Survey of Advanced Perl Topics

Database access
"OpSys"-like functions
Signal Handling
Inside-Out Objects

Database Access

- Standardized through the DBI module
  - not core module, but installed on CS system
- Differing Database systems have their own DB driver module
  - DBD::mysql, DBD::Oracle, DBD::Informix, etc
- By simply changing which DBD::* you use, your existing code can work with a new database system.
- Setting up an individual DB on CSNet is difficult, due to lacking root access. If you really need one, contact labstaff.

General Format of Perl DBI

- declare a "DataBase Handler" to connect to the Database
- define SQL statement(s)
- prepare the SQL, returning "STatement Handler(s)"
- execute the statement handlers, passing values for the bind parameters
- fetch the results from the statement handlers, if appropriate

DBI Example

```
use DBI;
my $dsn =
'DBI:mysql:database=game;host=db.example.com';
my $dbh = DBI->connect($dsn,$user,$password);
```

```
my $sql = "SELECT * FROM players ";
$sql   .= "WHERE score > ? AND score < ?";
```

```
my $sth = $dbh->prepare($sql);
$sth->execute($minscore, $maxscore);
```

```
while (my $r = $sth->fetchrow_hashref){
  print "$r->{name}'s score: $r->{score}\n";
}
```

- You can use the same $sth to now execute the prepared SQL with different values

Fetching Methods

- **`fetchrow_hashref`**
  - fetch "next" row, as a hash reference (key = column name, value = field value)
- **`fetchrow_arrayref`**
  - fetch next row, as an array reference (in order defined by the table)
- **`fetchrow_array`**
  - fetch next row as an array
- **`fetchall_arrayref`**
  - fetch entire results
    - no parameters - entire table as arrayref of arrayrefs
    - pass array ref - select numbers of which columns of entire table to return
    - pass hashref - select names of which columns to return, and return as an array of hash refernece s
    - pass empty hashref - return entire table as array of hash references
- **`fetchall_hashref($key)`**
  - fetch entire results, as a hashref of hashrefs. Key is index of table

DBI errata

- **`do()`** - combines **`prepare()`** and **`execute()`**.
  - if you don't need to repeatedly execute the statement
  - Frequently used for DELETE, UPDATE, and INSERT statements
    - can't be used for SELECT, because there's no $sth to fetch from
  - **`$dbh->do('DELETE FROM class WHERE drop = 1');`**
- **`$sth->rows()`** returns the number of rows inserted, updated, or deleted.
  - DOES NOT return number of rows that are selected!
- SQL NULL ==> Perl **`undef`**
- **`$dbh->{RaiseError} = 1`**, Perl will die on any SQL error.
  - (Otherwise, must check return value of every db call, and then check **`$DBI::err`**)
- http://dbi.perl.org & **`perldoc DBI`**

"OpSys-like" functions

- touched on these in the "external commands" presentation that we didn't cover in lecture.
  - basically said "Don't do that".
- please, take extreme caution when using these functions
- listing of Perl equivalents only
- for more information about the internals of Unix, take OpSys

fork()

- split off a separate process, duplicating the code and environment
- return value in parent is child's new pid
- return value in child is 0
-
```
my $pid = fork();
if ($pid) { #parent
  print "Child $pid just forked off\n";
  do_parent_stuff();
} else {
  print "I'm the child, just spawned\n";
  do_child_stuff();
}
```

`wait()`

- If there are any running children processes, waits for one of them to finish, and returns the PID of that child
  - **`$?`** is set to the exit status of the child that was just found with wait
- If there are no remaining running children, immediately returns -1
- **`waitpid($pid, 0)`**
  - wait for a specific child to exit

exec($cmd, @args)

- Execute **`$cmd`**, passing **`@args`** to that command
- executes IN THE CURRENT PROCESS, wiping out anything else this code was going to do
- therefore, has no return value
- any code below the **`exec`** (other than a warning that the exec failed) is meaningless.
- **`$retval = system($cmd, @args);`** is equivalent to:
-
```
if (my $pid = fork()){ #duplicate process
   waitpid($pid);#parent wait for child to exit
   $retval = $?; #exit status of child
} else {
   exec ($cmd, @args); #child executes $cmd
}
```

Signal Sending

- Processes can send signals to one another.
- Most common use is to tell a process to die
- Therefore, function to do this is **`kill`**
- **`kill($signal, $pid);`**
  - **`$signal`** is a number or signal name
    - **`6`**, **`'SIGABRT'`**, or **`'ABRT'`**
- to see which signals are available, run **`` `kill -l` ``** on your system
- By default, signals result in program termination
- Most shells respond to a CTRL-C by sending the current process a **`SIGINT`**

