In this fourth session, we’ll go over some more of Perl. Specifically:

- **Toughening up** - warnings, strict and undef,
- Perl’s standard library of predefined functions,
- defining our own functions,
- making complex data structures with references.

Run-time Warnings

As our Perl scripts grow in size and complexity, there are several things we can ask Perl to do to **toughen its checking regime** to help us to catch more errors before they bite us.

- We have seen compile-time syntax check: `perl -cw program`
- However, not all warnings can be detected at compile-time, so try switching run time warnings on.
- There are two ways of enabling run-time warnings: the first is `perl -w script`.

The second way is to write:

```perl
use warnings;
```

near the top of your program. I recommend you switch warnings on, and fix every problem that causes a warning immediately.

Now, what sort of things give run-time warnings?

- We have mentioned several times that functions return the *undefined value* when they fail; that uninitialized scalar variables (and array and hash elements) have the *undefined value*, etc.
- Perl’s *undefined value* (written as `undef`) is analogous to the concept of a null pointer.
- It is different from the empty string `''` – the empty string is a *string of length 0* whereas `undef` is not a string at all.
- However, `undef` behaves like the empty string in string contexts, like 0 in numeric contexts, and like false in boolean contexts.

The program runs fine without warnings, but with `-w`, a use of uninitialized value `$testval` in a string warning appears.

To fix this, we must decide how to display `undef`, and test for defined-ness using the function defined, as in:

```perl
my $display = defined $testval ? $testval : "UNDEF";
print "$display: <$testval>, $boolstr
```

This form of "value or default" is so common, Perl 5.10 introduces a new operator:

```perl
my $display = $testval // "UNDEF";
```
If you put the pragmas:

```perl
use strict;
use warnings;
```

near the top of your programs then Perl will perform stricter syntax checks for you. This has several effects - for example, all warnings become fatal errors.

Another way of enabling strict mode, and also enabling neat new Perl features like `say` and `given/when` is:

```perl
use v5.12;
use warnings;
```

Let's try adding `use strict` to `eg1` and see what error messages `perl -cw eg1` generates.

Oops! We forgot to declare some of our variables: under `use strict`, Perl insists that you declare all your variables properly using `my`.

Let's spend a moment making `eg1` work in strict mode - declare all our variables.

Note that we've been declaring our variables all along (or trying to!), but we didn't need to do so until now.

---

The `my` declarations that we've been using declare lexical variables which exist only for the duration of a particular lexical scope (for example, in a particular block).

Inside a block they are like local variables - outside of a block, they exist from the point of declaration down to the bottom of the Perl script - so are effectively global variables.

Most of the time, we declare and initialize variables at the same time, but you can declare one or many variables without initializing them by any of:

```perl
my $a;
my( $x, $y, @z );
( my $x, my $y, my @z );
```

There's another type of global variable, called package variables, declared by replacing `my` with `our`, as in:

```perl
our( $x, $y, @z );
```

Later on, we'll see a few places where package variables are needed, but for now I recommend that you use `my` variables everywhere until further notice.
Perl's Predefined Functions

@result = sort( list )

Sort an array such that each element is placed in an ascending order - by default into alphabetical order. You can specify a different sort order, such as numeric, using the highly magical:

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

Consult perldoc -f sort for more information.

@result = reverse( list )

$result = reverse( scalar )

In list context, reverse an array (such that each element is placed after all its predecessors). In scalar context, reverse a single string.

@result = glob( wildcard )

This performs a shell compatible file glob - returning a list of files which match the wildcard. For example glob(".*\.[ch]") matches all '.c' or '.h' files in the current directory.

@len = length( string )

Length of string; equivalent of C's strlen.

printf( 'format string', args );

$text = sprintf( 'format string', args )

When you need more formatting than print can do - use printf and sprintf. These are closely modelled on the C functions and are much too complex to explain here... For example, eg2:

my $string = 'pi'; my $pi = 3.1415926536;
printf( "<'$pi'>\<\$pi>\<\string>\<\$pi>");
would produce output:

<pi>< 3.14159265>

See perldoc -f sprintf for more details.

@result = grep ( expr ) list

This function evaluates the expression for each element in the list (with the element itself stored in a localized copy of $_), and extracts the elements for which the expression is true.

It is equivalent to (but more efficient than):

@result = ();
foreach $_ (list) { push @result, $_ if expr; }

Common use - pretend we're the Unix grep utility:

my @result = grep { /he*llo/ } @array;

Aside: We'll see lots of uses for map and grep in the 7th lecture.

map

map is very similar to the functional programming (Haskell etc) map function, which applies an operation to every element of an array, building a new array.

Just like grep, within the operation the current array element is stored in a localized $_.

