

# Lecture 4:

# Programming in Perl: Introduction 3

Torgeir R. Hvidsten

Professor

Norwegian University of Life Sciences

Guest lecturer

Umeå Plant Science Centre

Computational Life Science Cluster (CLiC)

# This lecture

- Go through Lab 2
- Introduction to Perl 3
  - regular expressions
  - parallelization
  - running external programs and commands
- Object-oriented programming
- BioPerl

# Examples of regular expression

```
if ($player =~ m/Kaká/) {  
    print "Traitor!\n";  
}
```

```
if ($player =~ /[K|k]aká/) {  
    print "Still a traitor\n";  
}
```

```
my @genes = split /\s+/, $line;
```

# Regular expression elements

## Symbol classes

`\s, \S` Whitespace character

`\w, \W` Word character

`\d, \D` Digit

`[...]` Character set

`[^...]` Set complement

`.` Wildcard; any character

`\$, \\, \[, \., \*, \+, etc.`

Quoted literals, meta-characters

## Quantifiers

`*` Zero or more

`+` One or more

`?` Zero or one

`{a,}` a or more

`{,b}` b or less

`{a,b}` a to b inclusive

## Logic

`|` Logical “or”

`(...)` Grouping

for

quantifiers

## Anchors

`^` Beginning of string

`$` End of string

`\b` Word-boundary

`\B` Non-word boundary

# Perl regular expression

- `/a+/` # Match one or more a's, ex: a,aa,aaa...
- `/[aeiou]/` # Match a vowel
- `/[^aeiou]/` # Match a non-vowel
- `/\s+/` # Match one or more whitespaces
- `/\S+/` # Match one or more non-whitespaces
- `/\d+/` # Match unsigned integer = `/[0-9]+/`
- `/\d+\.\d+/` # Match unsigned floats, ex: 3.1415
- `/\w*/` # Zero or more word characters = `/[a-zA-Z]*/`
- `/\W?/` # Zero or one non-word character = `/[^a-zA-Z]?/`
- `/this | that /` # Match *this* or *that* = `/th(is | at) /`
- `/c.t/` # Match cat, cut, ctt, c@t, c t, not tic tac, ...
- `/b.{2,4}t/` # Match boot, beat, blast, b- t, bastat, btttt
- `/b[^t]{2,4}t/` # Match *blast* and *beast* but not btttt

# Perl regular expression

- `/\bhunt/` # Match *hunt*, *hunter*, but not *shunt* or *\_hunt*
- `/\bsearch\B/` # Match *searching*, *searches* but not *search*
- `/\[[^]]*\]/` # Match anything surrounded by []
- `/^[A-Z][a-z]*/` # Capitalized word at beginning of string
- `/\.$/` # Match a period at the end of the string
- `/\n/` # Match a new-line character
  
- `/http:\/\/\//` # You need to quote literal slashes with backslashes

# Basic comparison

- Returns true if string `$string` contains substring "sought\_text", false otherwise:  
`$string =~ m/sought_text/;`
- Returns true if string `$string` contains substring "sought\_text" at the very beginning:  
`$string =~ m/^sought_text/;`
- Returns true if the sought text is the very last text in the string:  
`$string =~ m/sought_text$/;`
- Returns true only if `$string` contains the sought text and nothing but the sought text:  
`$string =~ m/^sought_text$/;`
- Case insensitive comparison:  
`$string =~ m/^sought_text$/i;`
- Note: `m` is optional
- `=~` return true if `$string` matches the pattern, file `!~` returns true if `$string` does not match the pattern

# Back-references in Perl

- Besides for grouping (e.g. / th(is|at) / ), parentheses save the part they match for use later in the Perl code (**\$1** form).
- The value matched by the first set of parentheses is accessed with **\$1**, the value matching the second set in **\$2** and so on.

```
my $string = "Protein structure: 1awa (P < 0.0001)";  
$string =~ m/^.*: (\w{4}) \(P < (\d\.\d+)\)$/;  
print "$1 $2\n";
```

