Perl Short Course
Fourth Session

Duncan C. White
dcw@doc.ic.ac.uk

- In this fourth session, we'll go over some of the more advanced parts 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**.

Warnings, Strict and the Undefined Value

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

- We have already seen that you can (must!) perform a *compile-time syntax check with warnings*:

  ```
  perl -cw program
  ```

- However, not all warnings can be detected statically at compile-time, so it is good to switch *run time* warnings on - to check *dynamically*.

- There are two ways of enabling run-time warnings: the Perl flag '`-w`' tells Perl to switch warnings on, as in `perl -w script`. On Unix, we can put '`-w`' into the hash-bang line: `#!/usr/bin/perl -w`

- On other platforms, the hash-bang trick doesn't work, fortunately (newish) Perl allows you to say:

  ```
  use warnings;
  ```

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

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

The Undefined Value

- We have mentioned several times that functions return *the undefined value* when they fail; that scalar variables (or array elements) that have not been initialized have *the undefined value*, etc.

- Perl's *undefined value* (written as `undef`) is analogous to the concept of a *null pointer* or to the denotational semantics concept of *bottom*.

- It is *not the same as* the empty string `''` – the empty string is a *string of length 0* whereas `undef` is not really a string at all - it is *undefined*.

- However, **undef** behaves like the empty string in string contexts, like 0 in numeric contexts, and like false in boolean contexts.

- Let's investigate boolean-ness and defined-ness:

```
#
# eg1: play around with empty strings and undef
#
@pairs = ( 0,       "zero",
           '',      "emptystr",
           undef,   "undef",
           1,       "one",
           17.3,    "17.3",
           'hello', "hello" );
# foreach (testval,label) in @pairs
while( ($testval,$label,@pairs) = @pairs )
{
        $boolstr = $testval ? "true" : "false";
        print "$label: <$testval>, $boolstr\n";
}
```

- Run without warnings, the program runs fine, but when '`-w`' is given, Perl warns you that you are interpolating an undefined value into a double-quoted string.

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

```
$display = defined $testval ? $testval : "UNDEF";
print "$label: <$display>, $boolstr\n";
```

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

```
$display = $testval // "UNDEF";
```

Strict Mode

- If you put the pragmas:

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

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

- However, the most immediately obvious is that you need to declare all your variables before using them. Let's try adding the above to `eg1` and see what error messages `perl -cw eg1` generates.

  There are 3 different ways of declaring variables:

- Firstly, the long established `use vars` pragma:

  ```
  use vars qw($testval $label @pairs $boolstr $display);
  ```

  Note that we declare all our variables (scalars, arrays and hashes - but not filehandles).

- Secondly, from Perl 5.6 onwards, declare using the new '`our`' syntax, which is similar to the above (but subtly different):

  ```
  our($testval, $label, @pairs, $boolstr, $display);
  ```

- All the variables we have seen so far are essentially global variables (actually called *package variables*, more later) - which are accessible via the main symbol table anywhere in the program.

- However, **Perl 5** added support for a second kind of variable - called *lexical variables* (written '`my`') these exist only for the duration of a particular lexical scope (between a particular '`{`' and it's corresponding '`}`'). They are like *local variables* - but you can declare them outside of functions, and they then exist from the point of declaration down to the bottom of the Perl script. Declare them via:

  `my($testval, $label, @pairs, $boolstr, $display);` or

  ```
  my $testval;
  my $label;
  my @pairs;
  my $boolstr;
  my $display;
  ```

- You can also declare and initialize '`my`' variables by prefixing the first assignment with '`my`', as in:

  ```
  my $x = 1;
  my $y = my $z = 2;
  my @arr = (1,2,3);
  ```

- Since Perl 5.6 you can use '`my`' in a `foreach` loop to localize the loop variable:

  ```
  foreach my $element (@array)....
  ```

- I recommend that you use '`my`' variables nearly everywhere, and use '`our`' where necessary. Only use the older `use vars` syntax if you find yourself using an old Perl interpreter. Let's finish making **eg1** work in strict mode.

Perl's Predefined Functions

- Perl has hundreds of predefined functions. Let's look at a few of the most useful ones - consult the Llama or Camel book or `perldoc perlfunc` for more details.

- `@arr = split( /regex/, string, n )`

  - Splits a string into an array of pieces, splitting the string wherever the regex matches - i.e. the regex indicates what *separates* the interesting parts.

  - If no value of `n` is given, the maximum number of splits are performed. If `$n` is given, the string is split into exactly `$n` pieces, the last piece contains the rest of the string.

