5) Input and Output

Matt Webster
IMBIM, BMC
matthew.webster@imbim.uu.se

reading from <STDIN>

- standard input = the keyboard
- standard output = the screen
  - can be redirected to a file using > in unix

```
while ($line=<STDIN>) {
    print "you typed $line";
}
```

or

```
while (<STDIN>) {
    print "you typed $_";
}
```

reading from <STDIN>

- an important difference

```
while (<STDIN>) {  }
# will read from standard input one line at a time

foreach (<STDIN>) {  }
# will read everything from standard input before looping
# not a good idea to use this...
```

reading from <>

- the diamond operator
- can be used to make programs that process files like unix commands
- e.g. on the command line you could type

```
./perl_script.pl my_file.txt
```

`my_file.txt` could accessed from `perl_script.pl` using

```
while (<>) {  }
```

but it is often better to use a filehandle (see later)

@ARGV

- it is common to run programs with invocation arguments
- these are just values coming after the program name. e.g.

```
./perl_script.pl -s -v 500
```

the values are place in @ARGV

```
$arg1 = shift @ARGV;
```

$arg1 contains -s

```
$arg2 = shift @ARGV;
```

$arg2 contains –v

```
$arg3 = shift @ARGV;
```

$arg3 contains 500

output to standard output

```
print @array;  # no interpolation
```

is not the same as

```
print "@array";   # interpolation
```

`print @array` prints the list with no spaces between

`print "@array"` performs interpolation

- `STDOUT` is normally the screen, it is redirected in unix with >
- if you print to `STDERR` (standard error), it will not be redirected by >

printf

- can be used to format output
- most useful to print numbers with decimal places

```
printf "pi is about %d\n", 3.1415126;
pi is about 3

print "5 million is %e\n",5000000
5 million is 5e+6

printf "pi is closer to %f1.2\n",3.1415126;
pi is closer to 3.14
```

- a code is placed in the string to format the values coming after

printf

Format	Result
%%	A percent sign
%s	A string
%d	A signed integer (decimal)
%e	A floating point number (scientific notation)
%f	A floating point number (fixed decimal notation)
%g	A floating point number (%e or %f notation according to value size)

Filehandles

- Very useful for reading in files and sending output to a file
- if you type

```
./perl_script.pl <infile.txt >outfile.txt
```

- then your script will process `infile.txt` through the standard input `STDIN` (instead of the keyboard) and output to `outfile.txt` through standard output `STDOUT` (instead of the screen)
- error messages are sent to the standard error `STDERR`, which is the screen unless you change it

opening (and closing) a filehandle

- standard filehandles
  - `STDIN, STDOUT, STDERR`
- also possible to open your own

```
open IN, 'infile.txt';
```

#opens `infile.txt` through the `IN` filehandle

```
open OUT,'>outfile.txt';
```

#opens `outfile.txt` through the `OUT` filehandle (always makes a new file)

```
open OUT, '>>outfile.txt';
```

#opens outfile.txt through the OUT filehandle (appends to the file if it already exists)

opening (and closing) a filehandle

- you can also use variables

```
$infile="NA28334_mapped_reads.fastq.txt";
$outfile="nature_paper.txt";
open SEQUENCES, $infile;
open PAPER,">$outfile";

# filehandle names can be anything but don't use:
 STDIN STDOUT STDERR

close SEQUENCES;
# to close a filehandle (not really important)
```

testing success and die

- open might not be successful (e.g. if you spelled the filename wrong)

```
if ( open LOG , 'logfile.txt' ) {
   print "opened file\n";
} else {
   die;
}
```

or

```
open LOG, 'logfile.txt' or die "cannot open file\n";
```

reading from filehandles

- a very common way to use perl:

```
open STUDENTS,"student_list.txt" or die;
while (<STUDENTS>) {
   chomp;
   if ($_ eq "Doreen") {
      print "found Doreen in the file\n";
   }
}
```

writing to filehandles

```
my @students = qw/Doreen Oskar Elin Sangeet Malin/;
open OUTFILE,">student_list.txt" or die;
$i=1;
foreach $person (@students) {
   print OUTFILE "student $i : $person\n";
   $i++;
}
```

contents of student_list.txt:

```
student 1 : Doreen
student 2 : Oskar
student 3 : Elin
student 4 : Sangeet
student 5 : Malin
```

scalar variable filehandles

- you can also specify a filename with a variable

```
$next_data_file="NA283729.mapped.fastq.sam";
open SAM,$next_data_file;
```

- you can even use a scalar as a filehandle

```
open my $student_file,">student_list.txt";
print $student_file "Sangeet";
# note there is no comma
```

standard filehandle names

- system filehandles
  - `STDIN`
  - `STOUT`
  - `STDERR`
- others
  - `DATA`
  - `ARGV`
  - `ARGVOUT`

summary: input and output

```
while (<STDIN>) {  }
```

- to read from standard input

```
<>
```

- to process invocation commands and read in files

```
@ARGV
```

- contains the invocation commands (useful)

```
printf
```

- formats scalars (e.g. decimal points on numbers)

```
open FH,"input.txt";
```

- opens a filehandle for reading

```
open FH,">output.txt";
```

- opens a filehandle to write (>> to append)

```
while (<FH>) {  }
```

- reads each line of a filehandle

```
die
```

- your script dies!

common perl errors

Syntax errors (program will not compile)

1) You have forgotton the semicolon (`;`) in the end of a statement.
2) You wrote a reserved word with a capital letter (`If` instead of `if`, etc).
3) You began a text string with a quote `"` but did not end with `"`.
4) You do not have matching parenthesis `( )`, curly brackets `{ }` or square brackets `[ ]`.

Trivial semantic errors (program will not run right)

5) You have forgotten to use `chomp` on input from keyboard or file.
6) You used `eq ne lt le gt ge` instead of `== != < <= > >=` in a comparison or vice versa.
7) You used = instead of == in a comparison.
8) You used a variable before assigning a value to it.
9) You made a never-ending loop, because you did not increment the loop counter.