1awa 0.0001



# Non-greedy versions of quantifiers

- By default, regular expression match: 1) the left-most valid substring, and 2) extend as far right as possible.
- In Perl, you can change the second behavior with *non-greedy* versions of quantifiers, e.g. `+?`, `*?`, `{2,4}?`

```
my $text = "milk and cookies";  
$text =~ /(\w+)/;  
print "$1\n";  
milk
```

```
$text =~ /(m.*i)/;  
print "$1\n";  
milk and cooki
```

```
$text =~ /(m.*?i)/;  
print "$1\n";  
mi
```

```
my $buttons = "<top> <bottom>";  
$buttons =~ /(<.*>)/;  
print "$1\n";  
<top> <bottom>
```

```
$buttons =~ /(<.*?>)/;  
print "$1\n";  
<top>
```

# Regex Substitutions: s/// and s///g

- The *substitution operator* `s///` is an incredibly powerful tool for text transformation.
- The pattern between the first two delimiters is replaced by the string between the last two.
- Use the `g` modified-form `s///g` to replace all matches in a line.

```
my $text = "milk and cookies";
```

```
$text =~ s/\s/_/;
```

```
print "$text\n";
```

```
milk_and cookies
```

```
$text =~ s/\s/_/g;
```

```
print "$text\n";
```

```
milk_and_cookies
```

# Object-oriented (OO) programming

- The key idea of OO programming is that all data is stored and modified with special data structures called **objects**,
- and each kind of object can be accessed only by its defined subroutines called **methods**.
- The user of an OO class is typically spared the effort of directly manipulating data, and can use class methods for this instead.

# Understanding *objects*

- Object = **Collection** of data that logically belongs together.
  - E.g., a “genome” object has parts (“attributes”) such as...
    - Name of the species
    - Its DNA sequence
    - A list of genes, each associated with one or more transcripts
    - A list of start and end points for each exon
    - etc
- A **type** of object (e.g., genome object) is called a *class*
  - All objects derive from a class

