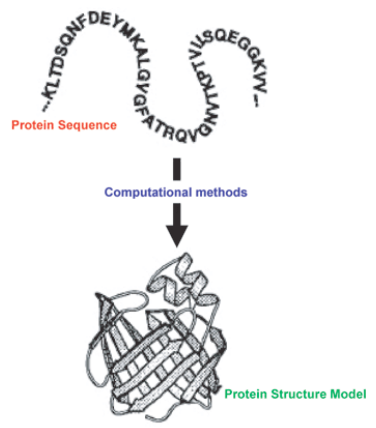
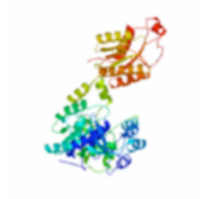
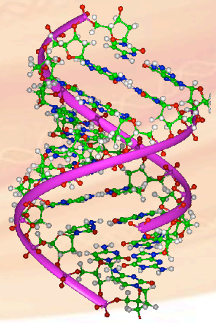




Bioinformatics Computational Methods 1 - BIOL 6308



September 24th 2013

http://155.33.203.128/cleslin/home/teaching6308F2013.php

Last Time

- Formatting Strings
- Perl Statements
- Scalars
- Arrays
- Hashes
- Operators
  - Assignment
  - Comparison
    - String and Number
  - Logical
- Control Structures
  - Loops
  - Decisions
- Open and Closing Files or input
  - filehandles
- `chomp` and `die`
- Perl's default variable `$_`, very important to understand
- Example Scripts
- Questions?

Example Scripts For This Lecture

- Scripts and data
  - Can be found in `/data/METHODS/Fall/LECT5/`

Default Variable (cont'd)

```
>mutated_sequence
```

```
#!/usr/bin/perl
use strict;
use warnings;
use feature qw(say);
# set hash
my %stuff = ( phone => "iPhone",
              desktop => "Dell",
              laptop => "Macbook Pro");

# iterate over array
foreach (keys %stuff ) {
   say;
}
```

```
laptop
Desktop
phone
```

```
#!/usr/bin/perl
use strict;
use warnings;
use feature qw(say);
#create a variable to store the datafile

my $infile = 'sequences.fasta';
unless (open(INFILE, "<", $infile) ){
   die "Can't open file: " , $infile , " " , $!;

}
while (<INFILE>){
    chomp; # using $_
    if ($_ =~ /^>/){
       say $_;
    }
    else{
       #could print here if I wanted the sequence data
    }

}
close INFILE
```

```
#!/usr/bin/perl
use strict;
use warnings;
use feature qw(say);
print "Enter another value: ";
while (<STDIN>) {
     chomp;
     say "You entered " , $_ ,
        " which is " , length , " long";
     say;
     print "Enter another value: ";;
}
```

control-c get out of loop

```
Enter another value: A
You entered A which is 1 long
A
Enter another value: Here
You entered Here which is 4 long
Here
Enter another value: Biology
You entered Biology which is 7 long
Biology
Enter another value: ^C
```

What's My Line Separator

- When you open file for reading, how are lines processed?
  - By default the "`\n`" character is what separates each line:
  - Means Perl is breaking up the input by "`\n`" newline characters

```
while(<INFILE>){
          ##DO SOMETHING
}
```

- But it doesn't have to be that way
- Special character - you can use to change record separator

```
$/ = "";  ##what will this do?
```

- You can set this to what you want to get chunks of data at a time, instead of individual lines

See testScriptFH2.pl

Case Example - Line Separator

- You have to implement a way to read in a fasta file in your Perl script
- You know the usual construction:
  - `open` with `filehandles`
  - `while(<INFILE>)`
- Default behavior of Perl
  - Handle things **one line** at a time
    - why might that be a problem?
  - Leads to loops that sometimes have to deal with the results of the **previous iteration** before doing new work
    - logic can be more difficult
    - iterators and temporary values
- Modules and BioPerl are widely available to just read this file, but remember, you're a new Perl programmer
- Any ideas?

./testScriptFasta.pl Output

- Use the record separator, `$/`
- Try using this statement in your declarations:
  - `$/ = "\n>";`

```
1 >mutated_sequence
MEFFGESWKKHLSGEFGKPYFIKLMGFVAEERKHYTVYPPPHQVFTGTQMCDIKDVKVVILGQNPYHGPNQA
HGLCFSVQRPVPPPPSLENIYKELSTDIEDFVHPGHGDLSGWAKQGVLLLNAVLTVRAHQANSHKERGWEQF
TDAVVSWLNQNSNGLVFLLWGSYAQKKGSAIDRKRHHVLQTAGPSPRSVYGGFFGCRHFSKTNELLQKSGKK
PIDWKEL
2 gi|5524211|gb|AAD44166.1| cytochrome b [Elephas maximus maximus]
LCLYTHIGRNIYYGSYLYSETWNTGIMLLLITMATAFMGYVLPWGQMSFWGATVITNLFSAIPYIGTNLV
EWIWGGFSVDKATLNRFFAFHFILPFTMVALAGVHLTFLHETGSNNPLGLTSDSDKIPFHPYYTIKDFLG
LLILILLLLLLALLSPDMLGDPDNHMPADPLNTPLHIKPEWYFLFAYAILRSVPNKLGGVLALFLSIVIL
GLMPFLHTSKHRSMMLRPLSQALFWTLTMDLLTLTWIGSQPVEYPYTIIGQMASILYFSIILAFLP
```

Notice Anything?

>

Writing to Files

- Of course we could....

```
$ ./test.pl >our_output.txt
```

- But we're beginning to implement real programs, so we'll need a way to open an outfile

```
my $outfile = 'codingGenes.fasta';
unless (open(OUTFILE, ">", $outfile) ){
    die "can't open " , $outfile , " for writing " , $!;
}
```

- We can use > or >>, what's the difference?
- Remember that writing to files will delete old file if use the ">"
  - Of course you need permissions to the file
  - Might want to give user the option to provide output name, instead of hard coding

FYI - Binmode

- Writing to a file doesn't necessarily mean text or strings
  - You can write to files in binary format
- When trying to write an:
  - Image, a sound file or an executable,
  - You'll need to write it in binary because some systems will translate all newline feeds \n into carriage returns /r/n.
- Files that are not in binary have CR LF sequences converted to LF on input, and LF to CR LF on output
- This is vital for operating systems that use two characters to separate lines within text files (MS-DOS)