The operation is any valid Perl expression - so, for example, eg3:

my @orig = (1,2,5,8,9,10,5);
my @doubled = map { $_ * 2 } @orig;
print join(',', @doubled)."

will double all the original numbers and then print the results in comma-separated format.

You can also use map destructively - if you modify $_ the original element is modified (eg4):

my @array = (1,2,5,8,9,10,5);
map { $x *= 2 } @array;

does the same but modifies @array in place.

Aside: We'll see lots of uses for map and grep in the 7th lecture.

map cont:

The operation in a functional map always receives a single array element, but it can return a list of scalars rather than a single scalar, in this case all the little lists are appended together.

For example, eg5:

my @orig = (1,2,5,8,9,10,5);
my @doubled = map { ( $_, $_*2 ) } @orig;
generates a pair ( x, 2x ) from each element of the original array, thus setting @result to a flat list twice as long - (1,2,2,5,8,9,10..).

This is often used to turn an array into a hash - when you assign a flattened list of (key,value) pairs to a hash, Perl initialises it pairwise. eg6:

my @orig = (1,2,5,8,9,10,5);
my %doubled = map { ( $_, $_*2 ) } @orig;

The comma can be written as => to look nicer:

my %doubled = map { $x => $x*2 } @orig;

One use of this is to turn an array into a set hash:

my @set = map { $x => 1 } @orig;
Defining Our Own Functions

Practically all languages provide functions, subroutines or procedures. In Perl, they are called subroutines.

You decide on a coherent block of code with a nameable purpose, for example, sum up an array and return the total, and give it a name like sumarray.

Decide what arguments sumarray takes, and what results it returns. I find the easiest way is to write down a typical call:

my $total = sumarray( @x );

This means: call function sumarray, passing the array @x in to it, and storing the scalar value that is returned into $total.

Then write the outer shell of your function as a sub declaration, including a comment describing the function’s purpose:

# my $total = sumarray( @array ): # sum up the elements of the @array.
# sub sumarray {

Now, inside the {}, write the body of the function:

my( @array ) = @_;    # copy @_ into my @array
my $total = 0;        # sum up the elements of the @array.
foreach my $elem (@array) {   # loop variable
    $total += $elem;
}
return $total;

Perl flattens all the arguments in a subroutine call into a single list, called @_. Note that the original argument array elements will change if you change @_ elements.

Perl’s local variables are our old friends - lexically scoped ‘my’ variables.

They spring into existence when we enter the function, eclipsing existing variables of the same names, and disappear when we leave the function.

In sumarray, we copy @_ into ‘my @array’ (to avoid any possibility of changing the parameters). Then declare two additional ‘my’ variables - $total and loop variable $elem.

Finally, to communicate the final result back to the caller we use return $total. This destroys all the function’s local variables.

Putting the whole program together (giving eg7):

#!/usr/bin/perl
#
# eg7: sum up the elements of an array,
# using a separate subroutine.
use strict;
use warnings;
#
# my $total = sumarray( @array ): # sum up the elements of the @array.
# sub sumarray {

Prototype were added in Perl 5.6 and allow us to specify how many parameters a subroutine takes. In a subroutine header write:

sub fred ($$@) # must have args > 1

This declares that fred must be called with two scalars and a (possibly empty) array. If Perl has already seen a prototype declaration for sub fred when it parses a call to fred it will produce a warning unless there are at least two scalar arguments.

One option is to separate the prototypes from the definitions:

# first declare the prototypes:
sub fred ($$@); # must have args > 1
sub bob ($;$);  # must have args == 1 or 2

# define the subroutines from here on, any order:
sub fred {
    my( $a, $b, @rest ) = @_; # arg2 may be undef
    ..... }
Defining Our Own Functions

Prototypes

Prototypes are not perfect; they're likely to undergo more change in future. They don't affect the fact that all arguments to a function call are still flattened into a single list - so you can't just say `sub fred (@@%)` and pass two whole arrays and a hash to `fred`... To do this, you have to use Perl references - read on.

Note that a Perl subroutine can return a scalar, an array or a hash. So for example it's fine to think of a subroutine as returning a `tuple`, as in:

```perl
my ($a, $b) = callme(arguments); return ($x, $y);
```

Exercise: take any of the programs that you've already done in previous sessions and restructure it into several functions with separate prototypes at the top.

Exercise: Choose some simple recursive function - perhaps `fibonacci`, `factorial` or `quicksort` - code it up in Perl, get it working and thus convince yourself that there's nothing abnormal about recursion in Perl. In particular, convince yourself that each recursive call has its own local argument array, and its own local set of `my` variables.