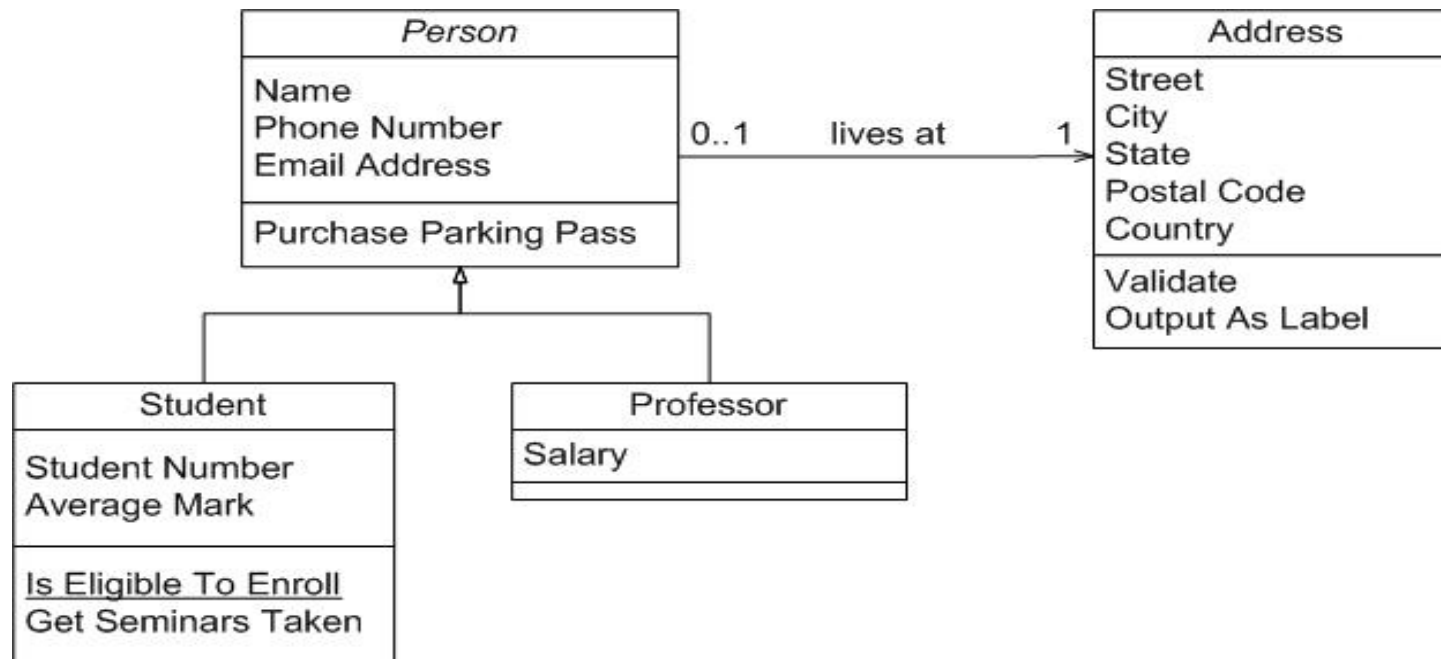
# Understanding *methods*

- A *Method* is just like a subroutine but associated **specifically** with a class; they are not shared, except by “inheritance”
- Each type of object has one or more methods that it can call, and **only** those methods
  - The only way to access the data in an object is via the methods defined for that class.
- E.g., a genome object might have ...
  - A *compare* method, for whole-genome comparisons
  - A *list-gene-families* method, for listing all gene families known to exist in a genome
  - A *GC-percent function*, for calculating %GC in specific areas of the genome, or all of it.

# Understanding *classes*

- A **Class** is an **object definition** + a collection of **methods**.
- A specific object (e.g. a genome object for *H. sapiens*) is called an *instance* of a class.

# Example of class definition and inheritance



# Class

```
package Person;
# The object constructor
sub new {
    my $self = {};
    $self->{NAME} = undef;
    $self->{AGE} = undef;
    $self->{PEERS} = [];
    bless($self);
    return $self;
}
# Methods to access object data
sub name {
    my $self = shift;
    if (@_) { $self->{NAME} = shift }
    return $self->{NAME};
}
sub age {
    my $self = shift;
    if (@_) { $self->{AGE} = shift }
    return $self->{AGE};
}
sub peers {
    my $self = shift;
    if (@_) { @{$self->{PEERS}} = @_ }
    return @{$self->{PEERS}};
}
1;
```

# OO in Perl

## Program

```
use Person;

my $him = Person->new();
$him->name("Jason");
$him->age(23);
$him->peers( "Norbert", "Rhys", "Phineas" );

my @all_recs;
push @all_recs, $him; # save object in array for later

my $name = $him->name();
my $age = $him->age();
print "$name is $age years old.\n";

my @peers = @{$him->peers};
print "His peers are: @peers\n";

Jason is 23 years old.
His peers are: Norbert Rhys Phineas
```



# BioPerl

BioPerl: >1,000 modules divided into several packages

- Free
- “Open Source” software

## **Bioperl Group**

## **Functions**

bioperl (the core)

Most of the main functionality of Bioperl.

bioperl-run

Wrappers to a lot of external programs.

bioperl-ext

Interaction with some alignment functions and the Staden package.

bioperl-db

Using bioperl with BioSQL and local relational databases.

bioperl-microarray

Microarray specific functions.

bioperl-gui

Some preliminary work on a graphical user interface to some Bioperl functions.

# BioPerl

BioPerl provides object classes for various types of bioinformatics analysis

- external programs (e.g. BLAST, FASTA, clustalw and EMBOSS).
- various types of databases for storage and retrieval of data
- sequence analysis
- gene expression analysis
- etc

# Bio::Perl

## module designed for beginners

- Bio::Perl is a module designed for beginners with easy access to a small number of Bioperl's functionality
- Bio::Perl is not object-oriented