- To prevent some systems from automatically rewriting this, we use binmode

```
binmode FILEHANDLE;
```

http://perldoc.perl.org/functions/binmode.html

cmp and <=>

- testScript5.pl**
- Sorting an array is easy in Perl – uses special variable `$a` and `$b`
- This why you should **never** use `$a` and `$b` in your scripts

  - To sort an array in **ASCII** (Alphabetical) order :

    ```
    sort(@array); #just doing this, lose the sort and get warning
    ```
  - Example of storing a sort result back into the same array :

    ```
    @array = sort (@array); #store the data
    ```
  - To sort an array in **reverse ASCII** (Alphabetical) order :

    ```
    sort {$b cmp $a} (@array);
    ```
    - OR

    ```
    @days = sort(@days); #sort
    @days = reverse(@days); #reverse the order
    ```
  - To sort an array in **numeric** ascending order :

    ```
    @array = sort {$a <=> $b} (@array);
    ```
  - To sort an array in **numeric** descending order :

    ```
    @array = sort {$b <=> $a} (@array);
    ```

** This scripts can be found in `/data/METHODS/Fall/LECT4`

Arrays - Easy String Sorting

```perl
use strict;
use warnings;
use feature qw(say);
my @input = (
    "Hello World!",
    "You're all I need.",
    "to be or not to be",
    "There's more than one way to do it.",
    "Absolutely Fabulous",
    "Give me liberty or give me death.",
    "Linux - Because software problems should not cost money",
);
# Do a case-sensitive reverse ASCII sort
my @sorted = (sort { $b cmp $a } @input);

foreach my $ele (@sorted){
    say $ele;
}
```

```
to be or not to be
You're all I need.
There's more than one way to do it.
Linux - Because software problems should not cost money
Hello World!
Give me liberty or give me death.
Absolutely Fabulous
```

sort1a.pl

Arrays - Easy String Sorting

```
use strict;
use warnings;
use feature qw(say);
my @input = (
    "Hello World!",
    "You're all I need.",
    "to be or not to be",
    "There's more than one way to do it.",
    "Absolutely Fabulous",
    "Give me liberty or give me death.",
    "Linux - Because software problems should not cost money",
);
# Do a case-insensitive sort
my @sorted = (sort { lc($b) cmp lc($a); } @input);

foreach my $ele (@sorted){
    say $ele;
}
```

```
You're all I need.
to be or not to be
There's more than one way to do it.
Linux - Because software problems should not cost money
Hello World!
Give me liberty or give me death.
Absolutely Fabulous
```

sort1.pl

Arrays - Numerical Sorting...?

```
#!/usr/bin/perl
use strict;
use warnings;

my @array = (100,5,8,92,-7,34,29,58,8,10,24);
my @sorted = (sort @array);
foreach my $ele (@sorted){
    print $ele , ",";
}
print "\n";
```

Will this work?

```
-7,10,100,24,29,34,5,58,8,8,92,
```

That's probably not what we wanted

Does anyone know what's going to happen here?

sort2.pl

Arrays - Easy Numerical Sorting, the Correct Way

```
#!/usr/bin/perl
use strict;
use warnings;

my @array = (100,5,8,92,-7,34,29,58,8,10,24);
my @sorted = (sort { $a <=> $b } @array);
foreach my $ele (@sorted){
    print $ele , ",";
}
print "\n";
```

```
-7,5,8,8,10,24,29,34,58,92,100,
```

sort3.pl

Hashes - Easy Sorting on a **Key**

```
#!/usr/bin/perl
use strict;
use warnings;
use feature qw(say);

my %grades = (
        jim   => 90,
        ted   => 75,
        april => 96,
        nancy => 55,
        john  => 76,
);
##go over each key
foreach my $key (sort (keys %grades) ){
        say $key , " " , $grades{$key};
}
```

```
april 96
jim 90
john 76
nancy 55
ted 75
```

But what if we wanted to sort by actual grades?

sort4.pl

Hashes - Easy Sorting on a **Value**

```perl
#!/usr/bin/perl
use strict;
use warnings;
use feature qw(say);

my %grades = (
        jim   => 90,
        ted   => 75,
        april => 96,
        nancy => 55,
        john  => 76,
);
##go over each key and sort by the value
##using an inline Perl function
foreach my $key (sort{ $grades{$b} <=> $grades{$a} } (keys %grades) ){
        say $key , " " , $grades{$key};
}
```

```
april 96
jim 90
john 76
ted 75
nancy 55
```

sort5.pl

Now that's better

Defensive Programming

- So far we know how to:
  - Get data from user, Read and write files, calculations, etc
  - For small programs this is adequate
- Might want to use files in context of larger applications:
  - Check their status before we try and open them
    - If necessary, take preventative measure
      - Maybe warn user if a file we wish to overwrite exists
      - Check to make sure we don't read a directory as if it was a file
  - Check make sure an `$ARGV` was given
- This sort of programming is called defensive programming
  - Anticipating the consequences of future actions
  - Assume everything is out to get you
- Properly anticipating, diagnosing, and working around these areas is the **mark of a strong programmer**
- Perl has built in ways to test for this (see next slide)

File tests

-r	**File or directory is readable by this (effective) user or group**
-w	**File or directory is writable by this (effective) user or group**
-x	**File or directory is executable by this (effective) user or group**
-e	**File or directory name exists**
-z	**File exists and has zero size (always false for directories)**
-s	**File or directory exists and has nonzero size (the value is the size in bytes)**
-f	**Entry is a plain file**

- Lets Look at testScriptFH3.pl

Subroutines

Subroutines - Getting Efficient with Perl

- Thus far, we been dealing with code that has been sequential

```
do this;
 do that;
  now do something;
   finish;
```

- That's a good start!
  - But there's problem:
    - You need something to be done over and over, perhaps slightly different depending on the context
      - i.e. translate a DNA sequence in different reading frames
      - Do you want to do this 6 times?
    - Say you do this six times, and then find a mistake
      - What's the problem?
  - Solution:
    - Put the code in a subroutine and call the subroutine when you need it!

