Getting Organised

Subroutines, modules and the wonder of CPAN

Named Blocks

```
print '-' x LINE_LENGTH, "\n";

----------------------------------------------------------------------



print "=" x LINE_LENGTH, "\n";
print "-o0o-" x 12, "\n";
print "- " x 30, "\n";
print ">>==<<==" x 8, "\n";

======================================================================
-o0o--o0o--o0o--o0o--o0o--o0o--o0o--o0o--o0o--o0o--o0o--o0o--o0o--o0o-
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==
```

Introducing Subroutines

Maxim 5.1

Whenever you think you will reuse some code, create a subroutine

Calling Subroutines

```
drawline "=", REPEAT_COUNT;
drawline( "=", REPEAT_COUNT );
drawline;
drawline();
```

Naming Subroutines

```
drawline
find_a_sequence
convert_data

my_subroutine
sub1
tempsub
```

Creating Subroutines

```
sub drawline {

}

sub drawline { }

sub drawline
{

}

sub drawline
    {

    }
```

The drawline Subroutine

```
sub drawline {
    print "-" x REPEAT_COUNT, "\n";
}
```

Using drawline

```
#! /usr/bin/perl -w

# first_drawline - the first demonstration program for "drawline".

use constant REPEAT_COUNT => 60;

sub drawline {
    print "-" x REPEAT_COUNT, "\n";
}

print "This is the first_drawline program.\n";
drawline;
print "Its purpose is to demonstrate the first version
               of drawline.\n";
drawline;
print "Sorry, but it is not very exciting.\n";
```

Results from first_drawline ...

```
This is the first_drawline program.
-----------------------------------------------------
Its purpose is to demonstrate the first version of drawline.
-----------------------------------------------------
Sorry, but it is not very exciting.
```

Processing parameters

```
print "$_[0]"; # The first parameter.
print "$_[1]"; # The second parameter.
print "$_[2]"; # The third parameter, and so on.

sub drawline {
    print $_[0] x $_[1], "\n";
}


drawline;

drawline "-", REPEAT_COUNT;

drawline( "-", REPEAT_COUNT );

drawline "=", REPEAT_COUNT;
drawline( "-o0o-", 12 );
drawline "- ", 30;
drawline( ">>==<<==", 8 );
```

Using the new drawline

```
#! /usr/bin/perl -w

# second_drawline - the second demonstration program for "drawline".

use constant REPEAT_COUNT => 60;

sub drawline {
    print $_[0] x $_[1], "\n";
}

print "This is the second_drawline program.\n";

drawline "-", REPEAT_COUNT;

print "Its purpose is to demonstrate the second version of
              drawline.\n";
```

The second_drawline program, cont.

```
drawline "-", REPEAT_COUNT;

print "Sorry, but it is still not exciting. However, it is more
              useful.\n";

drawline "=", REPEAT_COUNT;

drawline "-o0o-", 12;

drawline "- ", 30;

drawline ">>==<<==", 8;
```

Results from second_drawline ...

```
This is the second_drawline program.
----------------------------------------------------
Its purpose is to demonstrate the second version of drawline.
----------------------------------------------------
Sorry, but it is still not exciting. However, it is more useful.
====================================================
-o0o--o0o--o0o--o0o--o0o--o0o--o0o--o0o--o0o--o0o--o0o--o0o-
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==
```

Better processing of parameters

```
shift; # Or like this: shift();


sub drawline {
    print shift() x shift(), "\n";
}


sub drawline {
    $chars = shift || "-";
    $count = shift || REPEAT_COUNT;

    print $chars x $count, "\n";
}
```

Using the new drawline ...

```
drawline "=== ", 10;

drawline;

drawline "=";       # Prints sixty equal signs.
drawline "*";       # Prints sixty stars.
drawline "$";       # Prints sixty dollars.

drawline 40;        # Does NOT print forty dashes!
drawline 20, "-";   # Does NOT print twenty dashes!
```