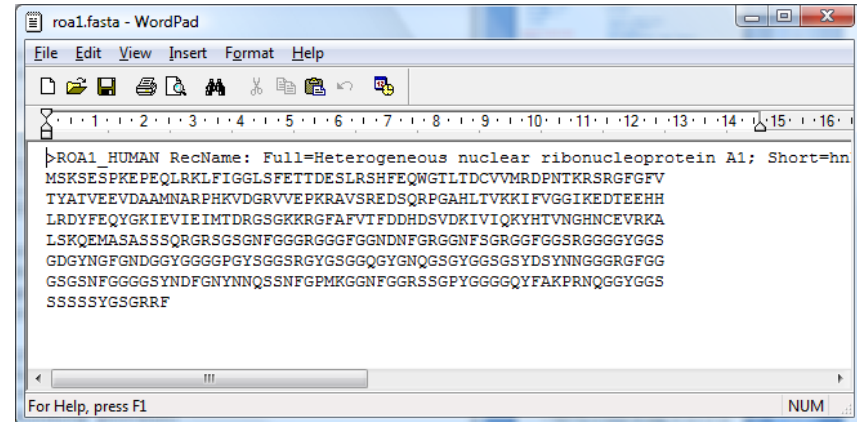
get_sequence	- gets a sequence from standard, internet accessible databases
read_sequence	- reads a sequence from a file
read_all_sequences	- reads all sequences from a file
new_sequence	- makes a Bioperl sequence just from a string
write_sequence	- writes a single or an array of sequence to a file
translate	- provides a translation of a sequence
translate_as_string	- provides a translation of a sequence, returning back just the sequence as a string
blast_sequence	- BLASTs a sequence against standard databases at NCBI
write_blast	- writes a blast report out to a file