Basics Behind a Subroutines

- Now we'll need to begin to discuss breaking our code up into smaller, manageable pieces
- Subroutines allow you to:
  - Input variables
  - Return variables
- These can be placed anywhere in your Perl program
- For maintainability, you can put them the end of your script
- Allow you to reduce code duplication
- General rule:
  - If you do something more then once, write a subroutine

Why Use Subroutines

- Subroutines give us the ability to give a name to a section of code
  - Allows us to break up our longer sections
  - When we need that code, we just call it by name
- Helps our programming 4 main reasons:
  - Reuse code
  - Easier fix bugs and faster for us to write programs
  - Organize our code into sections
  - Each sub can be responsible for a particular task

Case Example

- Imagine:
  - Have 3 dna sequences
  - Need to apply substitution to each of them:

```
# trim leading/trailing whitespace
$dna1 =~ s/^\s*(.*?)\s*$/$1/;
$dna2 =~ s/^\s*(.*?)\s*$/$1/;
$dna3 =~ s/^\s*(.*?)\s*$/$1/;
```

  - Not bad for a rookie
  - But, Increasingly cumbersome the longer your script becomes
  - Why?

Case Example

- Next, add several more variables
  - Which must have the exact same substitution applied
  - Do you search for the original code, then copy and paste it again and again?
    - Not very efficient
    - Greater possibilities for error
      - Could copy something incorrectly
      - Accidentally edit the code in a way contrary to your intention

```
# trim leading/trailing whitespace
$dna1 =~ s/^\s*(.*?)\s*$/$1/;
$dna2 =~ s/^\s*(.*?)\s*$/$1/;
$dna3 =~ s/^\s*(.*?)\s*$/$1/;
```

Case Example

- Consider the ongoing **maintenance** requirements for the code
  - Initial substitution proves to be: Faulty, Incomplete or Simply need extension
  - Maintenance will require you to find and re-edit each instance of the code to correct the problem
  - Does this sound good?

```
# trim leading/trailing whitespace
$dna1 =~ s/^\s*(.*?)\s*$/$1/;
$dna2 =~ s/^\s*(.*?)\s*$/$1/;
$dna3 =~ s/^\s*(.*?)\s*$/$1/;
```

So When Should I Use Subroutine in Perl?

- Probably two cases when a piece of code should be put into a subroutine
  - 1st
    - When you know it will be used to perform a calculation
    - When you know the action is going to happen more then once
      - Putting string into a specific format
      - Printing the header or footer of a report
      - Etc.
  - 2nd
    - **Logical units** of your program you want to **break up** to make your program easier to understand
    - Much easier to make changes
    - Easier to use elsewhere

When You Define and Use Subroutines

- Code logic is neatly contained in a single location of the script
- Can be called (and later modified, when necessary) without requiring you to track down every instance of the logic in the script at large
- Using a powerful programming tool
  - Syntax of most programming languages includes support for writing and using them
  - Judicious use of subroutines will often substantially reduce cost of developing and maintaining a large program, while increasing its quality and reliability

- So how is it done in Perl?

Declaring Subroutines in Perl

```
# main program
.
.

# subroutines defined here

sub sub1 {
...
}
sub sub2 {
...
}
sub sub3 {
...
}
```

Subroutines can be placed anywhere in the program but best to group them **at the end**

Defining a Subroutine - Its Easy!

```
sub test {
        ...
}
```

- Three sections:
  - Keyword sub (case-sensitive)
  - Name giving it
  - Block of code delimited by curly brackets

Naming a Subroutine

- Name of a subroutine is another **Perl identifier** (letters, digits, and underscores, but can't start with a digit)
- If the subroutine is:
  - Primarily about performing an activity
    - Name them with the verb first:
    - `summarizeData` or `downloadProtein`
  - Primarily about returning information
    - Name them after what they return:
    - `getGreeting` or `getHeader`
  - About testing whether a statement is true or not use **is:**
    - `isValid  isDna  isProtein`
  - Converting one thing into another, try to convey both things, use 2:
    - `text2htm` or `meters2feet`

Passing Parameters (Arguments) To Subroutines

- Pass **parameters** by placing them between parentheses

```
printProductTwoNumbers(10,15);
```

- What happens to the parameters?
  - End up in Perl's special variable, the array:

    ```
    @_
    ```
  - So we can get to these parameters:

We've seen some similar to this before?

```
sub printProductTwoNumbers{
    my ($firstNumber, $secondNumber) = @_;
    my $product = $firstNumber * $secondNumber;
    say $product;
}
```

Let's take a look at testScript14.pl

Return Values

- Sometimes want more than printing - may want to return a result
  - Here we can test if our call was successful

    `return(0);` or `return ('T');`
  - Return the summed value or a value from some other calculation
  - Return a translated dna sequence
- Two ways of returning things in Perl
  - Implicitly or explicitly
  - Use the **explicit** way:
    - Easier to understand
    - Forces you realize what you are returning
    - With Perl you can return multiple variables

      `return($sum1, $sum2);`

```
sub getProductOfTwoNumbers{
        my ($firstNumber, $secondNumber) = @_;
        my $product = $firstNumber * $secondNumber;
        return($product);
}
```

Return Values

- Can also return multiple scalars, arrays, etc.
- But just like the input everything ends up in a single array or list:

```
@DNA = read_file($filename);
.
.
($nG,$nC,$nT,$nA) = getCountBases($dna);
sub getCountBases {
        ...
        return (@results);
} # end sub
```

note (@)
for the list

Let's look at testScriptSub1.pl

Understanding Scope

- Remember when we declare a variable with `my`
- Perl has two types of Scope
  - Global - can be accessed anywhere in the program
  - Lexical
- We begin programming in a package called **main**

  ```
  my $dnaSeq;
  ```

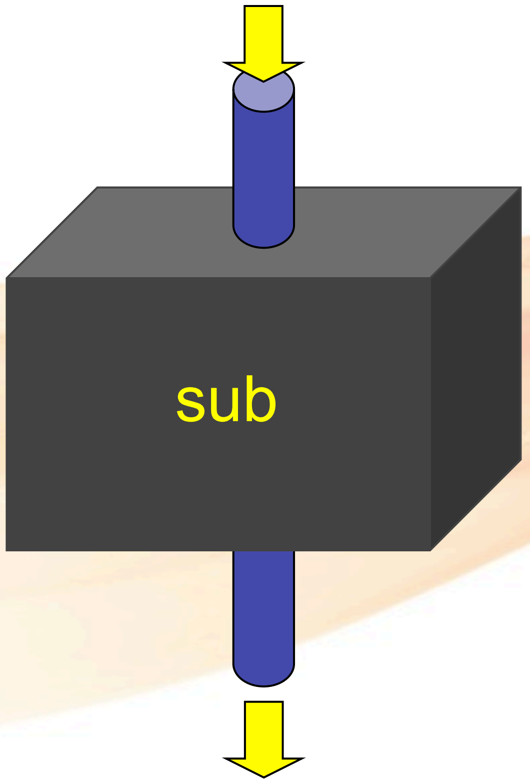
  - Perl sees this as my `$main::dnaSeq`
  - Since we're in main package, we can just use `$dnaSeq`
    - Like the phone system
    - Don't have to dial the area code when you call someone in the same region
- Now inside a subroutine
  - Variables declared with `my` have their own scope
  - They can only be seen w/in the sub
    - Lexical

