Perl Programming

Part 2

Andrew Arensburger

http://www.cfar.umd.edu/~arensb/text/perl/

Packages

- `package` *name* sets the default namespace.
- $*package*`::`*var* explicitly specifies the package.

```
package English;
$hello = "Hello";        # Sets $English::hello

package French;
$hello = "Bonjour";      # Sets $French::hello

package main;
print $hello;            # Wrong: $main::hello
                         # does not exist
print $French::hello;

package English;
print $hello;            # Prints $English::hello
```

Packages (cont.)

- You can use `package` anywhere.

```
if ($language eq "english") {
        package English;
        print $hello;   # Print $English::hello
}                       # End of English scope
elsif ($language eq "french") {
        package French;
        print $hello;   # Print $French::hello
}                       # End of French scope
```

- Packages can be nested.
- This does not imply any relation between the outer and inner packages (but others may think it does).

```
package Lang;
$hello = "Huh?";

package Lang::English;
$hello = "Hello";

package Lang::French;
$hello = "Bonjour";
print $hello;   # Prints $Lang::French::hello
```

BEGIN and END

- Packages may contain BEGIN and END blocks.
- BEGIN blocks are executed during compilation, as soon as they are seen.
- END blocks executed in reverse order of declaration.
- BEGIN and END blocks are executed even if you're only checking syntax!

```
package Moo;

print "Munch munch munch...\n";

BEGIN {
    print "Welcome to Moo.\n";
}

package Editor;
BEGIN { open THEFILE, "/some/file"; }
END   { close THEFILE; }

# Initialize lots of stuff.
$big_thing = &initialize();
END { &flush_changes($big_thing, THEFILE); }
```

AUTOLOAD

- `AUTOLOAD` will be called if Perl can't find a function.
- Called with the arguments to the original function.
- `$AUTOLOAD` set to the name of the function.
- `AUTOLOAD` function is responsible for handling errors.

```
package Adder;

sub AUTOLOAD {
    if ($AUTOLOAD !~ /add(\d+)$/)
    {
        die("No such function: $AUTOLOAD");
    }

    my $toadd  = $1;
    my $number = shift;

    return $number + $toadd;
}

package main;

$x = 1;
print &Adder::add1($x);
```

UMIACS

Modules

- `require "`*filename*`"` loads a file.
- `require` *filename* loads *filename*`.pm`.
- `@INC` specifies the search path.
- `%INC` lists loaded packages, to prevent multiple inclusion.
- A module should return a true value. Usually just put

  ```
  1;
  ```

  at the end.
- Nested packages are translated into directories:

  ```
  require Lang::French::Canadian;
  ```

  loads `Lang/French/Canadian.pm`.

Modules (cont.)

- `use` *package [list]*
- `use MyPackage 1, 2, 3;` is (almost) equivalent to

```
BEGIN {
    require MyModule;
    MyModule::import("MyModule", 1, 2, 3);
}
```

- `use` is evaluated at compile-time.
- `import` function provides a hook.

```
package Directories;

sub import {
    my $package = shift;

    # Add all arguments to the list of
    # include directories
    unshift @INC, @_;
}

1;
```

- To avoid calling `import`, use

```
use MyModule();
```

- `no` *package*: similar to `use`, but calls `unimport` (rarely used).

Symbolic References

Consider:

```
sub nroff { ... };
sub troff { ... };

if ($using_nroff) {
    &nroff("-ms", $filename);
} else {
    &troff("-ms", $filename);
}
```

- In a `sh` script, you'd just say

  ```
  $TYPESETTER -ms $filename
  ```

- Use

  $*expr*
  ${*expr*}

  ```
  $var = 3;
  $ref = "var";
  $$ref = 4;                   # Sets $var
  ${$ref} = 5;                 # Ditto
  ${$ref."two"} = 6;           # Sets $vartwo
  @$ref = ('a','b','c');       # Sets @var
  @refs = ("foo", "bar", "baz");
  ${$refs[0]} = "howdy";       # Sets $foo
  ${$refs[0]}[0] = "quux";     # Sets $foo[0]
  ```

Symbolic References (cont.)