# Bio::Perl example

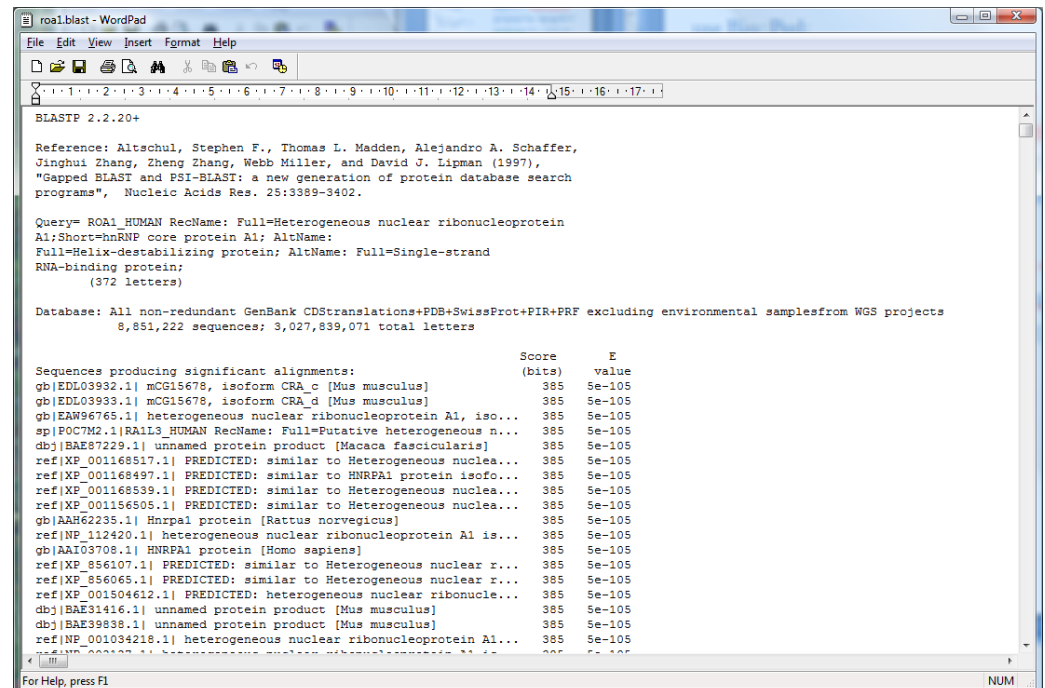
use Bio::Perl;

```
# the databases you can get sequences from  
# are 'swiss', 'genbank', 'genpept', 'embl', and 'refseq'  
my $seq = get_sequence('swiss',"ROA1_HUMAN");  
write_sequence(">roa1.fasta",'fasta',$seq);
```

```
# uses the default database - nr in this case  
my $blast_result = blast_sequence($seq);  
write_blast(">roa1.blast",$blast_result);
```



```
roa1.fasta - WordPad  
File Edit View Insert Format Help  
MSKSESPKEPEQLRKLFIGGLSFETTDLSLRSHFEQWGLTDCVVRDPNTRKSRGFGFV  
TYATVEEVDAAAMNARPHKVDGRVVEPKRAVSREDSQRPGAHLTVKKIFVGGIKEDTEHH  
LRDYFEQYQKIEVIEIMTDRGSGKKRGAFAVTFDDHDSVDKIVIQKYHTVNGHNCEVRKA  
LSKQEMASASSQRGRSGSNFSGGGRGGGFGGDNDFGRGGNFSGRGGFGGSRGGGGYGGGSDYNGFGNDGGYGGGPGYGGGSRGYGSGGQYGNQSGSYGSGSYDSYNNNGGGRGFGG  
GSGSNFSGGGSYNDGFGYNNQSSNFGPMKGGNFGRSSGPGYGGGGYFAKPRNQGGYGGG  
SSSSSYSGSRRF  
For Help, press F1
```



```
roa1.blast - WordPad  
File Edit View Insert Format Help  
BLASTP 2.2.20+  
Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,  
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),  
"Gapped BLAST and PSI-BLAST: a new generation of protein database search  
programs", Nucleic Acids Res. 25:3389-3402.  
Query= ROA1_HUMAN RecName: Full=Heterogeneous nuclear ribonucleoprotein  
A1;Short=hnRNP core protein A1; AltName:  
Full=Helix-stabilizing protein; AltName: Full=Single-strand  
RNA-binding protein;  
(372 letters)  
Database: All non-redundant GenBank CDStranslations+PDB+SwissProt+PIR+PRF  
excluding environmental samples from WGS projects  
8,851,222 sequences; 3,027,839,071 total letters  
Sequences producing significant alignments:  
Score E  
(bits) value  
gb|EDL03932.1| mCG15678, isoform CRA_c [Mus musculus] 385 5e-105  
gb|EDL03933.1| mCG15678, isoform CRA_d [Mus musculus] 385 5e-105  
gb|EAW96765.1| heterogeneous nuclear ribonucleoprotein A1, iso... 385 5e-105  
sp|POC7M2.1|RAL13_HUMAN RecName: Full=Putative heterogeneous n... 385 5e-105  
dbj|BAE87229.1| unnamed protein product [Macaca fascicularis] 385 5e-105  
ref|XP_001168517.1| PREDICTED: similar to Heterogeneous nuclea... 385 5e-105  
ref|XP_001168497.1| PREDICTED: similar to HNRPA1 protein isofo... 385 5e-105  
ref|XP_001168539.1| PREDICTED: similar to Heterogeneous nuclea... 385 5e-105  
ref|XP_001156505.1| PREDICTED: similar to Heterogeneous nuclea... 385 5e-105  
gb|AAH62235.1| Hnrpa1 protein [Rattus norvegicus] 385 5e-105  
ref|NP_112420.1| heterogeneous nuclear ribonucleoprotein A1 is... 385 5e-105  
gb|AAI03708.1| HNRPA1 protein [Homo sapiens] 385 5e-105  
ref|XP_856107.1| PREDICTED: similar to Heterogeneous nuclear r... 385 5e-105  
ref|XP_856065.1| PREDICTED: similar to Heterogeneous nuclear r... 385 5e-105  
ref|XP_001504612.1| PREDICTED: heterogeneous nuclear ribonucle... 385 5e-105  
dbj|BAE31416.1| unnamed protein product [Mus musculus] 385 5e-105  
dbj|BAE39838.1| unnamed protein product [Mus musculus] 385 5e-105  
ref|NP_001034218.1| heterogeneous nuclear ribonucleoprotein A1... 385 5e-105  
ref|NP_001034218.1| heterogeneous nuclear ribonucleoprotein A1... 385 5e-105  
For Help, press F1
```