The Need for Variable Scoping

A well implemented subroutine should work like a **black box**, apart from well-defined inputs/outputs it should not affect the rest of the program

Apart from input/output all vars needed by the sub should appear and disappear within the sub

Allows us also to use the same names for vars outside and inside the sub without conflict



input

output

Variable Scoping in Perl

- By default, all variables defined outside a sub are visible within it – all variables outside a subroutine are **global – is this good?**
- Therefore the sub can change variables used outside the **sub**
- This is where you can get yourself into trouble

Solution:

Restrict the *scope* of the variables by making them **local** to the subroutine

Eliminate the risk of altering a variable present outside the **sub**. Also makes it clear what the subroutine needs to function.

How? use **`my`**

.

Variable Scoping in Perl

In Perl, variables are made local to a subroutine (or a block) using the **my** keyword. For example:

```
sub testSub{
        my ($readingFrame) = @_;
        my $variable1;  # simple declaration
        my $dna = "GGTTCACCACCTG"; # with initialization
        my ($seq1,$seq2,$seq3); # more than 1
}
```

`$readingFrame, $variable1, $dna, $seq1, $seq2, and $seq3` can only be seen by `sub testSub`

you could have `$dna` in main, and it would **not** interfere with `$dna` inside the `testSub` subroutine

Call By Value

Consider

```
my $i = 5;
my $j = add100($i);
say "In Main i = ", $i;

sub add100 {

   my $i = @_;
   $i = $i + 100;
   say "In sub i = ", $i;
   return ($i);

}
```

← `$i` here not changed by the subroutine

Remember Context! `@_`

Any ideas of the output from above?

```
In sub i = 101
In Main i = 5
```

Why not 105?

This is called ***Call by Value*** (pass by Value) because a copy is made of the parameter passed and whatever happens to this copy in the subroutine doesn't affect the variable in the main program

But how could you change the actual $i? You could use a reference

Passing More Complex Parameters

- Passing scalars is easy
- What if you want to pass an array
  - Can you pass an array?????

```
mySub(@arr); #ok here
checkSame(@a, @b); #going to get in trouble
```

- When you pass an array inside a list, the list collapses
  - The original structure of the array is lost
  - Even before we put anything in the parameter array @_
- To get around this we use references

See testScriptSub2.pl

Overview of Subroutines

- Subroutines are a block of code that should carry out **ONE** particular task
- Best to keep it to **ONE** task
  - Get in the habit of implementing subroutines **that do one task**
    - Don't do multiple things in a subroutine
    - Try limit to the task at hand
  - This will help you become a more effective programmer
  - Becomes easier to add new functionality later on
- They can be called as many times as required within your script

Benefits of Implementing Subroutines

- Increased functionality, easy to go in and create changes
- Saves typing code over and over
  - Make a change once, instead of every time you did the cut & paste
  - Results in maintainable code
- Decrease the chance of error creeping into the code (decrease code smell as well)
- 1st step in creating code that can be reused and even packaged up
- Values passed to subroutine held in `@_`

5 minute break…

Passing More Complex Parameters into a Subroutine

References in a Subroutine

- Passing scalars is easy
- What if you want to pass an array
  - When you pass an array inside a list, the list collapses
  - The original structure of the array is lost
    - Even before we put anything in the parameter array @_
    - This is why you cannot implement like:

      `areArraysTheSame(@a, @b)`
- To get around this we use references - **Call by reference**

Also known as pass by reference

first a scalar

```
#!/usr/bin/perl
use strict;
use warnings;

my $a = 5;
increment(\$a);
print "$a\n";

sub increment{
    my ($refScalar) = @_;
    $$refScalar++;
}
```

pass reference

dereference

6

What is a Reference?

- Simple definition:
  - Piece of data that tells us the location of another piece of data
  - example: See the first paragraph on page 130
    - Reference to the text on that page
    - Not the actual data, but where to find it!
    - Immediately use the data despite it being somewhere else
- This is why references are so useful
  - Specify data once, then access it from wherever else we are
  - Allow us to pass arrays and hashes to our subroutines correctly b/c we can keep their structure

```
areArraysTheSame(\@a, \@b)
```

passing array reference directly

or

What's a reference look like?

```
my $refArr1 = \@a;
my $refArr2 = \@b;
areArraysTheSame($refArr1, $refArr2);
```

References - Simple, Nothing Special

- A reference is always a scalar

```
   my $refScalar = \$dna;  #get a reference to a scalar
```

- The data the reference refers does not have to be a scalar

```
   my $refArr = \@dna;  #get a reference to array
```

- Languages like C and C++ have a similar feature, called pointers
  - References similar, but pointers are nastier!
  - References store memory locations for specific clearly defined data structures
  - Don’t have to worry about memory management
    - no malloc, no new

Two Ways to Create Reference

- There are two ways to create a reference
  - 1st - You've already got the data in a variable
  - *2nd - You want to use anonymous data to go straight to a reference*
- **1st way:**

```
my @arr = (1, 2, 3, 4, 5, 6);
my $refArr = \@arr;
my $scalar = 42;
my $refScalar = \$scalar;
my %hash = (apple => ”pomme”, pear => ”poire”);
my $refHash = \%hash;
```

- Then treat the references just like scalars:

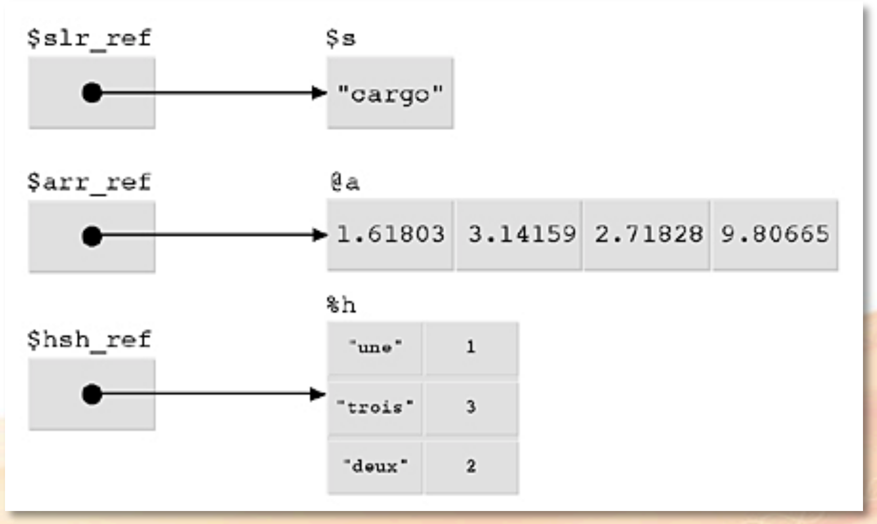
```
my @ref = ($refArr, $refScalar, $refHash);
```