The fourth_drawline program

```
#! /usr/bin/perl -w

# fourth_drawline - the fourth demonstration program for "drawline".

use constant REPEAT_COUNT => 60;

sub drawline {
    $chars = shift || "-";
    $count = shift || REPEAT_COUNT;

    print $chars x $count, "\n";
}
```

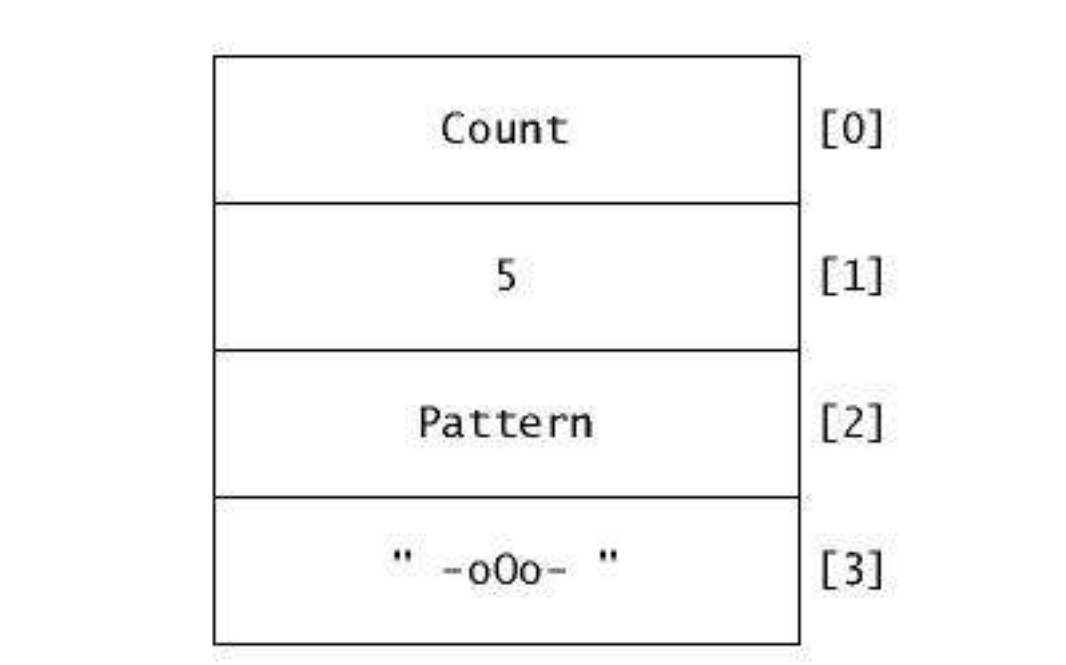
The fourth_drawline program, cont.

```
print "This is the fourth_drawline program.\n";

drawline;

print "Its purpose is to demonstrate the fourth version of
               drawline.\n";

drawline;

print "Sorry, but it is still not exciting. However, it is more
               useful.\n";

drawline "=", REPEAT_COUNT;
drawline "-o0o-", 12;
drawline "- ", 30;
drawline ">>==<<==", 8;
```

Even better processing of parameters

```
drawline;
drawline( Pattern => "*" );
drawline( Count => 20 );
drawline( Count => 5, Pattern => " -oOo- " );
drawline( Pattern => "===", Count => 10 );
```

The Default Array, @_, With Assigned Values



The %arguments Hash, With Assigned Values

Count	5
Pattern	" -oOo- "

Processing Named Parameters

```
%arguments = @_;

$chars = $arguments{ Pattern } || "-";
$count = $arguments{ Count } || REPEAT_COUNT;


sub drawline {
    $chars = $arguments{ Pattern } || "-";
    $count = $arguments{ Count } || REPEAT_COUNT;

    print $chars x $count, "\n";
}
```

The fifth_drawline program

```
#! /usr/bin/perl -w

# fifth_drawline - the fifth demonstration program for "drawline".

use constant REPEAT_COUNT => 60;

sub drawline {
    %arguments = @_;

    $chars = $arguments{ Pattern } || "-";
    $count = $arguments{ Count } || REPEAT_COUNT;

    print $chars x $count, "\n";
}
```

