Introduction to Perl: Fifth Lecture

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

Dept of Computing,
Imperial College London

January 2015

- In this session, we'll discuss using add-on modules to make Perl even more powerful.
- We'll cover:
  - **Perl's documentation system**
  - **what a module is**
  - **where we can find many useful modules**
  - **how we use them in our own programs**
  - **a brief detour into Perl objects** and
  - **lots of examples of using some common modules**.

Aside: Uses of Hashes

- But first: we've already said that you can omit the quotes on a *literal hash key string*, this is often used to pretend that a hash is a record/structure, as in:

  ```
  $info{forename} = 'Duncan';
  ```

  where `forename` is a string, pretending to be a field name.
- Hence, hashes can be used to represent *records*, *sets* (hashes where all values are 1), *bags* (frequency hashes), and of course general indexed mappings.

Perl has a tremendous amount of online documentation - reference info, tutorials, cookbooks and FAQs. All is accessed using **perldoc** - start by looking at the overview `perldoc perl`. You can also lookup documentation for:

- Any standard function: `perldoc -f functionname`.
- Any installed module: `perldoc modulename`.
- Standard library overview: `perldoc perlfunc`.
- Standard modules overview: `perldoc perlmodlib`.
- What a module really is: `perldoc perlmod`.
- The Perl FAQ: `perldoc perlfaq`.
- Search the FAQ for a term such as password:
  `perldoc -q password`.

All Perl documentation is written in Perl's own format called **POD: Plain Old Documentation** (see `perldoc perlpod` for details).
Perl allows **POD** documentation to be included directly in your own Perl scripts and modules - so you can't lose the documentation!

- A major motivation of the Perl 5 redesign was to allow additional C libraries to be dynamically loaded into a running Perl interpreter. This feature is used to add major new functionality to Perl without having to recompile the Perl interpreter.
- **Perl modules** can either be *pure Perl* – literally just containing Perl code – or *XS* – containing code in C as well as Perl, to (for instance) provide an interface to an existing C library (like `Tk`).
- In the sixth lecture, we'll cover how to write Perl modules, with all the usual good properties, eg. *data hiding*, *abstraction* and separate *name spaces*.
- For now, we're going to look at how to use modules which have been made available to us by others.
- As well as the Perl standard library of functions (see `perldoc perlfunc` and lecture 4 for details), Perl comes with a *large number of modules* installed by default, which we can use simply by writing `use modulename` in our Perl programs. So far we've met `Function::Parameters` and `Data::Dumper`.

- Beyond Perl's standard modules, there are thousands of high-quality modules written by Perl developers, held in a well organized globally mirrored collection called **CPAN**:

```
http://www.cpan.org/
```

- There are several Perl tools (called `cpan`, `cpanm` etc) which can install most CPAN modules automatically. See `perldoc cpan`.
- It is *definitely* worth looking at CPAN before you start to write significant chunks of code – there may well be a module that already does a large part of what you want to do!

Module Naming Conventions

- The Perl module namespace can be hierarchical. Many module names - like `Data::Dumper` - contain `::` and the first part of their name is usually a general indication of their area of interest.
- So `Data::Dumper` pretty-prints complex data structures, `XML::Simple` gives us a simple interface to XML, `Math::BigInt` allows us to do mathematics with very large integers, etc etc.

Installing and Using Modules

- If you have to download and install a module yourself, you will be pleased to discover that the vast majority of modules have a common installation method:

```
perl Makefile.PL
make
make install
```

- You can specify extra switches on the `perl Makefile.PL` line to either install the module system-wide or for you alone: `perldoc perlmodinstall` for details.
- Once a module has been installed, tell Perl that you want to use the module in your own program: `use Data::Dumper;`

Where are Modules found?

- Perl modules are always stored in files with the extension `.pm`, e.g. `POSIX.pm` – where `pm` stands for "Perl Module".
- A hierarchical module like `Data::Dumper` will be stored in a file called `Dumper.pm` inside a directory called `Data`.