- See testScriptRef1.pl

FYI - Anonymous References

- **Not Covered in 6308**
- **2nd way:**
  - To do all this w/o having to go through the interim stages of creating the variables
- Anonymous references will let us go straight from our raw data to a reference
  - Rules:
    - Get array reference use square brackets `[ ]` instead of parentheses
    - Get hash reference use curly braces `{ }` instead of parentheses
- Anonymous data structures are out of the scope of this class
  - More taught on this subject matter in BIOL6200

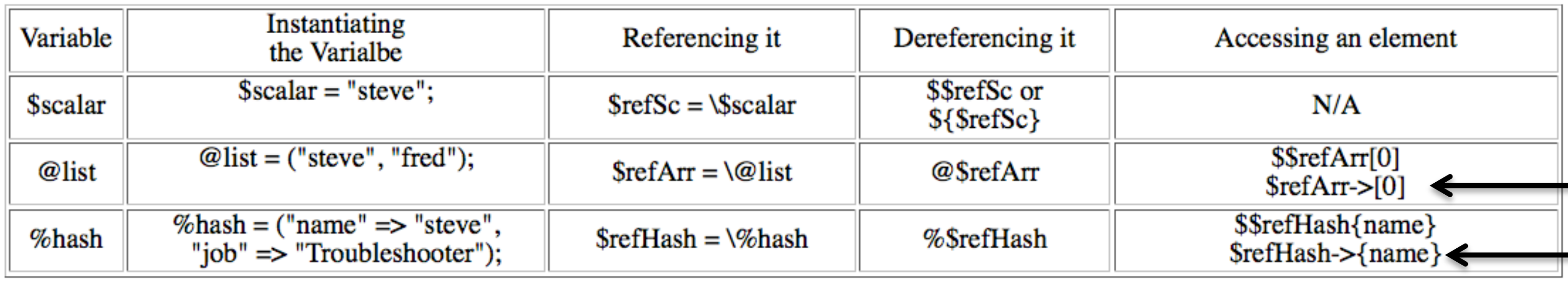
What it Looks Like



Using References

- Once you've created a reference, you'll want to use it:
  - Access the data
    - Operation of getting data back from a reference is called **dereferencing**
    - **In order to use the data - we have to dereference**
    - There are different options in Perl
  - See testScriptRef2.pl for arrays
  - See testScriptRef3.pl for hashes

Dereferencing the Variable Types



Variable	Instantiating the Varialbe	Referencing it	Dereferencing it	Accessing an element
$scalar	$scalar = "steve";	$refSc = \$scalar	$$refSc or ${$refSc}	N/A
@list	@list = ("steve", "fred");	$refArr = \@list	@$refArr	$$refArr[0] $refArr->[0] ←
%hash	%hash = ("name" => "steve", "job" => "Troubleshooter");	$refHash = \%hash	%$refHash	$$refHash{name} $refHash->{name} ←

← Use Second Form

Passing Reference to a Subroutine

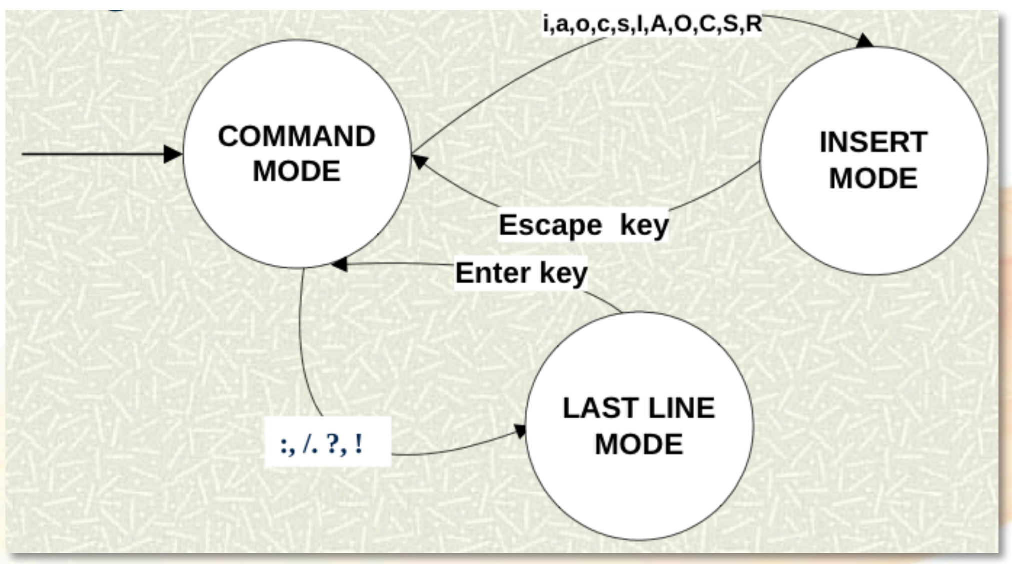
```perl
#! /usr/bin/perl
use warnings;
use strict;
use feature qw(say);
my @DNA1 = qw(A G C T A G G G C C A T T T A C G G T);
my @DNA2 = qw(A G C T A G G G C C T T T T A C G G T);

my $refArray1 = \@DNA1; #create a reference
my $refArray2 = \@DNA2;

compareArrays($refArray1, $refArray2);

sub compareArrays{
    my ($refArr1, $refArr2) = @_;
    my $size = @$refArr1; #dereference, to get the size, this look familar?
    for(my $i = 0; $i < $size; $i++){
        my $val1 = $refArr1->[$i]; #dereference to get the element
        my $val2 = $refArr2->[$i];
        if ($val1 eq $val2){

        }
        else{
            say "At index " , $i , " there is a difference " ,
                 $val1 , " " , $val2;
        }
    }#fore
}
```

```
At index 10 there is a difference A T
```

