Perl Short Course: Fourth Session

Duncan C. White (d.white@imperial.ac.uk)

Dept of Computing,
Imperial College London

December 2012

- In this fourth session, we'll go over some more of Perl. Specifically:
  - **Toughening up** - `warnings`, `strict` **and** `undef`,

- 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**,

- 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**,

- 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**.

- 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.

- 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`

- 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.

- 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:

  ```
  use warnings;
  ```

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

- The second way is to write:

  ```
  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.

- The second way is to write:

  ```
  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*.

- The second way is to write:

  ```
  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.

- The second way is to write:

  ```
  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.

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

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

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

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

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

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

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

- The program runs fine without warnings, but with ‘`-w`’, a *use of uninitialized value $testval in a string* warning appears.
- To fix this, decide how to display undef, and test for defined-ness using the function `defined`, as in:

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

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

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

- The program runs fine without warnings, but with ‘`-w`’, a *use of uninitialized value $testval in a string* warning appears.
- To fix this, decide how to display undef, and test for defined-ness using the function `defined`, as in:

```
my $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:

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

- 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 - for example, all warnings become fatal errors.

- 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 - for example, all warnings become fatal errors.
- Let's try adding `use strict` to eg1 and see what error messages `perl -cw eg1` generates.

- 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 - for example, all warnings become fatal errors.
- 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`).

- 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 - for example, all warnings become fatal errors.
- 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.

- 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 - for example, all warnings become fatal errors.
- 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, but actually we didn't need to do so until we encounter strict mode.

- 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).
- They are like *local variables* - but if you declare them outside of a block, they exist from the point of declaration down to the bottom of the Perl script - and 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:

```
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:

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

- 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).
- They are like *local variables* - but if you declare them outside of a block, they exist from the point of declaration down to the bottom of the Perl script - and 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:

```
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:

```
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 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.

- 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.
- `@pieces = 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.

- 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.
- `@pieces = 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+/ );
    ```

- 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.
- `@pieces = 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 given string as a separator, i.e. between every pair of elements.
  - For example:

    ```
    print join(',', @wd );
    ```

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

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

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

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

- `$x = pop( @array )`
  - Remove the last element from the array and return it. Common use: (with `push`) implement a stack:

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

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

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

- `$x = pop( @array )`
  - Remove the last element from the array and return it. Common use: (with `push`) implement a stack:

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

- `$x = shift( @array )`
  - This extracts the first element of the array, removing that element from the array, shifting every other element down one space.

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

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

- `$x = pop( @array )`
  - Remove the last element from the array and return it. Common use: (with `push`) implement a stack:

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

- `$x = shift( @array )`
  - This extracts the first element of the array, removing that element from the array, shifting every other element down one space.
- `unshift( @array, list )`
  - 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.

- `@result = sort( list )`
  - This function sorts 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 :
    `@sorted = sort { $a <=> $b } ( @array )`
  - Consult `perldoc -f sort` for more information.

- `@result = sort( list )`
  - This function sorts 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 :
    `@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 = sort( list )`
  - This function sorts 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 :
    `@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.

- `@result = sort( list )`
  - This function sorts 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 :
    `@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.

- `printf( "format string", args );`
  `$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.

- `printf( "format string", args );`
  `$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.

- `printf( "format string", args );`
  `$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; }
    ```

- `printf( "format string", args );`
  `$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:

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

- `@result = map { 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.

- `@result = map { 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 `$_`.

- `@result = map { 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.

- `@result = map { 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*.

- `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.

- 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 @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..).

- 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 @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 - when 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;
    ```

- `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 @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 - when 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;
    ```

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

    ```
    my %double = map { $_ => $_*2 } @orig;
    ```

- 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 @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 - when 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;
    ```

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

    ```
      my %double = map { $_ => $_*2 } @orig;
    ```

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

    ```
      my %set = map { $_ => 1 } @orig;
    ```

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

- 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`.

- 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:

  ```
  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`.

- 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:

  ```
  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 ) = @_;
my $total = 0;
foreach my $elem (@array)
{
  $total += $elem;
}
return $total;
```

- 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 change @_ elements.

- 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 change @_ elements.
- Perl's *local variables* are our old friends - lexically scoped 'my' variables.

- 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 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.

- 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 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`.

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

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

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

```
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.

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

```
  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.
- 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 ) = @_;
        ......
}

sub bob
{
        my( $arg1, $arg2 ) = @_;   # arg2 may be undef
        ......
}
```

- 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.