The earlier example can be rewritten as

```
sub nroff { ... };
sub troff { ... };
sub groff { ... };

   if ($using_nroff) { $typesetter = "nroff" }
elsif ($using_troff) { $typesetter = "troff" }
elsif ($using_groff) { $typesetter = "groff" }

&$typesetter("-ms", $filename);
```

- Symbolic references don't always work:

```
package Passwd;

sub expand_gcos {
    my $username = shift;
    my $gcosref  = shift;

    $$gcosref =~ s/&/$username/;
                # Modifies $Passwd::name, not
                # $main::name
}

package main;

$name = "andrew &urger";
&Passwd::expand_gcos("arensb", "name");
```

- Turn them off with `use strict 'refs'`.

Hard References

- Hard references work like pointers in C, but safer.
- They can only point to existing things.
- No pointer arithmetic.
- Everything is a thingy.
- Use *var* to get the address of *var*.

```
$ref    = \$myvar;
$$ref   = 3;              # Sets $myvar
${$ref} = 4;              # Ditto

@array        = (1, 2, 3);
$arrayyref    = \@array;
shift @{$arrayyref};      # shift @array;
$elementref   = \$array[1];
$$elementref = "two";     # @array is now
                          # (1, "two", 3)

sub hi { print "hello" }
$funcref = \&hi;
&$funcref;                # Calls &hi;
```

`ref`

- Can't use a scalar reference as an array, etc.
- `ref`(*reference*) returns the type of what *reference* points to, or empty string:

  `REF`
  `SCALAR`
  `ARRAY`
  `HASH`
  `CODE`
  `GLOB`
  Other.
- If interpolated into a string, `ref` expands to a unique string.

Anonymous References

Anonymous array:

```
$arrayref = [ 'a', 'b', 'c' ];
```

Anonymous hash:

```
$hashref = { Jan => 31,
             Feb => 29,
             Mar => 31,
             Apr => 30,
           }
```

Anonymous function:

```
$funcref = sub { print "Hello"; }
```

- A reference is a scalar.
- Hence, you can put references in arrays and hashes.
- Hence, you can build complex data structures:

```
@array2D = (            # Array of arrays
        [ 0, 1, 2 ],
        [ 'a', 'b', 'c', 'd', 'e' ],
        [ 31, [28, 29], 31, 30, 31, 30,
          31, 31, 30, 31, 30, 31
        ],
);
```

An Ugly Example

```
$complex =
        [ [ "Unix",
            "operating system",
            { gurus => [ "Brian",
                         "Dennis",
                         "Ken"
                       ]
            }
          ],
          [ "Perl",
            "language",
            { author => "Larry",
              gurus  => [ "Larry",
                          "Tom",
                          "Randal"
                        ]
            }
          ]
        ];
```

How do you get at "Randal"?

An Ugly Example (cont.)

Begin with

```
$complex
```

Dereference this:

```
@{$complex}
```

and subscript the result:

```
${$complex}[1]
```

Dereference and subscript again:

```
${${$complex}[1]}[2]
```

Dereference again:

```
%{${${$complex}[1]}[2]}
```

and subscript:

```
${${${$complex}[1]}[2]}{"gurus"}
```

Dereference and subscript again:

```
${${${${$complex}[1]}[2]}{"gurus"}}[2]
```

An Ugly Example (cont.)

- -> syntactic sugar:

```
$ $arrayref  [0]
${$arrayref} [0]
  $arrayref->[0]

$ $hashref  {string}
${$hashref} {string}
  $hashref->{string}
```

- This only works for scalars.

Simplifying the earlier expression:

```
${${${${$complex} [1]} [2]} {"gurus"}} [2]
${${${  $complex->[1]} [2]} {"gurus"}} [2]
${${    $complex->[1]->[2]} {"gurus"}} [2]
${      $complex->[1]->[2]->{"gurus"}} [2]
        $complex->[1]->[2]->{"gurus"}->[2]
```

- The -> can be omitted between successive subscripts:

```
$complex->[1][2]{"gurus"}[2]
```

- Use the explicit form for debugging.
- Use the clearer form once it works.