Getting More Out of `vim` with Less

A Conceptual Model of `vim`

- This is a picture of the way that `vim` works
- The arrows from one state to another are labeled by the keys that cause its state to change



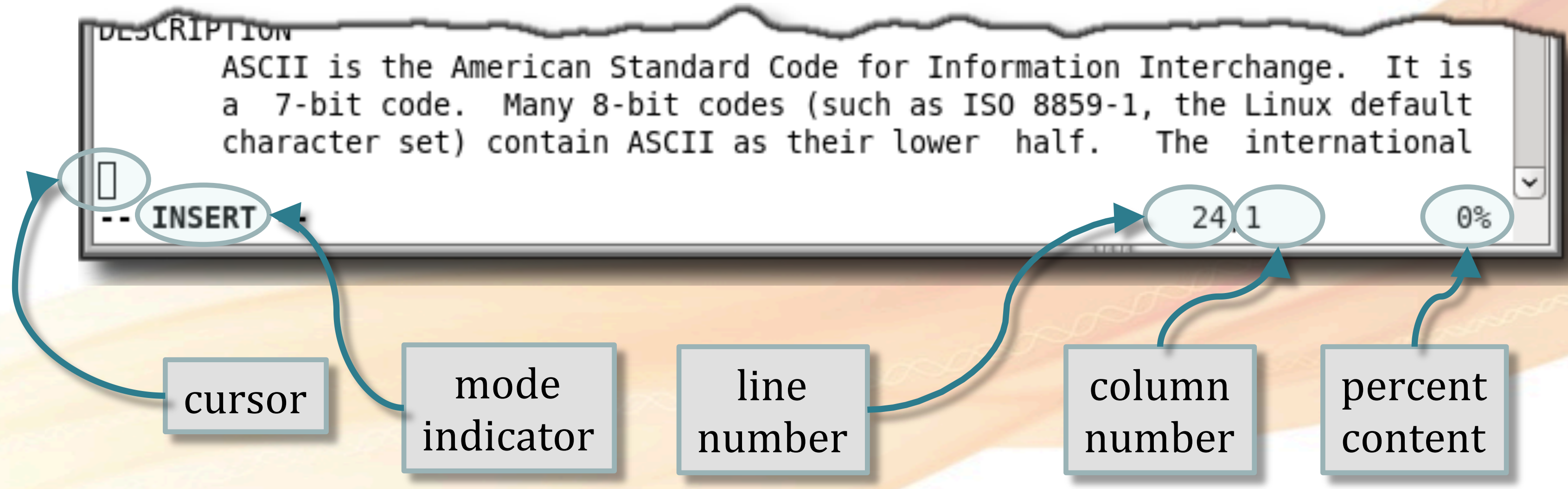
Command Mode is used to issue vim commands

Insert Mode is used to type in text.

Switch back and forth between two modes:
`i`
`<Esc>`

vim: Modes of Operation

- Command Mode: Accepts vim commands
- Input Mode: Accepts any text



```
DESCRIPTION
       ASCII is the American Standard Code for Information Interchange.  It is
       a  7-bit code.  Many 8-bit codes (such as ISO 8859-1, the Linux default
       character set) contain ASCII as their lower  half.   The  international

-- INSERT --                                                  24,1          0%
```

- Escape Key: Toggles between Command and Input Modes

Fred R. McClurg

Insert Mode

- In Insert Mode:
  - Everything you type is inserted at the text insertion point, until you type <ESC>
  - Very little you need to learn about Insert Mode
- Easy to tell when `vim` is in Insert Mode:
  - Will display a line at the bottom of the screen with the word

```
-- INSERT --
```

Last-Line Mode

- Last-Line Mode - odd name – when `vim` is in that mode, there is a prompt on the **last line** of your screen
- Last-Line Mode is entered only from Command Mode, and only by pressing either '`:`', '`/`' or '`?`'
- '`/`' and '`?`' tell `vim` to search in the file for the regular expression that you provide after it followed by a <CR>character
  - The first searches forward and the second, backward

Last-Line Mode

- The ':' tells `vim` to expect a command
- There are several types of commands, such as those to
  - Read a file
  - Write the current contents to a file
  - Do multiple-line operations such as replacements, deletions, copying to buffers
  - Set line markers or view marked lines
  - View special characters in lines
  - Run a shell command in a subshell

Command Mode

- Only way to know you're in Command Mode is to realize that it is not in the other modes
- If you do not see the word "INSERT" and you do not see the prompt for Last-Line mode (`:`,`/`,`?`), then it is in **Command Mode**
- In Command Mode
  - Navigate through the file
  - Make changes within single lines
  - Copy parts of lines or paste into them
- Considered base camp, so it is from here that you can enter Insert Mode or Last-Line Mode

The Most Important Commands

- NAVIGATION:
  - Commands to move the cursor left, down, up, right are `h` ,`j`, `k`, and `l`
  - The arrow keys usually work as well
  - Other keys move the cursor to various places
    - such as: `^ $ ( ) { } w e b`
- DELETION:
  - Commands to delete words or characters include `x` and `d`
  - Whole lines are deleted with `dd`
- SUBSTITUTION:
  - Substitutions are made by `s/pattern/replacement/` within a line
- SEARCHING:
  - Searches within a line use `f` or `t` or in the file by entering Last-Line Mode with / or ?.

Insertion Command

- Commands let you enter Insert Mode in various ways:
  - `i`   insert to the left of insertion point
  - `a`   insert to the right of insertion point
  - `o`   insert a line below current line
  - `O`   insert a line above current line
  - `I`   insert at beginning of line
  - `A`   insert at end of line

