message specified.

Also, you should close any open filehandle before your program terminates or else buffered data may not get written to the file.

Say one wants to read the contents of a directory, the commands for this are **`opendir(), readdir(), and closedir()`**

Ex:

```
opendir(D,"/home/me");
while($entry=readdir(D)){
      print "$entry\n";
}
closedir(D);
```

gives an **'ls'** of the directory **`/home/me`**

Also, no **`chomp()`** operation is necessary since **`readdir()`** does not tack on a newline **`\n`** at the end.

There are a number of 'file test' operators which can be used to give information about a given file or directory.

Ex: Let's modify the last example so that only subdirectories of **`/home/me`** are listed.

```
opendir(D,“/home/me");
while($entry=readdir(D)){
   (-d “/home/me/$entry") && (print "$entry\n");
}
closedir(D);
```

-d tests to see if the given object is a **directory**

There are others as well. (See the quick reference.)

As for interacting with the system directly, there are several possibilities.

`system("command")` - This is, as in C, allows one to invoke Unix commands from within a script.
Moreover, the script waits until the call finishes before proceeding.

`` `command` `` - This functions similarly to `system()` except that one can take output from the command and assign it to a variable.

Ex:

```
@wholist=split(/\n/,`who`);
# @wholist contains the lines of
# the output of the who command
```

Another option is to open a process as a filehandle.

Ex:

```
open(WHO,"who|");
while($line=<WHO>){
      print "$line";
}
close(WHO);
```

In this case, we read output from the who command as if it were a file.

Likewise, we can open such a process filehandle for output too.

Ex:

```
open(LP,"|lpr -Pprintername");
print LP "Hi There!\n";
close(LP);
```

Note, when one closes a process filehandle, Perl will wait for the process to terminate. If not closed, the given process keeps running.

References for further information on Perl

Books

- Learning Perl by Randal L. Schwartz & Tom Christiansen (O'Reilly)
- Programming Perl by Larry Wall, Tom Christiansen and Jon Orwant (O' Reilly)
- Perl in a Nutshell by Ellen Siever, Stephen Spainhour, and Nathan Patwardhan (O' Reilly)

Web

http://www.perl.com

http://math.bu.edu/people/tkohl/perl ← My Perl Page!

Intermediate Perl

Boston University
Information Services & Technology

Course Coordinator: Timothy Kohl