8) Control structures

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

Reminder: scalars

- variables in perl (e.g. `$example`) can be numbers or strings
- perl converts variable between the two depending on the operation
- backslash `\` used to indicate special characters like newline `\n` and tab `\t`
- compare numbers or strings with comparison operators and `if`
- `<STDIN>` gets input from the keyboard
- `chomp` removes `\n`
- `while` repeats a chunk of code while an expression is true

Reminder: arrays

`$array[0]`
- first element of `@array`

`$#array`
- is the final index of `@array`

`qw//`
- quoted by whitespace

`push pop`
- add or remove elements from the end

`shift unshift`
- add or remove elements from the start

`splice`
- add, remove, replace elements from the middle

`foreach`
- cycle through every element

`sort reverse`
- change order

`<STDIN>` can also be read into an array

Reminder : hashes

- a hash is a group of key-value pairs
- you can quickly retrieve the value for each key
- this data structure is extremely useful e.g. for
  - counting occurrences
  - associating data (such as different IDs)
- summary of syntax:

```
%hash
$hash{$key}=$value
keys %hash
values %hash
exists $hash{$key}
delete $hash{$key}
```

Reminder: subroutines

- subroutines
  - called by `&sub_example` or `sub_example()`
  - defined using `sub name { }`
  - use `return` or it will return last evaluated value
  - arguments passed within:

  `@_`

  `$_[0], $_[1], $_[2]`….

- `my` is used to restrict a variable to a block of code
- `use strict` means all variables must be declared using `my`

Reminder: 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!

Reminder: regular expressions

```
/ /   m/ /                  # pattern matching

\n \t \\                    # newline tab backslash

\d \w \s . \D \S \W         # built-in character classes

[a-z]                       # define character class

+ ? * {n}                   # quantifiers

/i /s /x /g                 # modifiers

\A \Z ^ $                   # anchors

\1 \2                       # back references

$1 $2 $& $` $'              # match variables

s/// tr///                  # substitutions

split join                  # split and join by pattern match
```

Control Structures

- so far we have looked at `while foreach if`

```
while (#this is true) {
    #loop
}

while (<STDIN>) {
    chomp ($line=$_);
}


foreach $loop_variable (@array) {
    #loop
}


if ($x eq $y) {
    #do something
} else {
    #do a different thing
}
```

unless

- the opposite of `if`

```
unless ($x == $y) {
    #do something
}
```

- the same as this:

```
if ($x != $y) {
    #do something
}
```

- and the same as this:

```
if ($x == $y) {
} else {
    #do something
}
```

- for example

```
unless (/^>/) {
    $sequence.=$_;
}
```

unless

- you can also use `else`

```
unless (/ATG/) {
   #did not find ATG
} else {
   #did find ATG
}
```

- but then it is probably better to start with `if`

```
if (/ATG/) {
   #did find ATG
} else {
   #did not find ATG
}
```

until

- the opposite of `while`

```
until ($x > 100) {
    $x++;
}
```

- the same as:

```
while ($x <= 100) {
    $x++;
}
```

`unless` and `until` are sometimes more convenient to write than `if` and `while`

different syntax

- you can also put modifiers after expressions
- this can be used to make more compact `if` and `unless` commands

```
print "matched ATG\n" if (/ATG/);

die "fatal error\n" unless $input>0;
```

- and also for `while until foreach`

```
print "$_\n" foreach @student;

$i+=10 until $i>100;
```

- can be used to make code easier to read and write

naked block

- you can put code inside curly brackets
- this will just restrict the my variables to this block

```
{

    my $private_variable = "secret";
    print "$private_variable\n";

}
```

elsif

- useful if you want to evaluate a number of possibilities

```
if (/^G/) {
   #matches G at the beginning
} elsif (/G/) {
   #matches G (but not at the beginning)
} elsif (/\w/) {
   #contains a word character but not G
} else {
   #does not contain any word character
}
```

++ and --

- it is very common to count things with ++
- you often want to count number of times you did a loop

```
@frequencyA=(0.1, 0.4, 0.5, 0.4, 0.1, 0.2, 0.5);
@frequencyB=(0.2, 0.3, 0.5, 0.1, 0.7, 0.1, 0.3);

$i=0;
$j=0;
foreach (@frequencyA) {
   $j++ if $frequencyA[$i]>$frequencyB[$i];
   $i++;
}
```

- `$i` is the loop count
- `$j` counts how many in `@frequencyA` are bigger than `@frequencyB`

++ and --

- it is very common to count things with ++
- you often want to count number of times you did a loop

```
$i=0;
foreach $word (@words) {
   $wordcount{$word}++;
   $i++;
}

$j=0;
foreach $word (sort keys %wordcount) {
   print "$word appears $wordcount{$word} times\n";
   $j++;
}

print "there were a total of $i words and a total of
$j different words\n";
```

`for` loops

```
for (initialization; test; increment) {
   #do something
}
```

- for example:

```
for ($i=1; $i<=100; $i++) {
   print "I have counted $i out of 100\n";
}
```

- which is a neater way of writing this:

```
$i=1;
while ($i<=100) {
   print "I have counted $i out of 100\n";
   $i++;
}
```

loop controls: `last`, `next` and `redo`

```
while (<STDIN>) {
    chomp;
    if ($_ eq $password) {
        print "you typed the password\n";
        last;
    }
}

while (<SEQFILE>) {
    chomp;
    next unless /^>/;
    push @seq_names, $_;
}
```

loop controls: `last`, `next` and `redo`

- `redo`

```
foreach (1..10) {
    print "counted to $_, shall I continue?\n";
    chomp ($line=<STDIN>);
    redo unless $line=~/yes/;
}
```

an infinite loop

- this is the normal way to make one

```
while (1) {
    $i++;
    last if $i>100;
}
```

- this does the same thing

```
for (;;) {
    $i++;
    last if $i>100;
}
```

logical operators

`&&` means AND
`||` means OR

```
foreach $base (@bases) {
   if (($base eq 'G') || ($base eq 'C')) {
      $gc_count++;
   } elsif (($base eq 'A') || ($base eq 'T')) {
      $at_count++;
   } else {
       die "$base is not a base!\n";
   }
}
```

logical operators

`&&` means AND
`||` means OR

```
if (($i == 10) && ($j == 10)) {
   print "i and j both equal 10\n";
} elsif (($i == 10) || ($j == 10)) {
   print "either i or j equals 10\n";
} else {
   print "neither i nor j equals 10\n";
}
```

summary: control structures

- loops:

```
while (test) { }
until (test) { }  # opposite of while
for (initialize; test; increment) { }
foreach $loop_variable (list) { }
```

- loop controls:

```
last  # jumps out of loop
next  # begin next iteration
redo  # redo current iteration
```

- conditional control structures

```
if (test) { }
unless (test) { }        # opposite of  if
} else {
} elsif (test) {
```

- logical operators

```
&&  ||      # AND  OR
```

- autoincrement autodecrement

```
++ --
```