

Bio 595

Computers in Biomedical Research

Fall 2003

Slides for Monday, November 17

Subroutines and “Scoping”

Note, with addition of subroutine(s), program now has two sections:

The ‘main’ (execution start to exit), and subroutine; put subroutine(s) at end of program. Actually, subroutines could be scattered throughout the code (but don’t).

Subroutines defined using reserved word ‘sub’. Block of code enclosed in curly braces: everything after the name.

Subroutines use two types of variables: (1) arguments passed to the subroutine, using special variable `*_`; (2) other variables declared with ‘my’ and restricted to the scope of the subroutine.

Subroutines (usually) return their results via ‘return’ function; could be a single scalar, or a list of scalars, or an array of lines.

Subroutines

In calling a subroutine, the names of the arguments used aren't important inside the subroutine. The subroutine gets values from the `@_` array and assigns them to new variables; these might or might not have the same names as the ones used in calling the subroutine. It's just the order of the variables given that's important.

Subroutines

First line of subroutine's block: `my($dna)=@_;`

Values of arguments from the call are passed into subroutine in the special array variable `@_`. (This is predefined in Perl; don't use it for your own arrays.)

The array `@_` contains all scalar values passed into subroutine. (in this example, there's only one, `$dna`). If there are more arguments, these are also passed; for example,

```
my($dna,$protein,$name_of_gene)=@_'
```

Omit this statement if there are no arguments to pass.

After `my($dna)=@_;` is executed, the passed variable is assigned to the subroutine's variable `$dna` (here the same name is used, but it doesn't need to be same).

Scoping

Keeping subroutine variables active only within the subroutine makes it possible to call the subroutines from anywhere. (Remember, you do this by declaring them as 'my' variables. Here, 'my' is a Perl keyword limiting range or scope to the current block of code.

Declaration: `my($x);` or just `my $x;`

Declaration can be combined with initialization:
`m($x)='49';` or, to collect an argument within a subroutine, `my($x)=@_;`

ex16

```
#!/usr/bin/perl -w
# ex16: illustrating the pitfalls of not using 'my' variables
$dna = 'AAAAA';
$result = A_to_T($dna);
print "I changed all the A's in $dna to T's and got $result\n\n";
exit;
```

```
# Subroutine
sub A_to_T {
    my($input) = @_;
    $dna = $input;
    $dna =~ s/A/T/g;
    return $dna;
}
```

ex16 examined

In the subroutine, Perl uses the same variable \$dna as is used in the main program. When the subroutine returned to the main program, it printed the same variable \$dna as had been used in the subroutine. Instead of printing the original DNA, it printed the subroutine-computed (altered) DNA.

Commonly used (and reused over and over) variable names: \$temp, \$x, \$a, \$number, \$variable, \$var, \$array, \$input, \$output, \$result, \$data, \$file, \$filename, \$dna, \$protein, \$sequence, etc.

Declare all your variables with 'my' variables. Also, the use of the directive "use strict;" will require that all variables be declared.

Note back in ex15, that because \$dna was declared within the subroutine, the value wasn't altered within the main program.

Command-line arguments, arrays

In ex17, a command line is used to provide information without requiring interactive answers to a prompt.

In ex17 also, note use of subscripts to access specific elements of an array.

Command-line programs: type name of program, followed by arguments to the program, if any, and then start program running via 'enter' key.

Alternative: use a GUI with menus and buttons to set parameters from the keyboard.

Command-line passing of parameters is useful with programs having a long execution time (e.g. searching sequence databases of gigabytes in size).

ex17

```
#!/usr/bin/perl -w
# ex17: counting the number of G's in some DNA on the command line
use strict;
# Collect the DNA from the arguments on the command line
# when the user calls the program.
# If no arguments are given, print a USAGE statement and exit.
# $0 is a special variable that has the name of the program.
my($USAGE) = "$0 DNA\n\n";
# @ARGV is an array containing all command-line arguments.
#
# If it is empty, the test will fail and the print USAGE and exit
# statements will be called.
unless(@ARGV) {
    print $USAGE;
    exit;
}
# Read in the DNA from the argument on the command line.
my($dna) = $ARGV[0];
# Call the subroutine that does the real work, and collect the result.
my($num_of_Gs) = countG ( $dna );
# Report the result and exit.
print "\nThe DNA $dna has $num_of_Gs G's in it!\n\n";
exit;

# Subroutine for ex17
sub countG {
    # return a count of the number of G's in the argument $dna
    # initialize arguments and variables
    my($dna) = @_;
    my($count) = 0;
    $count = ( $dna =~ tr/Gg// );
    return $count;
}
```

Some ex17 details explained

Every Perl program has an array variable `@ARGV` containing any command-line arguments.

There's also a variable called `$0` (a zero) containing the name of the program invoked from the command line.

In ex17, an informative message is displayed, giving the program name followed by the variables it needs.

Good practice: prompt the user if the program hasn't gotten the input it expects.

Here, the program checks `@ARGV`: if it contains something, the test in the program evaluates to 'true'; if empty, it evaluates to 'false'.

```
You could also say: unless(@ARGV) {  
    print $USAGE;  
    exit;  
}
```

Some ex17 details explained

**Extract an element from an array using a subscript:
remember the first element of an array is the 0th one.**

```
my($dna) = $ARGV[0];
```

**Later when we parse GenBank, PDB and BLAST files,
we'll use this structure to get various fields of data.**

Note that the element of the array @ARGV is \$ARGV[0].

```
my($num_of_Gs) = countG ( $dna );
```

Passing data to subroutines

Pass by value (or call by value):

(as demonstrated already) see ex18 for yet another example.

Pass by reference (or call by reference):

If a more complicated mix of data – some scalar, some array, some hashes (about these, later), if all arguments are passed to the same array `@_`, the elements get ‘flattened’ – elements of different arrays, for example, become one `@_` array and can’t easily be reassembled. See example ex19 to show the problem.

Pass by reference (or call by reference):

Any combination of scalars, arrays and hashes can be passed and can be distinguished in the subroutine.