- When you say `use module` where does Perl search for the file `module.pm`?
- Perl has a list of locations that it searches, called the *include path*. The include path is available within a Perl program as the special variable `@INC`.
- `@INC` always includes the *current directory* (so local modules work) and wherever *system-installed modules* (Perl standard and CPAN modules chosen by the sysadmin) have been placed.
- When we're writing programs for other users to use, the directory where you develop the code (the *source directory*) is not the same as where the code is installed for use (the *installation directory*).
- Typically, then, we build a Perl program and some associated modules (CPAN or our own locally written) and then want to:
  - Install the program into (say) `/homes/dcw/bin`.
  - Ensure that `/homes/dcw/bin` is on our path.
  - Install the modules into (say) `/homes/dcw/lib/perl`.
  - And have the program *know where to find the modules*.

- You can add an extra directory (`/homes/dcw/lib/perl` for example) to the include path in two ways:
  - Run your Perl script via:
    ```
    perl -I/homes/dcw/lib/perl ...
    ```
  - Alternatively, near the top of your script, add:
    ```
    use lib qw(/homes/dcw/lib/perl);
    ```
- This works, but it's a real pain to move to another location - because you have to change all references to `/homes/dcw/lib/perl` to (say) `/vol/project/XYZ/lib`.
- This becomes a serious problem as your applications grow larger; imagine editing 10 main programs and 50 support modules.
- Need a *position independent* way of finding the modules. The standard module `FindBin` lets you find the directory where the running Perl script is located, so you can specify the library location *relative to that directory*:

```
use FindBin qw($Bin);
use lib qw($Bin/../lib/perl);
use MyModule;
```

- Here, `MyModule.pm` in `../lib/perl` will be found and used.

- Often, Perl modules just provide a collection of functions for you to use, but many also provide an *object-oriented* view of their functionality.
- Let's briefly discuss how Perl does OO.
- In Perl, a class is a special kind of module, so class names like `IO::File` are common.
- In Perl a constructor can be called *whatever the class designer chooses*! However, the convention is to call the constructor `new`.
- So, assuming we have a class `Student`, if we want to create a new instance of a `Student`, we say:

```
use Student;
my $bob = Student->new();
```

- Perl provides the syntactic sugar, no longer recommended:

```
my $bob = new Student();
```

- Either way, `$bob` is now a `Student` instance.

- Now let's use `$bob` as an object:
- Assuming the class `Student` has an object method called `attend`, taking the name of a lecture and a room, we could say:

```
$bob->attend('Perl Short Course', 'Huxley 311');
```

- Note: the syntax is similar to reference syntax. Wonder why?
- Many methods want optional arguments, and a conventional way of doing this has emerged: pass a single hash literal, with parameter names as keys and parameter values as values. The keys are conventionally chosen starting with '`-`' and written without string quote marks, as in:

```
$object->method( -name => 'hello',
                 -values => [ 'a', 'b', 'c' ] );
```

- This tells us enough about Perl objects to begin discussing modules with OO interfaces.

- **CGI** stands for **Common Gateway Interface**, and specifies how external programs can communicate with a webserver, and hence with a client viewing a web page.
- We know what HTML looks like, producing it in Perl is simple:

```
print "<html>\n";
print " <head><title>Hello World!</title></head>\n";
print " <body><h1>This is a simple web page.</h1>\n";
print "  <h2>Brought to you the hard way.</h2>\n";
print " </body>\n";
print "</html>\n";
```

- All we need to know to get started with CGI scripting is that we must send a *Content-type header* before the content, followed by a blank line. So, to make our Perl script work from the web, we add to the beginning (giving example **eg1**):

```
print "Content-type: text/html\n\n";
```

- Having made **eg1** executable, syntax checked it, let's run it standalone (`./eg1`) and eyeball the output. Now, to install it as a CGI script:

```
cp eg1 ~/public_html/perl2014/eg1.cgi
```

  and then point a web browser at

```
http://www.doc.ic.ac.uk/~dcw/perl2014/eg1.cgi
```

- However, all this literal HTML is horribly unwieldy. Surely there must be a better.. more Perlish.. way?