---

Defining Our Own Functions

Experimental Feature: `Function::Parameters`

Recently, I discovered a Perl module called `Function::Parameters` that introduces a more convenient syntax for defining functions (eg8):

```perl
use Function::Parameters qw(:strict);
fun hello( $x, $y = 10 ) # 10 is a default value for $y
{
    print "hello: x=$x, y=$y\n";
}

hello( 1 );
hello( 1, 2 );
hello( 1, 2, 3 ); # error "too many arguments"
```

This module requires Perl >= 5.14 (so ok on DoC Linux lab machines, shell servers, and on the webserver).

But this new syntax doesn't affect the fact that all arguments are flattened to a single list: `fun fred(@x, @y)` is an error.

Really new stuff: in Perl 5.20, an experimental subroutine parameter syntax has been added (after how long?). When 5.20 is installed you will be able to write:

```perl
use v5.20;
sub hello( $x, $y = 10 ) # 10 is a default value for $y
{
    print "hello: x=$x, y=$y\n";
}

hello( 1 );
hello( 1, 2 );
hello( 1, 2, 3 ); # error "too many arguments"
```

---

References

In Perl 4, the only data structures that existed were scalars, arrays of scalars and hashes of scalars. No multi-dimensional array facilities (array of array ... of scalars) were provided.

In Perl 5, Larry Wall decided to graft multi-dimensional structures into the language. Rather than change the whole term syntax, he did it by adding a new type of scalar - a `reference`.

A reference is very like a pointer in C. To make a reference, use the backslash operator (like C's address-of operator, &):

```perl
my $x = 10;
my $ref = \$x;
```

$ref now refers to (or points to) $x. Dereferencing is done by using $ref instead of a variable name:

```perl
print "before: x is \$x\n";
print "before: ref refers to x - value \$ref\n";
$ref++;  
print "after: x is \$x\n";
```

Make this into a program eg9 and try it out...

You can also make a reference to an array (eg10):

```perl
my @a = ( 54, 17, 23 );
print "before: \". join(\',\', @a) . \"\n";
my $ref = @a;
$ref[2] = 18; # overrides '23' value
print "after: access via ref: " . join(\',\', @a) . \"\n";
```

You can also make an anonymous array ref (eg11):

```perl
my $ref = [ 54, 17, 23 ];
print "before: access via ref: " . join(\',\', $ref) . \"\n";
$ref->[2] = 18; # overrides '23' value
print "after: access via ref: " . join(\',\', $ref) . \"\n";
```

You can also have references to hashes (eg12):

```perl
my %hash = ( "duncan" => "d.white", "bilbo" => "b.baggins" );
my $ref = \%hash;
$ref->{frodo} = "f.baggins"; # stores a new key, value pair
while( my($key,$value) = each(%$ref) ) # now print all pairs out
{
    print "$key => $value\n";
}
```
We can declare anonymous hash refs (eg13):

```perl
my $ref = { duncan => "d.white", bilbo => "b.baggins"; };
$ref->{frodo} = "f.baggins"; # store a new key, value pair
delete $ref->{duncan}; # deletes a k,v pair
while( my($key,$value) = each(%$ref) ) # print out all pairs
{
    print "$key => $value
";
}
```

We can now create hashes of hashes, arrays of arrays, or any combination. For instance:

```perl
my @fred = ( 
    { "one" => "ena", "two" => "duo" }, 
    { "three" => "drei" }, 
    { "one" => "une", "two" => "deux" } 
); 
@fred
```

@fred is an array of references to hashes: $fred[$r] is now a reference to one hash, %{$fred[$r]} is one whole hash, and $fred[$r]->{$c} is a single element. This last can, as a special convenience, be written as $fred[$r]{$c}.

Thus, it looks like a multi-dimensional array, but it isn’t really!

Like C’s pointers to functions, you can take a reference to a function (called a coderef in Perl) and call it later through the reference - eg16 (and eg16a for the Function::Parameters equivalent):

```perl
sub double ($) # eg16a: fun double( $n )
{
    my( $n ) = @_; 
    return 2 * $n; # return 2 * $n; 
}
```

```perl
my $coderef = \&double; # make reference to function
my $x = $coderef->(10); # invoke: dereference and call with arg 10
print "10 doubled is $x
";
```

Perl also allows us to create anonymous coderefs on the fly, eg17 and eg17a:

```perl
my $doubleme = sub { return 2 * $_[0]; }; # eg17a: fun ($n) { return 2 * $n };
my $x = $doubleme->(10); # invoke: deference and call with arg 10
print "10 doubled is $x
";
```

In more complex examples, the coderef $doubleme might refer to any scalar -> scalar function, so when invoked, it might do anything!

Using coderefs, you can do lots of cool functional programming - higher order functions, callbacks, data-driven programming, factories, iterators, lazy evaluation. Wait for the 7th lecture!