# BioPerl: the *Sequence* object

```
use Bio::Seq;
```

```
my $seq_obj = Bio::Seq->new(-seq =>  
    "aaaatggggggggggggcccccgtt",  
    -display_id => "#12345",  
    -desc => "example 1",  
    -alphabet => "dna" );
```

```
print $seq_obj->display_id(), ": ", $seq_obj->seq();
```

```
#12345: aaaatggggggggggggcccccgtt
```

- `use Bio::Seq;` tells Perl to use a module on your machine called "Bio/Seq.pm".
- The variable `$seq_obj` is a *Sequence* object
- Arguments are passed to the method `new()` using "hash syntax"
- `display_id()` and `seq()` are methods that returns the id and sequence as strings.

# BioPerl: The *SeqIO* object

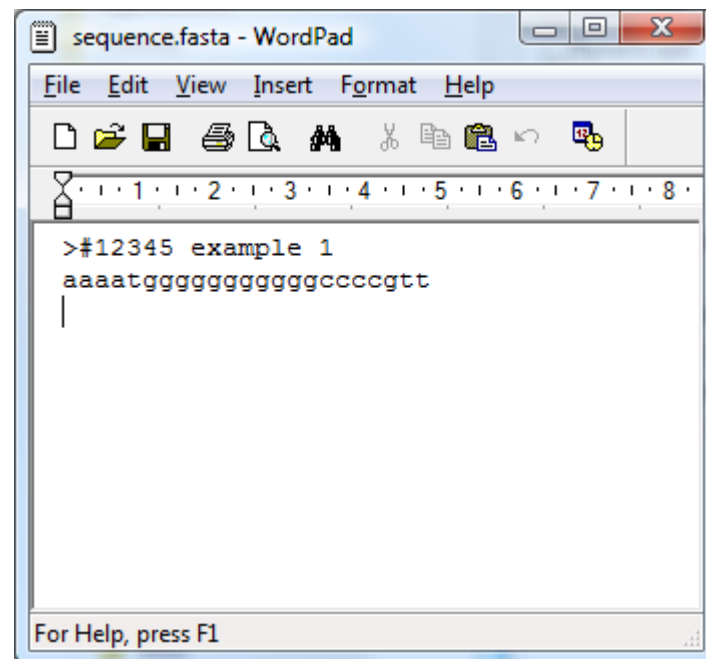
```
use Bio::Seq;  
use Bio::SeqIO;
```

```
my $seq_obj = Bio::Seq->new(-seq =>  
    "aaaatggggggggggggcccccgtt",  
    -display_id => "#12345",  
    -desc => "example 1",  
    -alphabet => "dna" );
```

```
my $seqio_obj = Bio::SeqIO->new(-file =>  
    '>sequence.fasta', -format => 'fasta' );
```

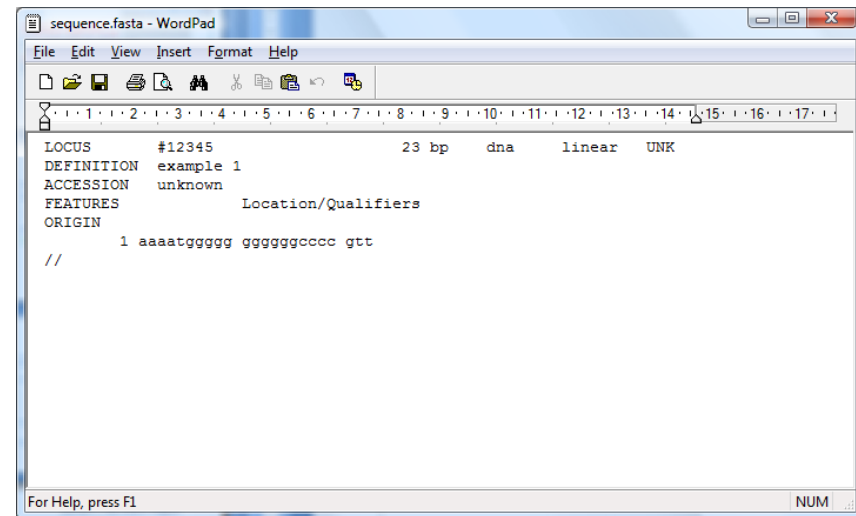
```
$seqio_obj->write_seq($seq_obj);
```

