Perl Short Course: Fifth Session

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

Dept of Computing,
Imperial College London

December 2011

- 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**.

Hashes as Records

- But first, an aside: 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 pretending to be a field name.

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). One of the cute things about **POD** is that Perl understands **POD** documentation being included in your own Perl scripts and modules - so there's no possibility of losing 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`).
- We're going to look at how to use modules which have been made available to us by others.
- When we are writing large programs, we want to structure our code as several Perl modules, with *data hiding*, *abstraction* and separate *name spaces*. This will be covered in the final session.
- As well as the Perl standard library of functions (see **perldoc perlfunc** and session 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 scripts. So far we've met `IO::File` and `Data::Dumper`.

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

  http://www.cpan.org/

- As well as the CPAN website – helpfully mirrored all over the world – there is a Perl module (called `CPAN`) which you can use to install most CPAN modules automatically. See **perldoc CPAN** for more information.
- 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

- Perl modules are always stored in files with the extension `.pm`, e.g. `POSIX.pm` – where `pm` stands for "Perl Module".
- The Perl module space is hierarchical. Module names - like `Data::Dumper` - may 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 for us, `XML::Simple` gives us a simple interface to manipulate XML structured data, `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 `Makefile.PL` line to either install the module system-wide or for you alone. See **perldoc perlmodinstall** for details.
- Once a module has been installed, you need to tell Perl that you want to use the module in your own program - do this with the keyword `use`, as in: `use Data::Dumper;`

- Often, Perl modules just provide a collection of subroutines for you to use, but many also provide an *object-oriented* view of their functionality.
- Some of the modules we're about to use are OO-based, and so we need to 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 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();
```

- For greater familiarity, Perl provides the syntactic sugar:

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

- Either way, `$bob` is now an instance of class `Student`.

- Now we have `$bob`, we can now use it 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 that the syntax is very similar to reference syntax. Behind the scenes, objects are implemented as references - usually hashrefs - associated (blessed) with a Perl package.
- Many member functions 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:

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

  This gives us example **eg1**.
- To install this as a CGI script:

```
cp eg1 ~/public_html/perl2011/eg1.cgi
```

  and then point a web browser at

```
http://www.doc.ic.ac.uk/~dcw/perl2011/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 that provides functions to deal with all of this nastiness. The following example **eg2** produces the same effect (albeit with somewhat more verbose HTML and fewer linebreaks!):

```
use CGI;

my $cgi = new CGI;
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 = new CGI;
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 } or "umm?";
        print $cgi->hr, "Hello, $name the $expertise - $msg";
}
```

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

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

- Use **perldoc CGI** to find out more. You'll also come across CGI again later this year.

- `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*.
- Take a simple example: perhaps we want to be able to read some random web page from within a Perl script (**eg4**):

```
use LWP::Simple;
my $url = shift @ARGV  ||
        "http://www.doc.ic.ac.uk/~dcw/perl2011/";
my $wp = get($url) || die "oops, no webpage $url\n";
print $wp;
```

- 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?
- Another powerful function provided by this module is

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

  which downloads the contents of `$url` directly to the named `$filename`.

- 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 (via **perldoc HTML::Parser**) to understand it fully!
- So, let's link `LWP::Simple` and `HTML::Parser` together to do something useful! (**eg5**):

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

my $url;
my @links = ();

#
# deal with a start tag with its attributes
#
sub findlinks ($$)
{
        my( $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/perl2011/";
my $webpage = get( $url ) || die "eg5: can't fetch URL $url\n";

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

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

- Now, suppose we want to fetch all linked `.ps` or `.tgz` files, storing them together in a new directory. Replace the link printout with:

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

foreach (@links)
{
        next unless m#([^/]+\.(ps|tgz))$#;
        my $filename = $1;
        print "fetching $_ -> $destdir/$filename\n";
        getstore( $_, $filename ) || next;
}
```


- `DBI` is a module which allows Perl to connect to databases and manipulate data within them.
- Databases supported by `DBI` include MySQL, Oracle, Sybase, Microsoft SQL server, and PostgreSQL – we use the last two here in DoC.
- `DBI` provides a *class method* `connect` 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 - 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.
- The first stage in querying via DBI is to `prepare` it. This entails specifying precisely what query we are going to run. We do this by calling an object method on the database handle:

```
my $sth = $dbh->prepare("select * from films");
```

- This returns a *statement handle*, which will contain the state of the query. We now need to `execute` this query:

```
$sth->execute || die "Database error: " . $dbh->errstr;
```

- `DBI` provides the last error to us in human-readable format via the database handle method `errstr`. Using the *do or die* idiom with this is a good idea!
- If we are running a `select` query, then we need to fetch the records returned by the query (see next slide). Otherwise, we just need to `finish` the statement handle:

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

- There are several methods in DBI to do this – probably the most useful are the statement handle methods `fetchrow_hashref` and `fetchrow_array`.
- `fetchrow_hashref` returns the next row resulting from the query as a reference to a hash, with field names as keys and field values as values.
- `fetchrow_array` returns the next row resulting from the query as an array containing the table values in the order in which the query requested the fields.
- Both of these return `undef` when all the records are exhausted, so we commonly use them in a `while()` loop:

```
while( my $record = $sth->fetchrow_hashref )
{
   # do something
}
```

- Having fetched all the records, we should `finish` the query, as stated earlier.
- 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:-)

I recommend wrapping all this clutter up into a reusable sql query function with a per-record callback function:

```
sub sql_foreach ($$$)
{
        my( $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;
}

sub printrecord ($)
{
        my( $record ) = @_;
        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";
}

.....
sql_foreach( $dbh, "select * from films", \&printrecord );
```

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

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


- 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{Romulan} = 1;
$secret{Klingon} = 1;
$secret{Vulcan} = 1;
untie(%secret);
```

- Note that SDBM actually creates two files: `secrets-sdbm.pag` and `secrets-sdbm.dir`. You still access the SDBM from Perl by calling it `secrets-sdbm`, though.

- 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).
- If in doubt which DBM format to use, **perldoc AnyDBM_File** provides useful information on which to choose in a given situation.

- 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 `@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 next slide we'll use `Getopt::Long` in anger, to provide a command-line interface for our DBM file converter (**eg9**). As usual, consult **perldoc Getopt::Long**.


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

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

die "Usage: eg11 [--format=S] filename [secrets]\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);
```