Intermediate Perl

Boston University
Information Services & Technology

Course Coordinator: Timothy Kohl

Last Modified: September 2021

1

Outline

- explore further the data types introduced before.
- introduce more advanced types
  - special variables
  - 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

2

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!
```

3

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 a cup.\n";
```

which yields

```
The coffee costs $1.20 a cup.
```

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 \

4

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

5

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

6

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.)

7

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 <u>without</u> the spaces between each element.

8

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

9

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

10

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!)

11

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

Ex:

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

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");
```

13

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
```

14

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.

15

Advanced Data Types

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";
}
```

16

One could rewrite this very compactly as follows

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

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

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

17

Actually, one could rewrite this even more compactly as follows

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

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

18

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
      }
}
```

19

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**);

20

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"
```

21

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";
}
```

22

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";
```

23

One can also create more exotic structures.

associative array of (ordinary) arrays

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

so the statement

```
print $Food{"vegetables"}[1];
```

yields

```
lettuce
```

24

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.

25

Behind the scenes, all these structures are managed using what are known as references which 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!

26

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 (mostly) optional.

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 **`@_`**.

31

parameters (by reference)

One may pass to a sub, a **reference** to a given variable, and thereby allow the sub to modify the underlying 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
```

32

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);
```

33

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.

34

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.

35

`parentheses as memory`

To remember a component of a regular expression, one must use `( )`

ex:

one or more word characters

```
$x = ~/(\w+)\s\1/;
```

the word in ( ) again

which would match if, for example,

```
$x="yo yo"
```

(\w+) \s \1

36

In fact, one can save the memorized portions of a regular expression match.

```
print "Enter Name: Last,First> ";
$entry=<STDIN>;
chomp($entry);
if($entry=~/(\w+),(\w+)/){
     $lastname=$1;
     $firstname=$2;
     print "Your name is $firstname $lastname.\n";
}
```

here `$entry=~/(\w+),(\w+)/`

```
$1      $2
```

```
Note: One can make
this more flexible by
allowing for spaces
before or after the
comma.

Ex:

/(\w+)\s*,\s*(\w+)/
```

37

`substitutions`

When matching a pattern, we can also force a substitution to take place if the pattern matches, and therefore modify the string.

```
$x = "You say hello, I say goodbye";
$x =~ s/hello/goodbye/;
# $x is now "You say goodbye, I say goodbye"
```

general syntax

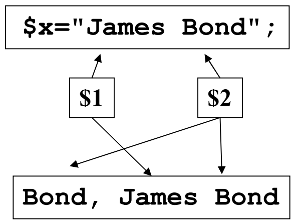
```
$x =~ s/pattern/replacement/;
```

38

One can also use memorized portions of a regexp with substitutions.

Ex: (neat trick)

```
$x = "James Bond";
$x =~ s/(\w+)\s(\w+)/$2, $1 $2/;
# $x is now "Bond, James Bond";
```



39

It is important to note that any kind of pattern matching operation takes place from left to right in a string.

```
This is particularly important when using memorized
Components via () since it is the first (leftmost)
occurrence of a pattern that will be assigned to
$1, $2 etc.

Also, when matching a pattern with a multiplier:

e.g.

            /\w+/

Perl will attempt to match as much
(i.e. be as 'greedy' ) as possible.
```

40

This is also important when doing substitutions.

Ex:

```
$x="three different words";
$x=~s/(\w+)/<$1>/;
print "$x\n";
```

yields

```
<three> different words
```

```
The first occurrence of \w+ was three
(remember greedy matching) and
so this is what was assigned to $1
and modified in $x by the substitution.
```

41

```
If we wanted to apply this operation to every
word in $x we need to use the 'g' switch
to make the substitution operation be repeated
on every match.
```

Ex:

```
$x="three different words";
$x=~s/(\w+)/<$1>/g;
print "$x\n";
```

```
<three> <different> <words>
```

```
force substitution
on every match in the
string
```

42

We can also take the results of memorized match and make the replacement be based upon an *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
```

43

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
```

44

split() and join()

Two useful 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)

45

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.

46

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'

47

Logical Short Circuiting

one liners

```
(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.

48

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**"

```
print "What day of the week is it?\n";
chomp($day=<STDIN>);
($day !~ /Friday/) || (print "End of the week!\n");
```

Here, if it's not Friday then we don't say that it's the end of the week, but if it is Friday then we let the user know it's the end of the week.

49

I/O and Interaction with the Operating System

As we saw previously, we can take (standard) input from the keyboard like so:

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

or we can take multiple lines of standard input from piped in data.

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

```
ls -al | bracket
```

50

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);
```

51

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);
```

52

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.

53

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.

54

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.)

55

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
```

56

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.

57

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.

58

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!

59

Intermediate Perl

Boston University
Information Services & Technology

Course Coordinator: Timothy Kohl

60