- Of course there is! Perl has a brilliant `CGI` module to deal with this. **eg2** produces the same effect:

```
use CGI;

my $cgi = CGI->new;
print $cgi->header,
      $cgi->start_html("Hello World!"),
      $cgi->h1("This is a simple web page."),
      $cgi->h2("Brought to you the easy way."),
      $cgi->end_html;
```

- `CGI` contains many more methods, and can produce web forms:

```
use CGI;

my $cgi = CGI->new;
print $cgi->header,
      $cgi->start_html('A trivial form'),
      $cgi->h1('A trivial form'),
      $cgi->start_form,
      'Enter your name:', $cgi->textfield('name'), $cgi->p,
      'Select your level of Perl expertise:',
      $cgi->popup_menu(
         -name => 'expertise',
         -values => ['Newbie', 'Adequate', 'Guru', 'Larry']
      ), $cgi->p,
      $cgi->submit,
      $cgi->defaults('Clear'),
      $cgi->end_form;
print $cgi->end_html;
```

- The CGI module can also deal with processing form responses - the param() method either tells you whether any parameters are available, or extracts a particular parameter’s value.
- So, let’s extend our form to generate a suitably sarcastic response when you fill in the form and submit it (**eg3**):

```
my %response = (
        Newbie => "Get on with it then!",
        Adequate => "One day you too may wear sunglasses",
        Guru => "Pretty cool sunglasses",
        Larry => "We bow before your godlike powers!" );
if( $cgi->param )
{
        # Process form parameters...
        my $name = ucfirst( lc( $cgi->param(’name’) ) );
        my $expertise = $cgi->param(’expertise’);
        my $msg = $response{ $expertise } || "umm?";
        print $cgi->hr, "Hello, $name the $expertise - $msg";
}
```

- Now, after selecting a name (Joe) and an expertise level (Newbie), we get output like:

  Hello, **Joe the Newbie** *- Get on with it then!*

- This is only scratching the surface of what CGI can do - use perldoc CGI to find out more. Also, Perl has several Rails-like *MVC web frameworks*: **Dancer, Catalyst** and **Mojolicious**.

- LWP::Simple is a very useful module - but strangely named - which provides a simple method of fetching web pages. If you’re curious, LWP stands for *libwww-perl*.
- Think of it as a misnamed *Webclient::Simple*.
- Let’s read an arbitrary web page from within a Perl script (**eg4**):

```
use LWP::Simple;

my $url = shift @ARGV || "http://www.doc.ic.ac.uk/~dcw/perl2014/";
my $contents = get($url) || die "oops, no webpage $url\n";
print $contents;
```

- Note how the entire text of the web page is stored in a single Perl scalar. Did we mention that Perl strings can be big?
- This is so simple, we could even do this as a one-liner:

```
perl -MLWP::Simple -e ’$_=get("http://www.doc.ic.ac.uk/~dcw/perl2014/")||die;print’
```

  (The -M flag is a shorthand way of using a module).
- Another powerful function provided by LWP::Simple is:

```
getstore($url, $filename)
```

  which downloads the contents of $url directly to the named $filename, without needing to store it all in memory first.

- One thing we can do with our newly-downloaded web page is to parse the HTML.
- HTML::Parser is a complex beast, read its Perl documentation to understand it fully! Despite the name, it’s really an HTML lexical analyser.
- Let’s link LWP::Simple and HTML::Parser together to find all links - **eg5**:

```
use Function::Parameters;
use LWP::Simple;
use HTML::Parser;
use URI::URL;

my $url;
my @links = ();

#
# deal with a start tag with its attributes
#
fun findlinks( $tag, $attr )
{
        return unless $tag eq "a";
        my $link = $attr->{href};
        return unless defined $link;
        $link = url( $link, $url )->abs;
        push @links, $link;
}
```

- And here’s the main program of **eg5**:

```
# main program
die "Usage: eg5 [url]\n" unless @ARGV < 2;
$url = shift @ARGV || "http://www.doc.ic.ac.uk/~dcw/perl2014/";
my $contents = get( $url ) || die "eg5: can’t fetch URL $url\n";

my $parser = HTML::Parser->new(
          start_h => [ \&findlinks, ’tagname,attr’] );
$parser->parse( $contents );

# now @links contains the links - print them out.
foreach (@links)
{
        print "link: <$_>\n";
}
```

