Perl for Biologists

Session 9

April 29, 2015

Subroutines and functions

Jaroslaw Pillardy

Suggestions Welcomed!

There are three more sessions devoted to practical examples

They are intended to show all the techniques and semantics of Perl in work on actual bioinformatics tasks

Do you have (or know about) any problems that you think qualify as an example?

I will be happy to hear about it! Please send an e-mail to jp88@cornell.edu – maybe your problem will be a part of one of the three practical sessions.

Session 8 Exercises Review

Modify the script `extract_from_fasta.pl` to select sequences which

- Are on the list of requested sequences **OR**
- Contain a given DNA motif

/home/jarekp/perl_08/exercise1_extract_from_fasta.pl

Modify the script `filter_bam.pl` to

- filter out all alignments with indels (use XO and XG tags)
- Accept SAM input from STDIN and write output to STDOUT, so that the filtering command would be

```
samtools view -h maize_tst.bam | ./filter_bam.pl |
samtools view -Sb - > maize_filtered.bam
```

/home/jarekp/perl_08/exercise2_filter_bam.pl

Subroutines and functions

Subroutine or function is:

- named block of code
- can be called with parameters from anywhere in the program
- returns a value

script1.pl

```
#!/usr/local/bin/perl
#script to list and count files
#with a given extension in a given directory

$ext = "pl";
$dir = "/home/jarekp";

$n = 0;
$m = 0;
opendir DIR, $dir;
foreach $entry (readdir DIR)
{
        if($entry ne "." && $entry ne ".."){$n++};
        if($entry =~ /\.$ext$/)
        {
                print "$dir/$entry\n";
                $m++;
        }
}
closedir(DIR);
$fperc = 100*$m/$n;

print "There were $fperc\% of .$ext files in directory $dir\n";
```

```perl
#!/usr/local/bin/perl
#script to list and count files
#with a given extension in a given directory

$ext = "pl";
$dir = "/home/jarekp";
{
        $n = 0;
        $m = 0;
        opendir DIR, $dir;
        foreach $entry (readdir DIR)
        {
                if($entry ne "." && $entry ne ".."){$n++};
                if($entry =~ /\.$ext$/)
                {
                        print "$dir/$entry\n";
                        $m++;
                }
        }
        closedir(DIR);
        $fperc = 100*$m/$n;
}
print "There were $fperc%% of .$ext files in directory $dir\n";
```

script1a.pl

```
#!/usr/local/bin/perl
#script to list and count files
#with a given extension in a given directory
$ext = "pl";
$dir = "/home/jarekp";
&listfiles;
print "There were $fperc%% of .$ext files in directory $dir\n";

sub listfiles
{
        $n = 0;
        $m = 0;
        opendir DIR, $dir;
        foreach $entry (readdir DIR)
        {
                if($entry ne "." && $entry ne ".."){$n++};
                if($entry =~ /\.$ext$/)
                {
                        print "$dir/$entry\n";
                        $m++;
                }
        }
        closedir(DIR);
        $fperc = 100*$m/$n;
}
```

Subroutines and functions

Subroutine can be declared in Perl script as a named block of code:

```
sub sub_name
{
        code;
}
```

There is no difference between subroutine and function: declaration is the same and it **ALWAYS returns a value** (not always a useful one …)

Subroutines and functions

Subroutine can be called or referenced in two ways

By name as an object

```
&sub_name;
```

By name as a subroutine

```
sub_name();
```

script1b.pl

```
#!/usr/local/bin/perl
#script to list and count files
#with a given extension in a given directory
$ext = "pl";
$dir = "/home/jarekp";
$ret = listfiles();
print "There were $fperc%% of .$ext files in directory $dir\n";
print "Return value is $ret\n";
sub listfiles
{
        $n = 0;
        $m = 0;
        opendir DIR, $dir;
        foreach $entry (readdir DIR)
        {
                if($entry ne "." && $entry ne ".."){$n++};
                if($entry =~ /\.$ext$/)
                {
                        print "$dir/$entry\n";
                        $m++;
                }
        }
        closedir(DIR);
        $fperc = 100*$m/$n;
}
```

Subroutines and functions

Subroutine ALWAYS return a value!

Implicit

Result of the last operation in the sub block.