- 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:

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

- 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:

```
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.

- One is always learning new stuff in Perl: Less than a week ago, I discovered a Perl module called `Function::Parameters` that introduces a more convenient syntax for defining functions (**eg8**):

```
use Function::Parameters qw(:strict);

fun hello( $x, $y = 10 )
{
        print "hello: x=$x, y=$y\n";
}

hello( 1 );
hello( 1, 2 );
hello( 1, 2, 3 );
```

- One is always learning new stuff in Perl: Less than a week ago, I discovered a Perl module called `Function::Parameters` that introduces a more convenient syntax for defining functions (**eg8**):

```
use Function::Parameters qw(:strict);

fun hello( $x, $y = 10 )
{
        print "hello: x=$x, y=$y\n";
}

hello( 1 );
hello( 1, 2 );
hello( 1, 2, 3 );
```

- This module requires Perl >= 5.14 (so ok on DoC linux lab machines, not yet ok on the webserver).
- But this new syntax doesn't affect the fact that all arguments to a function call are still flattened into a single list. `fun fred (@x, @y)` is an error.

- One is always learning new stuff in Perl: Less than a week ago, I discovered a Perl module called `Function::Parameters` that introduces a more convenient syntax for defining functions (**eg8**):

```
use Function::Parameters qw(:strict);

fun hello( $x, $y = 10 )
{
        print "hello: x=$x, y=$y\n";
}

hello( 1 );
hello( 1, 2 );
hello( 1, 2, 3 );
```

- This module requires Perl >= 5.14 (so ok on DoC linux lab machines, not yet ok on the webserver).
- But this new syntax doesn't affect the fact that all arguments to a function call are still flattened into a single list. `fun fred (@x, @y)` is an error.
- If you need to specify the prototype in the new syntax (very rarely necessary), add the prototype after the new-style parameters, using the syntax `:(PROTO)`.

- 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 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*.

- 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;
```

- 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. Dereferencing is done 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";
```

- 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`. Dereferencing is done 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 this into a program **eg9** and try it out...

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

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

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

```
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 make a reference to an array (**eg10**):

```
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->[$n]`.
- You can dereference the whole array as `@$ref`.

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

```
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->[$n]`.
- You can dereference the whole array as `@$ref`.
- You can also make an anonymous array ref (**eg11**):

```
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 make a reference to an array (**eg10**):

```
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->[$n]`.
- You can dereference the whole array as `@$ref`.
- You can also make an anonymous array ref (**eg11**):

```
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**):

```
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**):

```
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\n";
}
```

- We can declare anonymous hash refs (**eg13**):

```
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\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" }
);
```

- We can declare anonymous hash refs (**eg13**):

```
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\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}`.

- We can declare anonymous hash refs (**eg13**):

```
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\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:

- 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 (**eg14**):

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

- 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 (**eg14**):

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

- Or use Perl module `Data::Dumper` - which is designed to navigate and print reference structures (**eg15**):

```
use Data::Dumper;
... definition of fred ...
print Dumper \@fred;
```

- 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 (**eg14**):

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

- Or use Perl module `Data::Dumper` - which is designed to navigate and print reference structures (**eg15**):

```
use Data::Dumper;
... definition of fred ...
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.

- You can also take a reference to a function (called a *coderef* in Perl) and call it (later) through the reference (just like C's *pointers to functions*). As in **eg16**:

```
sub double ($)                               # or fun double( $n )...
{                                            #    {
  my( $n ) = @_;
  return 2 * $n;                             #       return 2 * $n;
}                                            #    }

my $coderef = \&double;                      # ref to function
my $x = $coderef->(10);                      # calls double with arg 10
print "10 doubled is $x\n";
```

- You can also take a reference to a function (called a *coderef* in Perl) and call it (later) through the reference (just like C's *pointers to functions*). As in **eg16**:

```
sub double ($)                                  # or fun double( $n )...
{                                               #    {
  my( $n ) = @_;
  return 2 * $n;                                #       return 2 * $n;
}                                               #    }

my $coderef = \&double;                         # ref to function
my $x = $coderef->(10);                         # calls double with arg 10
print "10 doubled is $x\n";
```

- Perl also allows us to create anonymous coderefs on the fly, as in **eg17**:

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

- You can also take a reference to a function (called a *coderef* in Perl) and call it (later) through the reference (just like C's *pointers to functions*). As in **eg16**:

```
sub double ($)                               # or fun double( $n )...
{                                            #    {
  my( $n ) = @_;
  return 2 * $n;                             #       return 2 * $n;
}                                            #    }

my $coderef = \&double;                      # ref to function
my $x = $coderef->(10);                      # calls double with arg 10
print "10 doubled is $x\n";
```

- Perl also allows us to create anonymous coderefs on the fly, as in **eg17**:

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

- Using coderefs, you can do lots of cool *functional programming* - higher order functions, callbacks, factories, iterators, lazy evaluation. This is the topic of the new 7th lecture!

- In the last session, there was a multi-part exercise that attempted to build word frequency indexes of files. We can do more now:
  - Use `split` to allow the indexing program to split each line into multiple words and index each word.
  - Convert the indexer into separate functions, nicely laid out. Add `use strict` and prototypes, and `Function::Parameters` if you like.
  - Record when each data file was last indexed. Write a `reindex` program to check the modification time of each indexed document file and reindex modified documents.
  - 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 **eg14** replacing the entire inner `foreach` loop that builds the `@x` array with a `map` invocation that begins `my @x = map ....` If you're feeling brave, you can remove `@x` altogether and make the body of the outer `foreach` a single statement beginning `print join( ", ", map....`