- Now, suppose we want to fetch all linked .ps, .pdf and .tgz files, storing them together in a new directory. Replace the link printout with (giving **eg5a**):

```
mkdir( $destdir, 0755 ) unless -d $destdir;
chdir( $destdir ) || die "can’t cd into $destdir\n";

foreach (@links)
{
        next unless /^http/ && m#/([^/]+\.(ps|pdf|tgz))$#;
        my $filename = $1;
        print "fetching $_ -> $destdir/$filename\n";
        getstore( $_, $filename ) || warn "can’t fetch $_\n";
}
```

- `DBI` is a module which allows Perl to connect to databases and manipulate data within them.
- Databases supported by `DBI` include MySQL, Oracle, Sybase, SQLite, PostgreSQL and Microsoft SQL server – we use the last two here in DoC.
- `DBI` provides a `connect` constructor to connect to a database. A typical example would be:

```
use DBI;

my $db = 'films';
my $host = 'db.doc.ic.ac.uk';
my $port = 5432;
my $user = my $password = 'lab';
my $dbh = DBI->connect(
            "dbi:Pg:dbname=$db;host=$host;port=$port",
            $user, $password
          ) || die "can't connect to $db as $user";
```

- `$dbh` is now a *database handle*, connected to the chosen database - in this case the DoC lab "films" database.
- When we have finished, we need to `disconnect` the handle.

```
$dbh->disconnect;
```

- Once we have a connection to the database, we then need to be able to issue queries over that connection to retrieve data. For each SQL query, we *prepare then execute* the SQL:

```
my $sth = $dbh->prepare("select * from films");
$sth->execute || die "Database error: " . $dbh->errstr;
```

- `$sth` is a *statement handle*, which contains the state of the query. Should an error occur, `$dbh->errstr` gives us the last DBI error in human-readable format.
- Next, we should fetch the records returned by the query. There are several methods in DBI to do this – we'll use `fetchrow_hashref`, which returns the next record (or `undef` when no more) as a hash reference, with field names as keys and field values as values:

```
while( my $record = $sth->fetchrow_hashref )
{
   # do something with $record hashref, eg...
   print "title: $record->{title}\n";
}
```

- Finally, we need to `finish` the statement handle:

```
$sth->finish;
```

- Let's put all this together with an example (**eg6**):

```
use DBI;

my $db   = "films";
my $host = "db.doc.ic.ac.uk";
my $port = 5432;
my $user = 'lab';
my $password = 'lab';

my $dbh = DBI->connect(
            "dbi:Pg:dbname=$db;host=$host;port=$port",
            $user, $password
          ) || die "can't connect to $db as $user";

my $sth = $dbh->prepare("select * from films");
$sth->execute || die "Database error: " . $dbh->errstr;

while( my $record = $sth->fetchrow_hashref )
{
        print "Title:    $record->{title}\n";
        print "Director: $record->{director}\n";
        print "Origin:   $record->{origin}\n";
        print "Made:     $record->{made}\n";
        print "Length:   $record->{length}\n";
        print "-" x 30 . "\n";
}
$sth->finish;

$dbh->disconnect;
```

- Let's run it!
- And then fix the warning:-)

- Why not wrap all this clutter up into a reusable sql query function with a per-record callback function - a coderef - giving **eg6a**:

```
fun sql_foreach( $dbh, $sql, $recordcb )
{
        my $sth = $dbh->prepare( $sql );
        $sth->execute || die "Database error: " . $dbh->errstr;
        while( my $record = $sth->fetchrow_hashref )
        {
                $recordcb->( $record );
        }
        $sth->finish;
}
fun printrecord( $record )
{
        print "Title:    $record->{title}\n";  print "Director: $record->{director}\n";
        print "Origin:   $record->{origin}\n"; print "Made:     $record->{made}\n";
        my $length = $record->{length} // '';  print "Length:   $length\n"; print "-" x 30 . "\n";
}
sql_foreach( $dbh, "select * from films", \&printrecord );
```

- Note that if the per-record work is trivial you can call `sql_foreach()` with an anonymous coderef, as in:

```
my $numrecords = 0;
sql_foreach( $dbh, "select count(*) from films",
     fun ($r) { $numrecords = $r->{count} } );     # or sub { $numrecords = $_[0]->{count} };
```

- If you hate SQL and want an **ORM** like **ActiveRecord** in Rails, Perl has several to choose from. The current favourite is `DBIx::Class`, pronounced **DBIC**. See `perldoc DBIx::Class`.