Explicit

```
return $var;
```

NOTE: `return` statement TERMINATES the execution of the subroutine and returns to the main code immediately!

script1c.pl

```
#!/usr/local/bin/perl
#script to list and count files
#with a given extension in a given directory
$ext = "pl";
$dir = "/home/jarekp";
$ret = listfiles();
print "There were $fperc%% of .$ext files in directory $dir\n";
print "Return value is $ret\n";
sub listfiles
{
        $n = 0;
        $m = 0;
        opendir DIR, $dir;
        foreach $entry (readdir DIR)
        {
                if($entry ne "." && $entry ne ".."){$n++};
                if($entry =~ /\.$ext$/)
                {
                        print "$dir/$entry\n";
                        $m++;
                }
        }
        $fperc = 100*$m/$n;
        closedir(DIR);
}
```

returns value of this function

script1d.pl

```
#!/usr/local/bin/perl
#script to list and count files
#with a given extension in a given directory
$ext = "pl";
$dir = "/home/jarekp";
$ret = listfiles();
print "There were $fperc%% of .$ext files in directory $dir\n";
print "Return value is $ret\n";
sub listfiles
{
        $n = 0;
        $m = 0;
        opendir DIR, $dir;
        foreach $entry (readdir DIR)
        {
                if($entry ne "." && $entry ne ".."){$n++};
                if($entry =~ /\.$ext$/)
                {
                        print "$dir/$entry\n";
                        $m++;
                }
        }
        $fperc = 100*$m/$n;
        closedir(DIR);
        return $fperc;
}
```

Why do we use subroutines?

- To reuse the code in various places of the program without copying it over
  copied code takes a lot of space and all the copies must be changed simultaneously if modified
- To make program more readable and clear
- To reuse the code in various other programs (function libraries and modules)

Global and local variables: scope

By default all variables declared in Perl script are accessible everywhere in the code, including subroutines.

By default Perl variables are **global**.

It is potentially very dangerous and typically leads to errors!

It also makes coding in Perl faster, especially for short projects.

Extensive use of global variables in long programs makes the code much harder to analyze and understand …

script2.pl (1)

```
#!/usr/local/bin/perl
#script to list and count files
#with a given extension in a set of directories
$ext = "pl";
$base = "/home/jarekp";
@dirs = qw(perl_01 perl_02 perl_03 perl_04 perl_05);

for($n=0; $n<$#dirs; $n++)
{
    $dir = $base . "/" . $dirs[$n];
    print "$n Listing $dir\n";
    $ret = listfiles();
    print "There were $ret\% of .$ext files in directory dir\n";
    print "\$n is now $n\n";
}
```

```
sub listfiles
{
        $n = 0;
        $m = 0;
        opendir DIR, $dir;
        foreach $entry (readdir DIR)
        {
                if($entry ne "." && $entry ne ".."){$n++};
                if($entry =~ /\.$ext$/)
                {
                        print "$dir/$entry\n";
                        $m++;
                }
        }
        $fperc = 100*$m/$n;
        closedir(DIR);
        return $fperc;
}
```

Global and local variables: scope

script2.pl lists only the first of the directories in `@dirs`

global variable `$n` is modified both in the main program body and in the subroutine `listfiles`

it is a mess!

Global and local variables: scope

Local variables, accessible only in a given code block can be declared using "**my**" keyword:

```
my $variable;
```

Local variable can be declared in ANY code block, not only in a subroutine.

Local variable declared in a code block is also declared in all child code blocks inside this code block

Using local variables wherever possible is a VERY good programming practice.

```
sub listfiles
{
        my $n = 0;
        my $m = 0;
        opendir DIR, $dir;
        foreach my $entry (readdir DIR)
        {
                if($entry ne "." && $entry ne ".."){$n++};
                if($entry =~ /\.$ext$/)
                {
                        print "$dir/$entry\n";
                        $m++;
                }
        }
        my $fperc = 100*$m/$n;
        closedir(DIR);
        return $fperc;
}
```

Global and local variables: scope

If there is a local variable and global variable with the same names, in the code block where it is declared only local one is accessible, while global one is inaccessible and unchanged.

Arguments

Communicating with a subroutine using global variables (like in all previous scripts) is a bad idea!

With global variables:

- hard to analyze the code
- names of the variables must be always the same
- subroutine is not a separate code – global variables bind it to the main code block

The proper way to communicate with a subroutine is to use arguments.