The fifth_drawline program, cont.

```
print "This is the fifth_drawline program.\n";

drawline;

print "Its purpose is to demonstrate the fifth version of
              drawline.\n";

drawline;

print "Things are getting a little more interesting.\n";

drawline( Pattern => "*" );
drawline( Count => 20 );
drawline( Count => 5, Pattern => " -oOo- " );
drawline( Pattern => "===", Count => 10 );
drawline;
```

Results from fifth_drawline ...

```
This is the fifth_drawline program.
------------------------------------------------------------
Its purpose is to demonstrate the fifth version of drawline.
------------------------------------------------------------
Things are getting a little more interesting.
************************************************************

--------------------
-oOo- -oOo- -oOo- -oOo- -oOo-
==============================
------------------------------------------------------------
```

A more flexible drawline subroutine

```
print "+";
drawline( Count => 15 );
print "+";
```

```
+---------------
+
```

Not Quite Right!

```
print "+";
drawline( Count => 15 );
print "+\n";

print "+", drawline( Count => 15 ), "+\n";

---------------+1+
```

Returning results

```
sub drawline {
     %arguments = @_;

     $chars = $arguments{ Pattern } || "-";
     $count = $arguments{ Count } || REPEAT_COUNT;

     return $chars x $count;
}
```

The boxes program

```
#! /usr/bin/perl -w

# boxes - the box drawing demonstration program for "drawline".

use constant REPEAT_COUNT => 15;

sub drawline {
    %arguments = @_;

    $chars = $arguments{ Pattern } || "-";
    $count = $arguments{ Count } || REPEAT_COUNT;

    return $chars x $count;
}
```

The boxes program, cont.

```
print "+", drawline, "+\n";

print "|", drawline( Pattern => " " ), "|\n";
print "|", drawline( Pattern => " " ), "|\n";
print "|", drawline( Pattern => " " ), "|\n";
print "|", drawline( Pattern => " " ), "|\n";
print "|", drawline( Pattern => " " ), "|\n";

print "+", drawline, "+\n";
```

Maxim 5.2

When determining the scope of a variable, think about its visibility

Visibility and Scope

```
#! /usr/bin/perl -w

# global_scope - the effect of "global" variables.

sub adjust_up {
    $other_count = 1;
    print "count at start of adjust_up: $count\n";
    $count++;
    print "count at end of adjust_up: $count\n";
}

$count = 10;
print "count in main: $count\n";
adjust_up;
print "count in main: $count\n";
print "other_count in main: $other_count\n";
```

Results from global_scope ...

```
count in main: 10
count at start of adjust_up: 10
count at end of adjust_up: 11
count in main: 11
other_count in main: 1
```

Using private variables

```
#! /usr/bin/perl

# private_scope - the effect of "my" variables.

sub adjust_up {
    my $other_count = 1;
    print "count at start of adjust_up: $count\n";
    $count++;
    print "other_count within adjust_up: $other_count\n";
    print "count at end of adjust_up: $count\n";
}

my $count = 10;
print "count in main: $count\n";
adjust_up;
print "count in main: $count\n";
print "other_count in main: $other_count\n";
```

Results from private_scope ...

```
count in main: 10
count at start of adjust_up:
other_count within adjust_up: 1
count at end of adjust_up: 1
count in main: 10
other_count in main:
```

Maxim 5.3

Unless you have a really good reason not to, always declare your variables with my

Using global variables properly

```
#! /usr/bin/perl

# hybrid_scope - the effect of "our" variables.

sub adjust_up {
    my $other_count = 1;
    print "count at start of adjust_up: $count\n";
    $count++;
    print "other_count within adjust_up: $other_count\n";
    print "count at end of adjust_up: $count\n";
}

our $count = 10;
print "count in main: $count\n";
adjust_up;
print "count in main: $count\n";
print "other_count in main: $other_count\n";
```