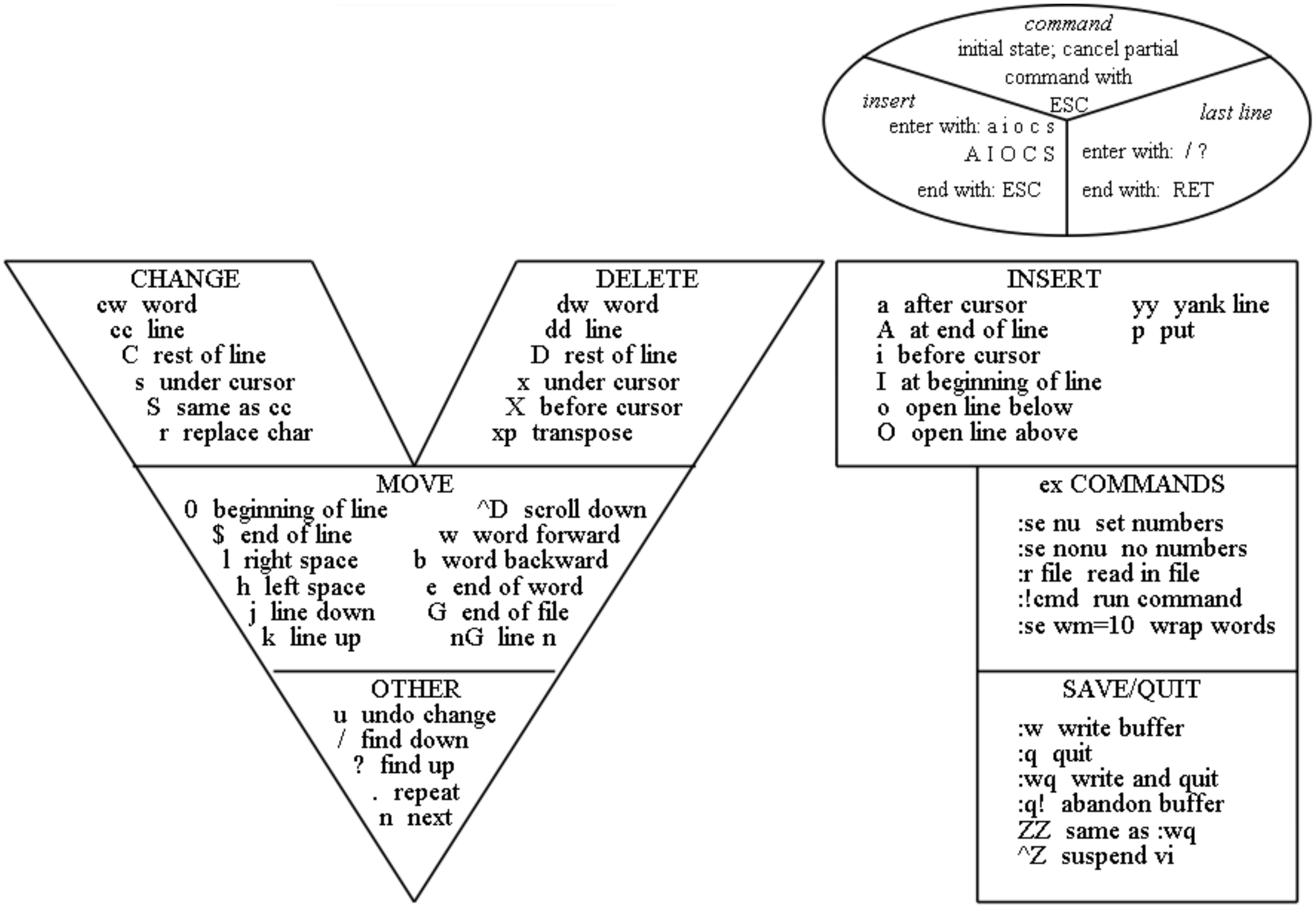
The Buffer

- `vim` has a buffer that it uses in the same way that Windows applications use the clipboard
- Lets you copy text into buffer and paste text from buffer into specific points in the file
- Makes a copy of text deleted with the delete command (`d` or `D`) in the buffer, so that it can be pasted

Command Repetition

- Most commands can be preceded by a positive integer to affect their behavior in some way
- For example:
  - Command `w` advances to next word in the file from cursor, and 5`w` advances to the 5th word from the cursor
- `a` command followed by text and then `<ESC>` adds the text to the right of the insertion point, so `100a#<ESC>` adds the string #100 times:

```
####################################################################################################
```


command
initial state; cancel partial command with ESC
insert
enter with: a i o c s A I O C S
end with: ESC
last line
enter with: / ?
end with: RET

CHANGE

cw	word
cc	line
C	rest of line
s	under cursor
S	same as cc
r	replace char

DELETE

dw	word
dd	line
D	rest of line
x	under cursor
X	before cursor
xp	transpose

INSERT

a	after cursor
A	at end of line
i	before cursor
I	at beginning of line
o	open line below
O	open line above
yy	yank line
p	put

MOVE

0	beginning of line
$	end of line
l	right space
h	left space
j	line down
k	line up
^D	scroll down
w	word forward
b	word backward
e	end of word
G	end of file
nG	line n

ex COMMANDS

:se nu	set numbers
:se nonu	no numbers
:r file	read in file
:!cmd	run command
:se wm=10	wrap words