Arguments

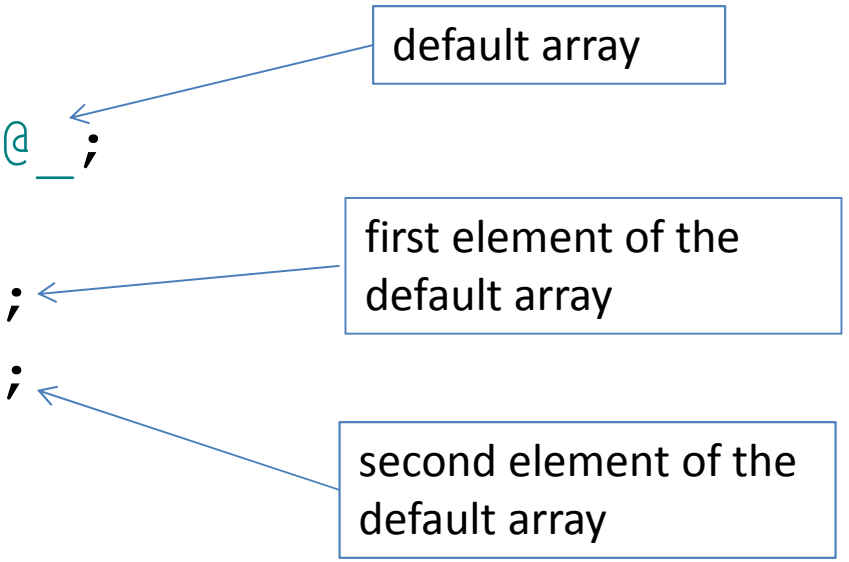
Subroutine's arguments are a list following sub name:

```
$ret = listfiles($dir, $ext);
```

They can be accessed inside the subroutine as parts of the DEFAULT ARRAY

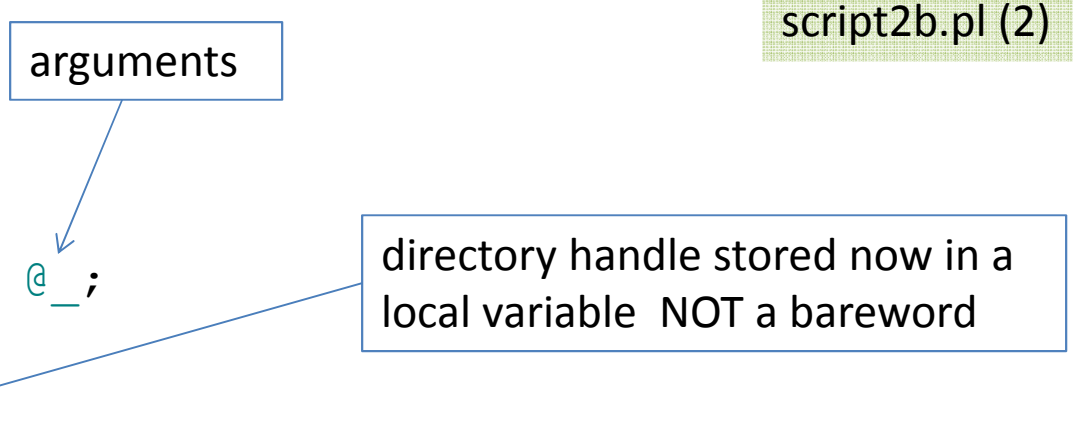
```
my @arguments = @_;

my $arg1 = $_[0];
my $arg2 = $_[1];
```



```
#!/usr/local/bin/perl
#script to list and count files
#with a given extension in a set of directories
my $ext = "pl";
my $base = "/home/jarekp";
my @dirs = qw(perl_01 perl_02 perl_03 perl_04 perl_05);

for(my $n=0; $n<$#dirs; $n++)
{
  my $dir = $base . "/" . $dirs[$n];
  print "$n Listing $dir\n";
  my $ret = listfiles($dir, $ext);
  print "There were $ret\% of .$ext files in directory $dir\n";
}
```




```
sub listfiles
{
        my ($dir, $ext) = @_;
        my $n = 0;
        my $m = 0;
        opendir my $DIR, $dir;
        foreach my $entry (readdir $DIR)
        {
                if($entry ne "." && $entry ne ".."){$n++};
                if($entry =~ /\.pl$/)
                {
                        print "$dir/$entry\n";
                        $m++;
                }
        }
        my $fperc = 100*$m/$n;
        closedir($DIR);
        return $fperc;
}
```