- The variable `$seqio_obj` is a *SeqIO* object
- The “>” in the `-file` argument indicates that we're going to write to the file



# BioPerl: The *SeqIO* object

- Changing format from **fasta** to **genbank** in the previous program changes the output
- This illustrates some of the flexibility and power of using an IO object over **open**



```
sequence.fasta - WordPad
File Edit View Insert Format Help
LOCUS #12345 23 bp dna linear UNK
DEFINITION example 1
ACCESSION unknown
FEATURES Location/Qualifiers
ORIGIN
1 aaaatggggg ggggggcccc gtt
//
For Help, press F1 NUM
```

# BioPerl: The *SeqIO* object

```
use Bio::SeqIO;
```

```
$seqio_obj = Bio::SeqIO->new(-file =>  
    "sequence.fasta", -format => "fasta" );
```

```
while (my $seq_obj = $seqio_obj->next_seq()) {  
    # print the sequence  
    print $seq_obj->seq(),"\n";  
}
```

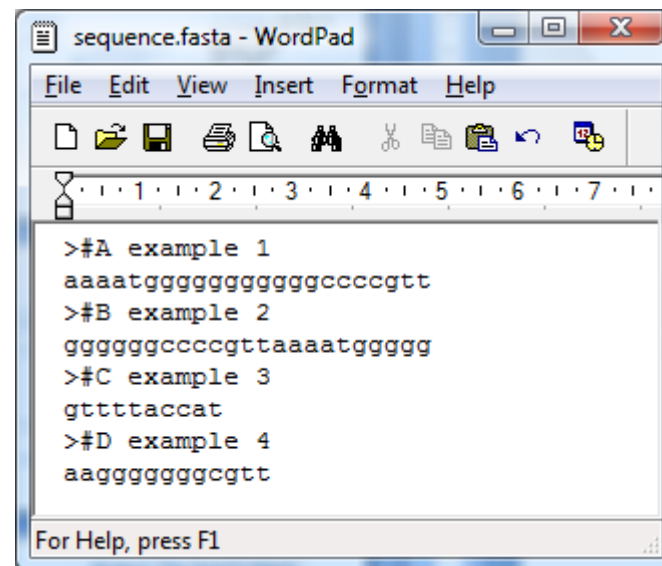
```
aaaatggggggggggggcccccgtt
```

```
ggggggcccccgttaaaatggggg
```

```
gttttaccat
```

```
aaggggggggcgtt
```

- No “>” in the -file argument indicates that we're going to read from the file
- The method `next_seq()` is typical for BioPerl





# Retrieving a sequence from a database

```
use Bio::DB::SwissProt;  
  
my $db_obj = Bio::DB::SwissProt->new;  
  
my $seq_obj = $db_obj->get_Seq_by_acc("ROA1_HUMAN");  
  
print $seq_obj->seq(),"\n";
```

```
MSKSESPKEPEQLRKLFIGGLSFET'TDESLRSHFEQWGT'LT'  
DCVVMRDPN'TKRSRGFGFV'TYATVEEVDAAMNARPHKV  
DGRVVEPKRAVSREDSQRPGAHLTVKKIFVGGIKEDTEE  
HHLRDYFEQYGKIEVIEIMTDRGSGKKRGFAFVTFDDHD  
SVDKIVIQKYHTVNGHNCVVRKALSKQEMASASSSQRGR  
SGSGNFGGGRGGGFGGNDNFGRGGNFSGRGGFGGSRG  
GGGYGGSGDGYNGFGNDGGYGGGGPGYSGGSRGYGS  
GGQGYGNQGSYGGSGSYDSYNNGGGRGFGGGSGSNF  
GGGGSYNDFGNYNNQSSNFGPMKGGNFGRSSGPYGG  
GGQYFAKPRNQGGYGGSSSSSSYGSRRF
```

- Bio::DB retrieves sequences from online databases
- Other alternatives:
  - GenBank (Bio::DB::GenBank)
  - GenPept (Bio::DB::GenPept)
  - EMBL (Bio::DB::EMBL)
  - SeqHound (Bio::DB::SeqHound)
  - Entrez Gene (Bio::DB::EntrezGene)
  - RefSeq (Bio::DB::RefSeq)

# Running external programs/system commands

- Both Perl's `exec()` function and `system()` function execute a system shell command

- `system()` runs the command and returns when done.

```
system("mkdir TEST");
```

```
print "Finished!\n";
```

Finished!