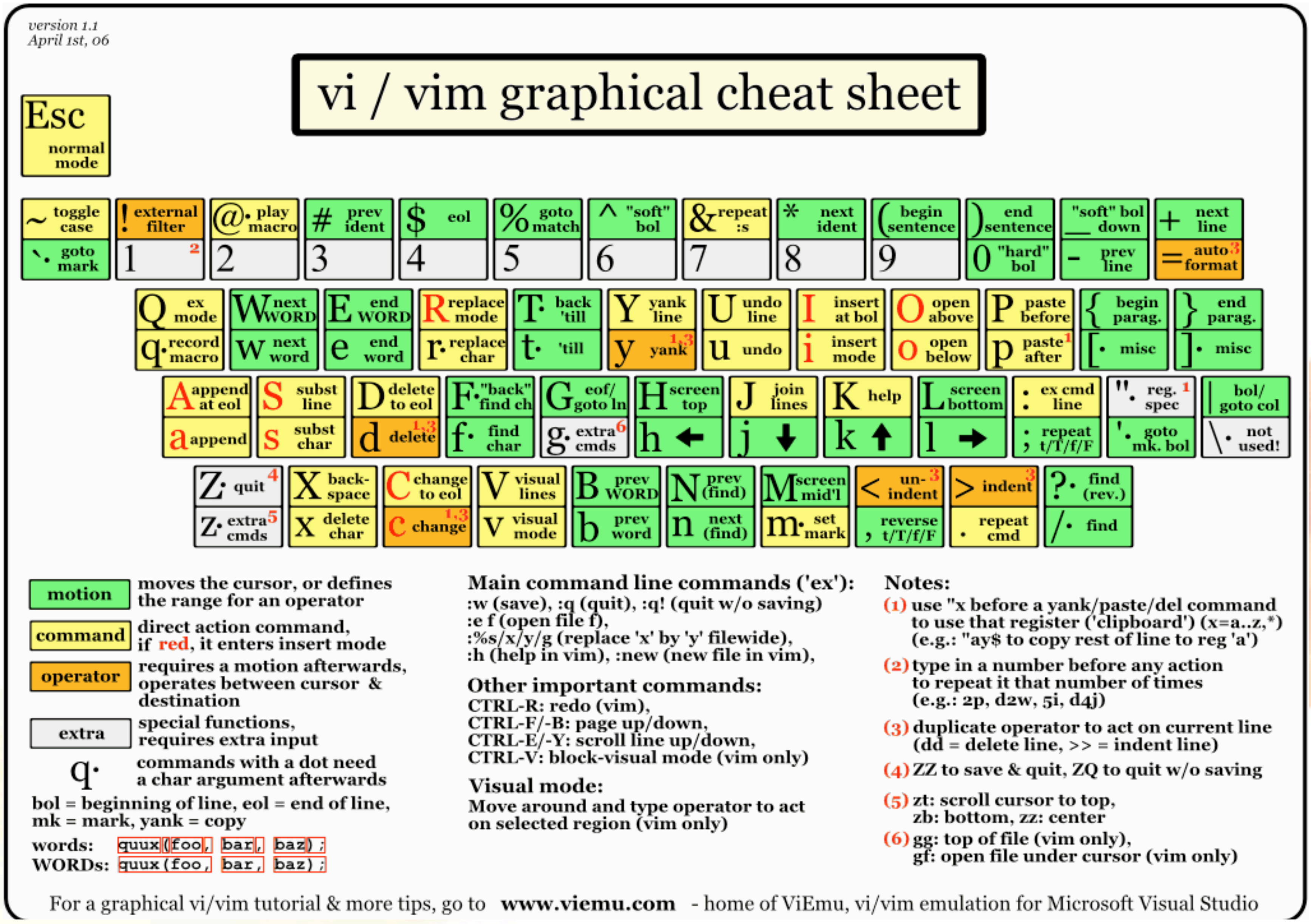
OTHER

u	undo change
/	find down
?	find up
.	repeat
n	next

SAVE/QUIT

:w	write buffer
:q	quit
:wq	write and quit
:q!	abandon buffer
ZZ	same as :wq
^Z	suspend vi

http://amalloy.hubpages.com/hub/Why-old-text-editors-are-still-great



vi / vim graphical cheat sheet

motion	moves the cursor, or defines the range for an operator
command	direct action command, if red, it enters insert mode
operator	requires a motion afterwards, operates between cursor & destination
extra	special functions, requires extra input
q·	commands with a dot need a char argument afterwards

bol = beginning of line, eol = end of line,
mk = mark, yank = copy

words: quux(foo, bar, baz);
WORDs: quux(foo, bar, baz);

Main command line commands ('ex'):
:w (save), :q (quit), :q! (quit w/o saving)
:e f (open file f),
:%s/x/y/g (replace 'x' by 'y' filewide),
:h (help in vim), :new (new file in vim),

Other important commands:
CTRL-R: redo (vim),
CTRL-F/-B: page up/down,
CTRL-E/-Y: scroll line up/down,
CTRL-V: block-visual mode (vim only)

Visual mode:
Move around and type operator to act on selected region (vim only)

Notes:
(1) use "x before a yank/paste/del command to use that register ('clipboard') (x=a..z,*) (e.g.: "ay$ to copy rest of line to reg 'a')
(2) type in a number before any action to repeat it that number of times (e.g.: 2p, d2w, 5i, d4j)
(3) duplicate operator to act on current line (dd = delete line, >> = indent line)
(4) ZZ to save & quit, ZQ to quit w/o saving
(5) zt: scroll cursor to top, zb: bottom, zz: center
(6) gg: top of file (vim only), gf: open file under cursor (vim only)

For a graphical vi/vim tutorial & more tips, go to **www.viemu.com** - home of ViEmu, vi/vim emulation for Microsoft Visual Studio

http://www.viemu.com/vi-vim-cheat-sheet.gif

vim: Location of vimrc

- Location:
  - Unix:
    - `$HOME/.vimrc`
    - `~/.vimrc`
  - Windows:
    - `:version`
    - `:echo $HOME`
    - `:echo $VIM`

Some Linux Tips

One Danger of `cp`

- Suppose you already have a file named source copy when you type
  ```
  cp source source_copy
  ```
- The cp command will silently replace your old file with a copy of source, and you cannot get it back. It's gone
- It is safer to type
  ```
  cp -i source source_copy
  ```
- which will ask you first if you want to replace source copy:
  ```
  cp: overwrite 'source_copy'?
  ```
- to which you can type '`y`' or '`n`'

```
alias cp='cp -i'
```

Copying Multiple Files into a Directory

- Another form of cp is

```
cp file1 file2 file3 ... fileN destdir
```

- which, if the first N arguments are regular files (not directories, and the last is an existing directory, will make copies of file1, file2, ... fileN and put those copies into destdir

Commands with `less`

- Less if very powerful, and includes many other ways to navigate through a file
- In the following, ***N*** is a number you type before the letter
- If you omit the number, less uses its default value
  - ***N*d** scroll forward N lines(1/2 page by default)
  - ***N*u** scroll backward N lines (1/2 page by default)
  - ***N*g** go to line N in the file (top by default)
  - ***N*%** go to the position that is N% of the file (top)
  - if more than one file has bee `less'd`
    - **`:n`** go to next file,
    - **`:p`** go to previous file

The `file` Command

- `file` command attempts to guess the type of a file
- Is if a file is a shell script, or an executable file, or a device file, or some other kind of special file
  - type:

    `$ file` *filename*
- `file` command will display information about the file whose name is supplied
- For example:

```
$ file .bashrc
.bashrc: ASCII text
```

More on the `file` Command

- The `file` command can also determine the form of an executable file in UNIX

```
$ file /usr/bin/mysql

/usr/bin/mysql: ELF 64-bit LSB executable, x86-64, version
   1 (SYSV), dynamically linked (uses shared libs), for
   GNU/Linux 2.6.18, stripped
```

- This shows that mysql can run on an Intel x86-64 or compatible chip, that it is a 62-bit executable, among other things

For Thursday

- Read: Overview of Programming and Problem Solving
  - http://155.33.203.128/teaching/BIOL6308-Fall2013/local/Literature/mcmillan01.pdf
  - Read up to page 32 - **Algorithmic Problem Solving**
- Go over examples from today, and go through code examples provided in:
  - `/data/METHODS/Fall/LECT5/`
- Go over Perl Readings:
  - 6. Files and Data
  - 7. References
  - 8. Subroutines