Arguments

Arrays can be passed as arguments

```
$ret = sub_name($var1, $var2, @arr);
```

All the arguments are interpreted in list context

They can be accessed inside the subroutine as parts of `@_`
=> in this example first two elements of `@_` array are `$var1` and `$var2` the rest is just array `@arr` :

```
my $subvar1 = shift @_;
my $subvar2 = shift @_;
my @subarr = @_;
```

Arguments

Hashes can be passed as arguments

```
$ret = sub_name($var1, $var2, %hash);
```

All the arguments are interpreted in list context
Hash is converted into an array (keys odd, values even)
They can be accessed inside the subroutine as parts of `@_`
=> in this example first two elements of `@_` array are `$var1` and `$var2` the rest is hash `%hash` :

```
my $subvar1 = shift @_;
my $subvar2 = shift @_;
my %subhash = @_;
```

array converted back into hash

script3.pl (1)

```
#!/usr/local/bin/perl
#script to sort first n elements
#of an array

#initialize the array to
#random integers between 1 and 100
my @arr;
for(my $i=0; $i<15; $i++)
{
        $arr[$i] = 1 + int(rand(100));
}

print "Unsorted array:\n";
print_array(@arr);
my @sorted = sort_array(10, "desc", @arr);
print "Sorted array:\n";
print_array(@sorted);

sub print_array
{
        my @arr = @_;
        for(my $i=0; $i<=$#arr; $i++)
        {
                printf("%3d. %3d\n", $i+1, $arr[$i]);
        }
}
```

script3.pl (2)

```
sub sort_array
{
#sorts an array in order specified in arg #2 ('desc', 'asc')
#sorts only arg #1 first elemnts
#array follows arg #1 and #2
        my $limit = shift @_;
        my $order = shift @_;
        my @arr = @_;

        for(my $i=0; $i<$limit-1; $i++)
        {
                my $extr = $i;
                for(my $j=$i+1; $j<$limit; $j++)
                {
                        if($order eq "asc")
                        {
                                if($arr[$j] < $arr[$extr]){$extr=$j;}
                        }
                        else
                        {
                                if($arr[$j] > $arr[$extr]){$extr=$j;}
                        }
                }
                my $tmp = $arr[$i];
                $arr[$i] = $arr[$extr];
                $arr[$extr] = $tmp;
        }
        return @arr;
}
```

A subroutine can of course call other subroutines

A subroutine can also call itself and become **recursive**

Recursive subroutines are a very powerful tool, but they are also very risky

They usually use a lot of more memory and can bring your computer to a standstill.

They must be carefully controlled!

Example: write a program to list all files with a given extension in a directory tree.

```
#!/usr/local/bin/perl
#script to list and count files with a given extension in a directory tree
if($#ARGV != 1)
{
        print "USAGE: script3.pl directory extension\n";
        exit;
}
my $base = $ARGV[0];
my $ext = $ARGV[1];

my ($n, $m, $nf, $nd) = listfiles($base, $ext);
print "----------------------\n";
print "There were $m of .$ext files out of total $n objects\n";
print "including $nf files total and $nd directories total\n";
print "in a directory tree starting from $base\n";
```

script4.pl (2)

```
sub listfiles
{
        my ($dir, $ext) = @_;
        my $ntot = 0, $n_ext = 0, $nfiles = 0, $ndirs = 0;
        opendir my $DIR, $dir;
        foreach my $entry (readdir $DIR)
        {
              if($entry eq "." || $entry eq ".."){next;}
              $ntot++;
              if(-f "$dir/$entry")
              {
                      $nfiles++;
                      if($entry =~ /\.$ext$/)
                      {
                              print "$dir/$entry\n";
                              $n_ext++;
                      }
              }
              if(-d "$dir/$entry")
              {
                  $ndirs++;
                  my ($ntot1,$n_ext1,$nfiles1,$ndirs1)=listfiles("$dir/$entry",$ext);
                  $ntot += $ntot1;
                  $n_ext += $n_ext1;
                  $nfiles += $nfiles1;
                  $ndirs += $ndirs1;
              }
        }
        closedir($DIR);
        return ($ntot, $n_ext, $nfiles, $ndirs);
}
```

There is a serious problem with the script4.pl …

It can potentially lead to an infinite recursion when encountering circular file reference!