Results from hybrid_scope ...

```
count in main: 10
count at start of adjust_up: 10
other_count within adjust_up: 1
count at end of adjust_up: 11
count in main: 11
other_count in main:
```

Maxim 5.4

If you must use a global variable, declare it with our

The final version of drawline

```
sub drawline {
    my %arguments = @_;

    my $chars = $arguments{ Pattern } || "-";
    my $count = $arguments{ Count } || REPEAT_COUNT;

    return $chars x $count;
}
```

In-Built Subroutines

```
$ man perlfunc
```

```
$ perldoc -f sleep
```

Grouping and Reusing Subroutines

Maxim 5.5

When you think you will reuse a subroutine, create a custom module

Modules

```
#! /usr/bin/perl -w

use lib "$ENV{'HOME'}/bbp/";
use UsefulUtils;

drawline;
```

Modules – Example Template

```
package;

require Exporter;

our @ISA = qw( Exporter );

our @EXPORT = qw();
our @EXPORT_OK = qw();
our %EXPORT_TAGS = ();

our $VERSION = 0.01;

1;
```

The UsefulUtils Module

```
package UsefulUtils;

# UsefulUtils.pm - the useful utilities module from
# "Bioinformatics, Biocomputing and Perl".

require Exporter;

our @ISA = qw( Exporter );

our @EXPORT = qw();
our @EXPORT_OK = qw( drawline );
our %EXPORT_TAGS = ();

our $VERSION = 0.01;

use constant REPEAT_COUNT => 60;
```

The UsefulUtils Module, cont.

```
sub drawline {
    # Given: a character string and a repeat count.
    # Return: a string that contains the character string
    # "repeat count" number of times.
    #
    # Notes: For maximum flexibility, this routine does NOT
    # include a newline ("\n") at the end of the line.

    my %arguments = @_;

    my $chars = $arguments{ Pattern } || "-";
    my $count = $arguments{ Count } || REPEAT_COUNT;

    return $chars x $count;
}

1;
```

Installing UsefulUtils

```
$ mkdir ~/bbp/

$ cp UsefulUtils.pm ~/bbp/
```

Using UsefulUtils

```
#! /usr/bin/perl -w

# boxes2 - the box drawing demonstration program for "drawline".

use lib "$ENV{'HOME'}/bbp/";
use UsefulUtils qw( drawline );

print "+", drawline( Count => 15 ), "+\n";

print "|", drawline( Pattern => " ", Count => 15 ), "|\n";
print "|", drawline( Pattern => " ", Count => 15 ), "|\n";
print "|", drawline( Pattern => " ", Count => 15 ), "|\n";
print "|", drawline( Pattern => " ", Count => 15 ), "|\n";
print "|", drawline( Pattern => " ", Count => 15 ), "|\n";

print "+", drawline( Count => 15 ), "+\n";
```

The Standard Modules

```
$ man perlmodlib
```

Maxim 5.6

Don't reinvent the wheel, use or extend a standard module whenever possible

CPAN: The Module Repository

http://www.cpan.org

Maxim 5.7

Don't reinvent the wheel, use or extend a CPAN module whenever possible

Maxim 5.8

When you think others might benefit from a custom module you have written, upload it to CPAN

Searching CPAN

`http://search.cpan.org`

Installing a CPAN module manually

```
$ tar zxvf ExampleModule-0.03.tar.gz

$ cd ExampleModule-0.03

$ perl Makefile.PL

$ make

$ make test

$ su

$ make install

$ <Ctrl-D>
```

Testing the installation

```
$ man ExampleModule

$ perl -e 'use ExampleModule'



# Can't locate ExampleModule.pm in @INC.
# BEGIN failed--compilation aborted at -e line 1.
```

Installing a CPAN module automatically

```
$ perl -MCPAN -e "install 'ExampleModule'"
```

Maxim 5.9

Always take the time to test downloaded CPAN modules for compliance to specific requirements

Where To From Here