Closures

```
package Numbers;

$counter = 0;   # Gratuitous counter

sub makefunc {
    my $divisor = shift;

    # Build a function on the fly, and return it
    return sub {
        my $number = shift(@_);

        $counter++;

        # Return true if $number is
        # evenly divisible by $divisor
        return ($number % $divisor == 0);
    };
}

package main;

$is_even   = &Numbers::makefunc(2);
$isdivby10 = &Numbers::makefunc(10);

foreach $n (1, 2, 3, 4, 5)
{
    print "$n is ",
        &$is_even($n) ? "even" : "odd",
        "\n";
}
```

Closures (cont.)

- A closure knows what context it was defined in.
- Anonymous functions are closures.
- `&is_even` is defined in `Numbers` but called in `main`.
- Which variables, precisely, is it using?
- `$number` is `my` in `&$is_even`; it is private.
- `$counter` is `$Numbers::counter`. Just look at the code.
- `$divisor` is `my` in `makefunc`. `&$is_even` has a private copy.
- `$divisor`'s initial value is whatever it was when the anonymous function was created.
- Private copies are independent.

Historical Interlude

In the beginning:

- `JMP` begat `GOTO`.
- No dynamic memory allocation.
- No complex arrays.

The 1960's:

- Structured Programming.
- Dijkstra, *Goto Statement Considered Harmful*.
- Code should be divided into atomic blocks.
- One entrance, one exit.
- Not so fervent about that, nowadays.

Code can be grouped, why not data?

- Encapsulation: `struct`s.
- API functions: advertised means of access.

Inheritance:

- *isa* relationship.
- Polymorphism (often problematic).

Encapsulation + Inheritance = Objects

Structured Programing Example 1

```
sub count_elements {
    my @list = @_;              # List of elements
    my $i;
    my $num_elements;           # Return value
    my $seen_end = 0;           # Have we reached the end
                                # of the list?

    if (@list == ()) {
        $num_elements = 0;
    } else {
        # @list is not empty
        for ($i = 0; ($i <= $#list) && !$seen_end; $i++) {
            if (&is_invalid($list[$i])) {
                # Do nothing
            } else {
                # Check for end of list
                if (&is_end_of_list($list[$i])) {
                    $seen_end = 1;
                } else {
                    # Current element is valid
                    $num_elements++;
                }
            }
            $i++;
        }
    }
    return $num_elements;
}
```

Structured Programing Example 2

Without the one-exit rule:

```
sub count_elements {
    my @list = @_;           # List of elements
    my $i;
    my $num_elements = 0;    # Number of elements in list

    return 0 if @list == ();# Trivial case

    for ($i = 0; $i <= $#list; $i++) {
        next if &is_invalid($list[$i]);
                             # Ignore invalid elements
        last if &is_end_of_list($list[$i]);
                             # Nothing useful after this
        $num_elements++;
    }
    return $num_elements;
}
```

Encapsulation:

```
    /* Sales information */
    int     serial_no[1024];
    double  price[1024];
    char *  customer[1024];
```

vs.

```
    struct {
        int     serial_no[1024];
        double  price[1024];
    } sales[1024];
    char *  customer[1024];
```

Objects

- Each thingy can be labeled with a package name.
- When this happens, it's called an object.
- Use `bless` *ref [,package]*

```
package Sneeze;

$var = [ "achoo" ];
bless $var, Sneeze;
```

- A class is just a package that happens to define methods.
- A method is just a function.
- Class methods take a class name as their first argument.
- Instance methods take an object reference as their first argument.
- A single function can be a plain function, a class method, and an instance method.

Class methods

```
package User;

# Create a new User
sub new {
    my $class = shift;          # Class name
    my $newthing = {};          # The new object

    bless $newthing, $class;    # Bless the object
    return $newthing;
}

# Look up a User by username
sub lookup {
    my $class = shift;          # Class name
    my $name  = shift;          # Name to look for
    my $newuser  = {};          # The new object

    # Look up the User
    my ($uname, $passwd, $uid, $gid) =
        getpwnam($name);
    $newuser->{"uname"} = $uname;
    $newuser->{"uid"}   = $uid;
    $newuser->{"gid"}   = $gid;

    bless $newuser, $class;     # Bless the object
    return $newuser;
}

package main;

$user1  = new User;
$arensb = lookup User "arensb";
$bin    = lookup User ("bin");

$user2  = User->new;
$root   = User->lookup("root");
```