=> demo with /home/jarekp/testdir directory tree

Arguments

In programming, there are two methods of passing variables to subroutines:

BY VALUE

The subroutine receives a **copy of the data**, and any changes made in a subroutine **DO NOT** affect original variables

BY REFERENCE

The subroutine receives **exactly the same variables** as listed in the arguments, any changes made in a subroutine **DO affect** original variables.

Arguments

Perl default method of passing arguments to the subroutine is by **reference**.

However, most programmers use them in a way simulating passing by **value**.

script5.pl (1)

```
#!/usr/local/bin/perl
#script to sort first n elemnts
#of an array

#initialize the array to
#random integers between 1 and 100
my @arr;
for(my $i=0; $i<15; $i++)
{
        $arr[$i] = 1 + int(rand(100));
}
print "Unsorted array:\n";
print_array(@arr);
my @sorted = sort_array(10, "desc", @arr);
print "Sorted array:\n";
print_array(@sorted);
print "Original array after sorting:\n";
print_array(@arr);

sub print_array
{
        my @arr = @_;
        for(my $i=0; $i<=$#arr; $i++)
        {
                printf("%3d. %3d\n", $i+1, $arr[$i]);
        }
}
```

```
sub sort_array
{
#sorts an array in order specified in arg #2 ('desc', 'asc')
#sorts only arg #1 first elemnts
#array follows arg #1 and #2
        my $limit = shift @_;
        my $order = shift @_;
        my @arr = @_;

        for(my $i=0; $i<$limit-1; $i++)
        {
                my $extr = $i;
                for(my $j=$i+1; $j<$limit; $j++)
                {
                        if($order eq "asc")
                        {
                                if($arr[$j] < $arr[$extr]){$extr=$j;}
                        }
                        else
                        {
                                if($arr[$j] > $arr[$extr]){$extr=$j;}
                        }
                }
                my $tmp = $arr[$i];
                $arr[$i] = $arr[$extr];
                $arr[$extr] = $tmp;
        }
        return @arr;
}
```

here programmer requests a **COPY** of the parameter , all further operations are done on this copy

```
sub sort_array
{
#sorts an array in order specified in arg #2 ('desc', 'asc')
#sorts only arg #1 first elemnts
#array follows arg #1 and #2
        my $limit = shift @_;
        my $order = shift @_;

        for(my $i=0; $i<$limit-1; $i++)
        {
                my $extr = $i;
                for(my $j=$i+1; $j<$limit; $j++)
                {
                        if($order eq "asc")
                        {
                                if($_[$j] < $_[$extr]){$extr=$j;}
                        }
                        else
                        {
                                if($_[$j] > $_[$extr]){$extr=$j;}
                        }
                }
                my $tmp = $_[$i];
                $_[$i] = $_[$extr];
                $_[$extr] = $tmp;
        }
        return @_;
}
```

ORIGINAL parameters (by reference) are used

changes are made on the **ORIGINAL** parameters

Arguments

Variables passed to the subroutine are always passed in list context, i.e. as if they were one long array.

It works if we want to pass a few scalar variables followed by ONE array:

```
subroutine($var1, $var2, @arr);
```

We can recover variables in the subroutine if we know how many scalars are in the front:

```
my ($var1, $var2, @arr) = @_;
```

Arguments

Variables passed to the subroutine are always passed in list context, i.e. as if they were one long array.

It DOES NOT work if we want to pass more than one array, several arrays, or mixed scalars, arrays and hashes:

```
subroutine(@arr1, $var1, @arr2);
```

We cannot recover variables in the subroutine, Perl sees one long list, so in the example below ALL data will be assigned to `@arr1` while `$var2` and `@arr2` will stay empty

```
my (@arr1, $var2, @arr2) = @_;
```

script6.pl

```
#!/usr/local/bin/perl

my @arr1  = (1, 2, 3, 4, 5, 6, 7);
my @arr2 = qw(a b c d e f g h);
my $var1 = "Test string";

my_test_sub(@arr1, $var1, @arr2);

sub print_arr
{
        foreach my $entry (@_)
        {
                print "'$entry' ";
        }
        print "\n";
}

sub my_test_sub
{
        my (@arr1, $var1, @arr2) = @_;
        print "ARRAY 1: \n";
        print_arr(@arr1);
        print "Variable var1: '$var1'\n";
        print "ARRAY 2: \n";
        print_arr(@arr2);
}
```

References

It is possible to explicitly pass any variable by reference.