  - If no string is given then `$_` is used by default. A common use of this is to split `$_` into whitespace separated 'words' or 'tokens':

    ```
    my @wd = split( /\s+/ );
    ```

- `$str = join( sep_string, array )`

  - This function joins the elements of the array together, using the string as a separator.

  - A common use is to join array elements together with separators ready for printing.

  - For instance:

    ```
    print join(',', split( /\s+/ ) );
    ```

  - Is there any other way of doing this?

- `push( @array, list )`
  - This function appends the given list (or single scalar) to the end of an array.
  - Common use is to accumulate entries into an array:

    ```
    push( @done, $filename );
    ```

- `$x = pop( @array )`
  - This function removes the last element from the array and returns it. The inverse operator to `push`.
  - Common use (with `push`) is to simulate a stack:

    ```
    push( @stack, ( 3, 2 ) );
    $x = pop( @stack );   # gets 2
    $y = pop( @stack );   # gets 3
    ```

- `$x = shift( @array )`
  - As we've already seen, this function extracts the first element of the array, removing that element from the array, shifting every other element down one space.

- `unshift( @array, list )`
  - This is the opposite of `shift` - the list (or single scalar) is inserted into the array at the front, shifting all existing elements up out of the way.

- `@result = sort( list )`
  - This function sorts an array such that each element is placed in an ascending order - by default into alphabetical order.
  - Common use is to sort hash keys into alphabetical order:

    ```
    foreach my $key (sort(keys(%assoc)))
    {
      ...
    }
    ```
  - It is possible to specify a different order, such as numeric, using the highly magical :

    `@sorted = sort { $a <=> $b } ( @array )`
  - Consult `perldoc -f sort` for more information.
- `@result = reverse( list )`
  - This function reverses an array (such that each element is placed after all its predecessors).
- `@result = glob( wildcard )`
  - This performs a shell compatible file glob - returning a list of files which match the wildcard. For example `glob("*.[ch]")` matches all '`.c`' or '`.h`' files in the current directory.
- `$len = length( string )`
  - This trivial function is the equivalent of C's `strlen` - it returns the length of the given string expression.

- `$str = 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( "<%-10s><%12.8f>\n", $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, select elements that match a regex:

    ```
      my @result = grep( /he*llo/, @array );
    ```

- `@result = map { operation } @array;` or `map { destructive operation } @array;`
  - `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)."\n";
    ```

    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 { $_ *= 2 } @array;
    ```

    does the same but modifies `@array` *in place*.
  - The map operation can build a list rather than a single scalar - so one element comes in but a little list goes out; in this case all the *little lists* are appended together.

- `map` cont:
  - For example, **eg5**:

    ```
    my @orig = (1,2,5,8,9,10,5);
    my @result = 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 - `(1,2,2,4,5,10..)`.
  - This is often used to turn an array into a *hash* (because if you assign a flattened list of (key,value) pairs to a hash, Perl initialises it pairwise), as in **eg6**:

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

  - In this case, the comma can be written as `=>` - as in `$_ => $_*2` - largely because it looks nicer!

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.

- The easiest way of showing this is to write down a typical call:

```
$sum = sumarray( @arr );
```

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

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

```
#
# $sum = sumarray( @array ):
#     sum the elements of the @array.
#
sub sumarray
{
}
```

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

```
my( @array ) = @_;
my $total = 0;
foreach my $elem (@array)
{
        $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 attempt to change any values in `@_`.

- Perl's *local variables* are precisely the lexically scoped 'my' variables that we saw a few slides back.

- They spring into existence when we enter the function, eclipsing any global variables of the same names while we're inside the function.

- They also disappear when we leave the function – leaving any previous (global or lexical) variables of the same names completely unchanged.

- In `sumarray`, we take a copy of `@_` into a 'my' variable `@array` (for clarity and to avoid any possibility of changing the arguments passed in). Then we declare ourselves two additional 'my' variables `$elem` and `$total`.

- Finally, to communicate the final result back to the caller we use `return $total`. This simultaneously leaves the function, returns the result value and destroys all the function's lexical variables.

- On the next slide we'll put the whole program together, building an array of your choice, calling `sumarray` and printing out the result:

```
#!/usr/bin/perl
#
# eg7: sum up the elements of an array,
#      using a separate subroutine.

use strict;
use warnings;

#
# $sum = sumarray( @array ):
#     sum the elements of the @array.
#
sub sumarray
{
        my( @array ) = @_;
        my $total = 0;
        foreach my $elem (@array)
        {
                $total += $elem;
        }
        return $total;
}

# main program
my @x = @ARGV > 0 ? @ARGV : (10, 39, 45, 28, 49, 3);
my $sum = sumarray( @x );
my $str = join(',', @x );
print "sum of $str is $sum\n";
```

Prototypes

- *Prototypes* were added in Perl 5.6 and allow us to specify how many parameters a subroutine takes. The extra syntax is very simple - in a subroutine header write something like:

  ```
  sub fred ($$@)
  ```

- 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.

- The easiest way to ensure that Perl has seen prototype declarations for subroutines before they are called is to separate the prototype headers from the subroutines:

  ```
  # first declare the prototypes:
  sub fred ($$@);
  sub bob ($);
  ......