- In the first session's exercises, we briefly mentioned DBM - a very efficient storage system which can associate an arbitrary string with an arbitrary key with efficient indexed access.
- Back then, we used `dbmopen` and `dbmclose` to access the file using a platform-specific default DBM format. However, a much better way is to use `tie`, since it will let us specify exactly *which* DBM format to use (for there are many)!
- Here's our simple **mksecret** program from the first session, but using `tie` instead to create an SDBM, which is good for small amounts of data (**eg7**):

```
use Fcntl;
use SDBM_File;

tie(my %secret, 'SDBM_File', 'secrets-sdbm',
        O_RDWR|O_CREAT, 0666
    ) || die "oops, couldn't tie SDBM";
$secret{Davros}   = 1;
$secret{Zygon}    = 1;
$secret{Cyberman} = 1;
untie(%secret);
```

- Note that SDBM actually creates two files: `secrets-sdbm.pag` and `secrets-sdbm.dir`.

- Using `tie` more than once allows us to convert between DBM formats easily! Let's convert our secrets file from SDBM to Berkeley DB format, provided by the `DB_File` module (**eg8**):

```
use Fcntl;
use SDBM_File;
use DB_File;

tie(my %secret, 'SDBM_File', 'secrets-sdbm',
        O_RDWR, 0666
    ) || die "oops, couldn't tie SDBM";
tie(my %newsecret, 'DB_File', 'secrets-bdb',
        O_RDWR|O_CREAT, 0666
    ) || die "oops, couldn't tie BDB";

%newsecret = %secret;                   # shazam!

untie(%newsecret);
untie(%secret);
```

- Berkeley DB is a single-file DBM format, and so it really writes a file called `secrets-bdb` (with a `.db` file extension on some platforms).
- `perldoc AnyDBM_File` provides useful information on which DBM format to choose in a given situation. Note that no DBM implementation will scale into trillions of *(key,value)* pairs.

- Many programs take extra options or switches on their command line. For example, many Unix commands understand `--help` to mean "tell the user how to use me".
- We've already discussed command line arguments in `@ARGV`, and we could obviously just use that to detect and process switches. However, someone else has already written a module: `Getopt::Long`.
- `Getopt::Long`'s primary function is `GetOptions`, which looks at `@ARGV` and deals with anything which looks like an option you've told it about, removing them from `@ARGV`.

```
use Getopt::Long;

my $list;
my $format = "DB_File";
my $result = GetOptions('list'     => \$list,
                        'format=s' => \$format);
```

- Here `--list` is merely a flag, whereas `--format` will require a string `(=s)`. Both `--list` and `--format` are optional.
- On the last slide we'll use `Getopt::Long` to provide a multi-format DBM file viewer (**eg9**).

```
use Fcntl;
use SDBM_File;
use DB_File;
use Getopt::Long;

my $format = "DB_File";
my $result = GetOptions('format=s' => \$format);

die "Usage: eg9 [--format=S] filename [word word...]\n"
    unless $result && @ARGV >= 1;

my $filename = shift @ARGV;

tie(my %secret, $format, $filename, O_RDONLY, 0666) ||
    die "can't tie $filename using $format\n";
if( @ARGV == 0 )
{
    foreach (keys %secret)
    {
        print "$_ is a secret\n";
    }
} else
{
    foreach (@ARGV)
    {
        if( exists $secret{$_} )
        {
            print "Yes, $_ is a secret\n";
        } else
        {
            print "No, $_ is not a secret\n";
        }
    }
}
untie(%secret);
```