Instance Methods

```
package User;

sub get_fullname {
    my $self = shift;            # The object
    my $fullname;                # Return value

    $fullname = $self->{"gcos"};    # Get the GCOS field
    $fullname =~ s/&/\u$self->{uname}/;
                                 # Replace & with username
    return $fullname;
}

sub add_to_groups {
    my $self   = shift;          # The object
    my @groups = shift;          # Groups to add user to

    push @{$self->{"groups"}}, @group;
}

package main;

$user = new User;
get_fullname $user;
add_to_group $user "operator", "bin";

$arensb = User->lookup("arensb");
$arensb->get_fullname;
$arensb->add_to_group("wheel");
```

Inheritance

- Inheritance allows packages to inherit other packages' functions.
- `@ISA` lists parent packages.
- `@ISA` traversed depth-first.
- If a function isn't found, Perl looks for `AUTOLOAD` in `@ISA`
- If it still can't find it, looks in `UNIVERSAL`.

```
package Child;
@ISA = qw( Mother Father );

package main;
$billy = new Child;
$billy->say("Hello");
Father::say("Da-da!");          # Don't use this
$billy->Father::say("Da-da!");  # Use this
```

Inheritance (cont.)

- Use the `SUPER` pseudo-class to pass the buck:

```
package FilledPolygon;
@ISA = qw( Polygon );

sub draw {
    my $self = shift;

    # Look for parent's draw()
    $self->SUPER::draw;
    &fillme;
}
```

- Use the two-argument form of `bless`.
- Don't assume you know what you're blessing into:

```
package Parent;

sub new {
    my $newobject = {};

    # Bless into current package (wrong!)
    bless $newobject;
    return $newobject;
}

package Child;
@ISA = qw( Parent );

package main;

$thing = new Child;
# $thing is a Parent, not a Child
```

Objects (cont.)

Destructors:

- Destructors are named `DESTROY`.
- Destructors are not chained automatically.
- But they're rarely used.

Privacy:

- Perl isn't fascist.
- There is no built-in privacy enforcement.
- Use comments and documentation (or lack thereof).

Odds and Ends

`eval`

`eval` *string*
`eval` *block*

- Parses and runs its argument.
- Catches `die` (check `$@`).
- Use the block form if you can.

Typeglobs

- Not terribly useful these days.
- `*var` refers to *all* variables named `var`.
- To make everything named `foo` an alias for `bar`:

```
*foo = *bar;
```

- To make `$foo` an alias for `$bar`:

```
*foo = \$bar;
```

- Also useful for creating references to filehandles:

```
open(THEFILE, ">/my/file");
$fh = \*THEFILE;
print $fh "Hello, world!\n";
```

Tied Variables

- Allows you to control precisely how a variable works.
- Database files are usually accessed through tied hashes.
- All other uses are suspect.

Scalars

`TIESCALAR` *class*,*list*

`FETCH` *this*

`STORE` *this*, *value*

`DESTROY` *this*

Arrays

`TIEARRAY` *class*,*list*

`FETCH` *this*, *index*

`STORE` *this*, *index*, *value*

`DESTROY` *this*

Hashes

`TIEHASH` *class*, *list*

`FETCH` *this*, *key*

`STORE` *this*, *key*, *value*

`DELETE` *this*, *key*

`CLEAR` *this*

`EXISTS` *this*, *key*

`FIRSTKEY` *this*

`NEXTKEY` *this*, *lastkey*

`DESTROY` *this*

Style

- Check for odd conditions early, so you can make simplifying assumptions.
- Use `next` and `last` to break out of loops early.
- `return` early.
- Put functions at the end; make the main body easy to find.
- Put customization variables at the top.
- Choose wisely between prefix and postfix conditionals.
- Choose wisely between `if` and `unless`:

  ```
  next unless defined($key);
  ```

  vs

  ```
  next if !defined($key);
  ```
- You almost certainly want `my`, not `local`.

Further Reading

- Man page roadmap. See perl(1).
- Command-line arguments. See perlrun(1).
- PODs. See perlpod(1).
- Tricks and techniques. See perlbot(1), but mainly other people's code.
- Embedding and extending. Start with perlembed(1).

```
srand(@}=split $},"Pe thoe\nrslrtcaeu knrJa h");
print  splice  @},  rand @}  =>0<= 1  while  @};
```