  # define the subroutines from here on, any order:
  sub fred
  {
          my( $a, $b, @rest ) = @_;
          ......
  }

  sub bob
  {
          my( $arg ) = @_;
          ......
  }
  ```

- Prototypes are not perfect, they're likely to undergo more change in Perl 6. 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:

  ```
  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 it's own local argument array, and it's own local set of `my` variables.

- Exercise: Try taking some medium complexity set of interdependent Haskell functions that you are familiar with, and translate them into Perl. Most functional programs are largely trivial to translate into Perl.

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, `&`):

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

- `$ref` now refers to (or points to) `$x`. We can read or change `$x` via it's reference, simply by using `$ref` instead of a variable name:

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

  Make the above into a program `eg8` and try it out...

- You can also make a reference to an array (`eg9`):

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

- Note that `$$ref[2]` binds like `${$ref}[2]`. This `$$ref[$n]` syntax is so unpleasant that Perl gives us a nicer alternative: `$ref->[$n]`.

- You can also refer to the whole array as `@$ref`.

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

```
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 (`eg11`):

```
my %hash = ( "duncan" => "d.white", "bilbo" => "b.baggins" );
my $ref = \%hash;
$$ref{"frodo"} = "f.baggins";   # stores a new key, value pair
while( my($k,$v) = each(%hash) )# now print all k,v pairs out
{
     print "$k => $v\n";
}
```

- Once again, the `$$ref{$key}` syntax is so unpleasant that Perl gives us the syntactic sugar `$ref->{$key}`.
- We can also declare anonymous hash references (eg12):

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

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

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

- `@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 very like a multi-dimensional array, but it isn't really!

- How might we printout such a complex data structure as `@fred` (an array of references to hashes)? There are two ways:

- Write our own function, carefully tailored to the exact specification we want (`eg13`):

```
foreach my $hashref (@fred)
{
    my @x = ();
    foreach my $k (sort keys %$hashref)
    {
        my $v = $hashref->{$k};
        push @x, "$k->$v";
    }
    print join( ", ", @x ). "\n";
}
```

- Or, alternatively, use a Perl module (our first!!) specifically designed to navigate and print reference structures (`eg14`):

```
use Data::Dumper;

print Dumper \@fred;
```

- How does that work? It's complicated... However, the core of it uses a Perl function `ref()` which takes a reference and returns a string such as '`HASH`' to tell you what the reference is currently referring to. Using that information, a reference navigator can be written pretty easily.

References to Functions

- You can also take a reference to a function (a subroutine) and call it (later) through the reference - just like C's *pointer to function* concept:

```
sub fred ($)
{
    ...
}

my $funcref = \&fred;  # ref to function

...

&$funcref( 10 );       # calls fred with arg 10
```

- As you might expect by now, there's a syntactic sugar form of the call:

```
$funcref->(10);
```

- Using function references, you can mess about as much as you like with higher order functions. `map` and `grep` help too.

- Finally, Perl also allows us to create anonymous function references on the fly, as in:

```
my $doubleme = sub { return 2 * $_[0]; };
my $x = $doubleme->(10);
print "10 doubled is... wait for it.. $x\n";
```

Exercises

- In the last session, there was a multi-part exercise that attempted to build word frequency indexes of files. We can do more with that example now:

- Use `split` to allow the indexing program to split each line into multiple words and index each word.

- Now convert the indexer program into several separate functions, nicely laid out and documented. Add prototypes, declare your variables and `use strict`.

- Taking the indexer further still, we might want to record when each data file was last indexed. Then, a `reindex` program could be written to check the modification time of each indexed document file and reindex any documents that have been modified.

- Hint: use Perl's `stat` function to find a file's modification timestamp (see `perldoc -f stat` for details).

- Familiarise yourself with complex references - use the `Data::Dumper` module to print them out.

- Modify `eg13` replacing the entire inner `foreach` loop that builds the `@x` array with a `map` invocation that begins `my @x = map ....` If you're feeling really brave, you can remove `@x` altogether and make the body of the outer `foreach` a single statement beginning `print join( ", ", map....`