- `exec()` runs the command and do not return.

```
exec("mkdir TEST");
```

```
print "Finished!\n";
```

# Running external programs/system commands

- To capture the output of a system command, use the backtick operator:

```
my $result = `dir bioperl.pl`;
```

```
print "$result\n";
```

Volume in drive C is OS

Volume Serial Number is CA2C-F64B

Directory of C:\Labs

2009-05-12 14:20

732 bioperl.pl

1 File(s)

732 bytes

0 Dir(s)

848 bytes free

# Running BLAST from Perl

Standard approach:

```
system("blastall -d C:\Blast\db\yeast.nt -i sequence.txt >hits.txt");
```

```
open (H, "hits.txt");
```

```
# Parse output
```

```
...
```

```
close(H);
```

# External programs in BioPerl

Bio::Tools::Run contains a large number of modules for running bioinformatics tools

```
use Bio::Seq;
use Bio::Tools::Run::StandAloneBlast;

my $blast_obj = Bio::Tools::Run::StandAloneBlast->new(program => 'blastn', database => 'yeast.nt');

my $seq_obj = Bio::Seq->new(-id =>"test_query", -seq =>"TTTAAATATATTTTGAAGTATAGATTATATGTT");

my $report_obj = $blast_obj->blastall($seq_obj);

while(my $result = $report_obj->next_result) {
    while(my $hit = $result->next_hit) {
        while(my $hsp = $hit->next_hsp) {
            if ( $hsp->percent_identity > 75 ) {
                print "Hit: ", $hit->name, ", Length: ", $hsp->length('total'), ", Percent_id: ",
                    $hsp->percent_identity, "\n";
            }
        }
    }
}
Hit: gi|6323989|ref|NC_001146.1|, Length: 15, Percent_id: 100
Hit: gi|6322623|ref|NC_001143.1|, Length: 17, Percent_id: 94.1176470588235
...
```

# Parallelization

- Many bioinformatics problems can be divided into smaller ones, which are then solved concurrently ("in parallel").
  - E.g. finding paralogs in a huge genome. How?
- Parallelization has become more important in recent years do to the commonness of multicore processors

# Parallelization in Perl

- A process is an instance of a computer program that is being sequentially executed
- `fork()`: Create a duplicate process (child) of the current process (parent)
- Each process is given a process ID by the operating system
- `fork()` returns **the child process ID** to the parent on success, **0** to the child on success and **undef** on failure to fork

# Parallelization in Perl

```
my @genome = get_all_gene_sequences();

my $num = 10;
my @children;

for (my $i = 0; $i < $num; $i++) {
    my $pid = fork();
    if ($pid) { # parent
        push @children, $pid;
    } elsif ($pid == 0) { # child
        print "child $i\n";
        find_paralogs($i, $num, \@genome);
        exit;
    } else {
        print "couldn't fork\n";
    }
}

foreach my $child (@children) {
    waitpid($child, 0);
}

collect_paralogs($num);
```

```
sub get_all_gene_sequences {
    # Read all gene sequences in the genome from file
    return @genome;
}

sub find_paralogs {
    my $i = $_[0];
    my $num = $_[1];
    my @genome = @{$_[3]};

    my $n = @genome;
    my @slice = @genome[$i*$n/$num..($i+1)*$n/$num-$n/$num-1];

    # Blast @slice against @genome and write paralogs to file
}

sub collect_paralogs {
    my $num = $_[0];
    # Read paralogs from files and write them to a common file
}
```



# Acknowledgements

- Several slides were taken or re-worked from David Ardell and Yannick Pouliot.