Signal Handling

- With the exception of SIGKILL (9) and SIGSTOP (23), all signals can be caught and processed.
- If your program receives a certain signal, you can decide what to do about it.
- Assign a reference to the handler subroutine to the %SIG hash, where the key is the 'name' of the signal
  - signal name also passed as first argument to the subroutine.
-
```
$SIG{INT} = sub {
  print "Bwaha, your CTRL-C doesn't scare me!";
};
```
- Now, if the user tries to CTRL-C your program, the message will be printed out instead of the program dieing.
  - (To actually kill this script, find out its pid from `ps -u <rcsid>`, and then send it a SIGKILL: `kill -9 <pid>`)

Signal Yourself

- There is one signal that's a little more special.
- **`ALRM`** - used to set an alarm clock for yourself.
- **`alarm($seconds)`** - $seconds from now, send yourself (ie, the current process) a **`SIGALRM`** signal
- Before calling **`alarm()`**, set a signal handler in **`$SIG{ALRM}`** for what you want to do when this signal is received.
- Only one alarm can be set at once
  - additional calls cancel the previous alarm
    - return how many seconds remaining until alarm would have gone off
  - **`alarm(0)`** to cancel alarm without setting new.

Advanced Perl Objects

- As we've seen, Perl's notion of objects were kludged into the code
- Perl's OO is lacking keywords and definitions of a "real" Object Oriented system
- Many attempts have been made to remedy this
- The latest and greatest: **`Moose`**
- Full blown Object Oriented layer on top of Perl
  - Inheritance, Inspection, Interfaces, Polymorphism, Hard typing, Triggers, etc

Moose

- Several new keywords for you to define your class
- **`package Point;`**
- **`use Moose;`**
-
```
has x => (
   isa      => 'Int',
   is       => 'rw',
   required => 1,
);
```
-
```
has y => (
   isa      => 'Int',
   is       => 'rw',
   required => 1,
);
```

Attribute Options

- **`is`** is one of "rw" or "ro"
  - rw => "Read/Write". The corresponding method both returns the attribute's value, and allows it to be set.
  - ro => "Read Only". The corresponding method only returns the attribute's value.
- **`isa`** is the type of data the attribute contains
  - Str, Num, Int, ArrayRef, Object, <Class>, etc
- **`required => 1`** if the user must pass the arg into the constructor
- **`default`** sets initial value of arg not passed in
  - Can be a non-ref scalar, or a subroutine reference

Moose Methods

```
sub show {
   my $self = shift;
   my ($x,$y) = ($self->x(), $self->y());
   print "($x, $y)\n";
}
sub reset {
   my $self = shift;
   $self->x(0);
   $self->y(0);
}
```

- x() and y() methods were created by the attribute definitions

Moose Subtyping

- You can create your own types for attributes.

```
use Moose::Util::TypeConstraints;
...
subtype 'octnum'
   => as 'Str'
   => where { /^0[0-7]+$/ }
   => message {"$_[0] not a valid octal
num"};
has permissions => (
    is => 'rw',
    isa => 'octnum'
    default => '0775'
);
```

Moose Triggers

- You can setup methods to be called before or after any other method is called.

```
sub withdraw {
  my $self = shift;
  $self->{bal} -= shift;
}
sub deposit {
  my $self = shift;
  $self->{bal} += shift;
}
after qw/withdraw deposit/ => \&show_bal;
sub show_bal {
  print "balance: $_->{bal}\n";
}
```

Moose Subclassing

- To create a subclass, use the 'extends' keyword:

```
package Person;
use Moose;
#...
1;
package Student;
use Moose;
extends 'Person';
has id_num => ( is => 'ro', isa => 'Num',
required => 1);
1;
```

- Student retains all the attributes, methods, triggers, etc of Person.

Moose location & documentation

- Moose is a CPAN module. Not installed on CSNet.
- use lib "/cs/lallip/lib/perl5/site_perl/5.10.0/";
- Read "Moose::Intro" and "Moose::Manual" on the CPAN
  - or
    export PERL5LIB=/cs/lallip/lib/perl5/site_perl/5.10.0
    perldoc Moose::Intro
    perldoc Moose::Manual

Recommended Further Reading

- *Advanced Perl Programming*, Simon Cozens
  - Introspection, parsing beyond Regexps, Templating, Databases, Unicode, Event Driven Programming, Embedding C in Perl
- *Higher Order Perl*, Mark Jason Dominus
  - Recursion & Callbacks, Dispatch Tables, Iterator-based programs, Parsing & Infinite Streams, "Higher-Order" functions, Program transformations
- *Perl Best Practices*, Damian Conway
  - Tips and Advice for writing clean, elegant, maintainable Perl programs