A variable is a named space of memory (RAM).

A reference is just a **pointer to the address** of this space.

The **backslash** operator produces a reference to any variable:

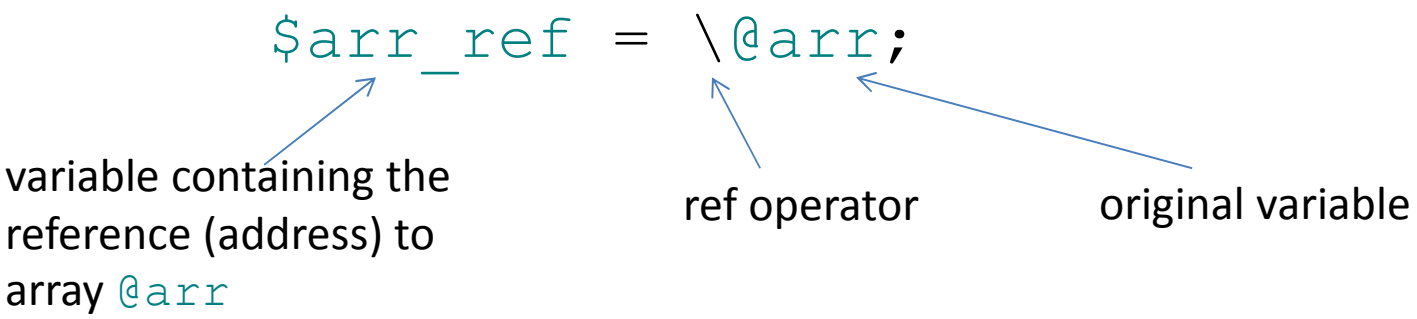
```
$var_ref = \$var;
```

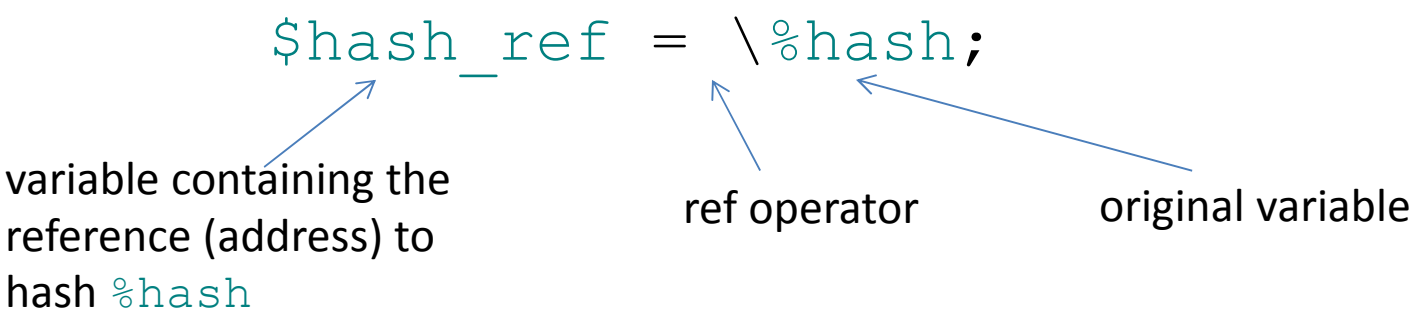
variable containing the reference (address) to scalar `$var`

ref operator

original variable

References


$arr_ref = \@arr;
variable containing the reference (address) to array @arr
ref operator
original variable


$hash_ref = \%hash;
variable containing the reference (address) to hash %hash
ref operator
original variable

References: de-referencing

How to recover the original variable from its reference?

```
$var = ${$var_ref};
$var = $$var_ref;

@arr = @{$arr_ref};
@arr = @$arr_ref;


%hash = %{$hash_ref};
%hash = %$hash_ref;
```

Arguments

We can pass more than one array, several arrays, or mixed scalars, arrays and hashes with references :

```
subroutine(\@arr1, $var1, \@arr2);
```

In the subroutine we have now a list of three values, the first and the last being references:

```
my ($arr1ref, $var2, $arr2ref) = @_;
@arr1 = @{$arr1ref};
@arr2 = @{$arr2ref};
```

script6a.pl

```
#!/usr/local/bin/perl
my @arr1  = (1, 2, 3, 4, 5, 6, 7);
my @arr2 = qw(a b c d e f g h);
my $var1 = "Test string";

my_test_sub(\@arr1, $var1, \@arr2);

sub print_arr
{
        foreach my $entry (@_)
        {
                print "'$entry' ";
        }
        print "\n";
}
sub my_test_sub
{
        my ($arr1ref, $var1, $arr2ref) = @_;
        @arr1 = @{$arr1ref};
        @arr2 = @{$arr2ref};
        print "ARRAY 1: \n";
        print_arr(@arr1);
        print "Variable var1: '$var1'\n";
        print "ARRAY 2: \n";
        print_arr(@arr2);
}
```

Local variables in subroutines

By default, all local (private) variables defined in a subroutine with **my** keyword vanish when the subroutine returns, i.e. their values are **NOT preserved** between the subroutine calls.

It is possible to declare persistent local subroutine variables which values **will be preserved** between the calls:

```
state $n = 0;
```

The variable `$n` is still private (cannot be accessed outside of the subroutine, same as if declared with **my**), but its value will persist during program execution.

Error handling in subroutines

If an error occurs in a subroutine there should be a pre-defined value returned indicating an error occurred.

If possible, a string with error description should be passed to the calling code block.

Alternatively, an error can be printed to STDERR or STDOUT.

Usually it is best to leave handling of the errors to the calling code block – sometimes you may want to ignore the error.

```
#!/usr/local/bin/perl
#script to list and count files with a given extension in a directory tree
if($#ARGV != 1)
{
        print "USAGE: script3.pl directory extension\n";
        exit;
}
my $base = $ARGV[0];
my $ext = $ARGV[1];
my ($n, $m, $nf, $nd, $counter) = listfiles($base, $ext);
if(! defined $n)
{
        print "ERROR: $listfileserror\n$!\n";
        exit;
}
print "----------------------\n";
print "There were $m of .$ext files out of total $n objects\n";
print "including $nf files total and $nd directories total\n";
print "in a directory tree starting from $base\n";
print "LISTFILES sub called $counter times\n";
```

```
sub listfiles
{
        $listfileserror = "";
        state $counter = 0;
        my ($dir, $ext) = @_;
        my $ntot = 0, $n_ext = 0, $nfiles = 0, $ndirs = 0 , $DIR;
        $counter++;
        if(!(opendir $DIR, $dir))
        {
                $listfileserror = "Cannot open directory $dir";
                return (undef, undef, undef, undef);
        }
```

persistent counter incremented with each call

if error occurs **undef** values are returned and error description passed via global variable

```
        foreach my $entry (readdir $DIR)
        {
                if($entry eq "." || $entry eq ".."){next;}
                $ntot++;
                if(-f "$dir/$entry")
                {
                        $nfiles++;
                        if($entry =~ /\.$ext$/)
                        {
                                print "$dir/$entry\n";
                                $n_ext++;
                        }
                }
                if(-d "$dir/$entry")
                {
                    $ndirs++;
                   my ($ntot1,$n_ext1,$nfiles1,$ndirs1)=listfiles("$dir/$entry",$ext);
                   $ntot += $ntot1;
                   $n_ext += $n_ext1;
                   $nfiles += $nfiles1;
                   $ndirs += $ndirs1;
                }
        }
        closedir($DIR);
        return ($ntot, $n_ext, $nfiles, $ndirs, $counter);
}
```

returned counter ignored on recursive calls

counter must be returned since it is local

Exercises

1. Take a close look at script2b.pl. There are several potential problems with this script, find them and modify the script to fix the problems.

2. Modify script4.pl to eliminate problem with symbolic links circular references.

3. Write a program computing factorial of an integer specified in its command line. Use a recursive function to compute factorial, make sure it handles errors (like invalid input) properly.

4. Write a program that computes total GC content of all sequences in a given fasta file (file name should be given as an argument). Use subroutines to make the code clean, understandable and reusable. Test the program on fasta file from session 8 (/home/jarekp/perl_08/fasta_in.fa).