1

A Quick Tour of Perl

Ed Cashin

UGA ACM Presentation
November 4, 1999

toc

1

ecashin@coe.uga.edu

toc

3

Contents

1 hello 4
1.1 hello.pl 4
1.2 hello-nowarn.pl 5
1.3 hello-warn.pl 6
2 types 7
2.1 scalar.pl 7
2.2 array.pl 9
2.3 hash.pl 10
3 subroutines 11
3.1 hello-sub.pl 11
4 references 13
4.1 multiarray.pl 13
4.2 hasharray.pl 15
5 Packages 17
5.1 packages.pl 17
6 OOP 19
6.1 a Document 19

toc

3

4

6.2 a Letter 25
7 modules 29
7.1 CGI 29
8 More 33
Colophon 35

toc

4

1 hello

The "hello world" program is straightforward in perl, except that the shebang line gives away the fact that this is a script. Although perl is a "scripting language", there is a distinct compilation phase. In some situations (e.g. mod perl) the lines begin to blur.

1.1 hello.pl

```
1  #! /usr/bin/perl

2  print "Hello, world!\n";
```

toc

6

hello-nowarn.pl

1.2 hello-nowarn.pl

```
1 #! /usr/bin/perl

2 print "Hello, world!\n";

3 print "Hello, $modifier world!\n";
```

toc

6

7

hello-warn.pl

1.3 hello-warn.pl

```
1  #! /usr/bin/perl -w

2  print "Hello, world!\n";

3  print "Hello, $modifier world!\n";
```

toc

7

2

types

There are only three primary types in perl, but these are enough to build quite complex data structures.

2.1 scalar.pl

The scalar data type holds any value that is a singularity. Strings are thought of as "one thing" in perl. Treating strings as arrays of characters is not something you do in perl.

```
1 #! /usr/bin/perl -w

2 $modifier       = "big";

3 print "Hello, $modifier world!\n";

4 $modifier       = 1;
```

toc

9


```
5  print "Hello, $modifier world!\n";
```

toc

9

2.2 array.pl

```
1 #! /usr/bin/perl -w
2 @list           = ("cruel", "fantastic", "homely");
3 $modifier       = join ", ", @list;
4 print "Hello, $modifier world!\n";
```

toc

2.3 hash.pl

```
1  #! /usr/bin/perl -w

2  %world_by_wealth        = (
3                             'rich'              => "unfulfilling",
4                             'middle-income'     => "tolerable",
5                             'poor'              => "hard",
6                             );

7  foreach (keys %world_by_wealth) {
8      $modifier   = $world_by_wealth{$_};

9      printf("%20s people say: \"Hello, %s world!\"\n",
10           $_, $modifier);
11 }
```

toc

3 subroutines

3.1 hello-sub.pl

The subroutine in perl is like the function in C. To take advantage of lexical scoping like one finds in C, we can use *my variables*.[1]

The compiler pragma, `use strict`, makes sure that the compiler is working for the programmer, finding even more bugs before runtime.

```
1  #! /usr/bin/perl -w

2  use strict;

3  while (<>) {
```

[1] When a "thingy" in memory can't be accessed from the perl code, it is garbage collected. Lexically-scoped variables provide an easy way to manage memory, since their contents are freed when their scope ends.

toc

13


```
4       &process_line($_);
5   }

6   sub process_line {
7       my $line    = shift;

8       print $line if $line =~ /barney/;
9   }
```

toc

13

4 references

4.1 multiarray.pl

Perl does not use raw pointers like one finds in C; rather perl uses references to point to objects in memory. Like pointers, references facilitate the construction of complex data types.

Perl's syntax for creating references allows for effortless construction of complex data structures with very high readability.

```
1 #! /usr/bin/perl -w

2 use strict;

3 &show(&multi_array);

4 sub multi_array {
```

toc

```
 5      my @twoD    = ([ "Hans", "Hagen", "ConTeXt", "documents" ],
 6                     [ "Don", "Knuth", "TeX", "documents" ],
 7                     [ "Elizabeth", "Frasier", "voice" ]);
 8      return \@twoD;
 9  }

10  sub show {
11      my $ref     = shift;
12      print "\$ref scalar holds value: $ref\n";
13      print "\$ref->array holds values: "
14        . (join " ", @$ref) . "\n";
15      print "\$ref->array->[0] holds values: "
16        . (join " ", @{ $ref->[0] }) . "\n";

17      print "\n------------all values----\n";
18      foreach my $rowR (@$ref) {
19         print join " ", @$rowR;
20         print "\n";
21      }
22  }
```

toc

4.2 hasharray.pl

```
1  #! /usr/bin/perl -w

2  use strict;

3  my @students    = (
4                      {
5                          'fname'          => "Fred",
6                          'lname'          => "Dupperon",
7                          'GPA'            => "4.0",
8                          'polite'         => "no",
9                      },
10                     {
11                         'fname'          => "Lefty",
12                         'lname'          => "Fredrickson",
13                         'GPA'            => "3.2",
14                         'polite'         => "no",
15                     },
```

toc

```
16                      {
17                          'fname'          => "Rita",
18                          'lname'          => "Marley",
19                          'GPA'            => "3.0",
20                          'polite'         => "yes",
21                      }
22                      );

23  foreach my $student (@students) {
24      if ($student->{'polite'} eq "no") {
25         print $student->{'fname'} . " received a B.\n";
26      } else {
27         print $student->{'fname'} . " received an A+ !!\n";
28      }
29  }
```

toc

18

5 Packages

5.1 packages.pl

Packages act as *namespaces*, and are symbol tables implemented as hashes.

```
1  #! /usr/bin/perl -w

2  package Cow;
3  use strict;
4  use vars qw($malename $femalename);

5  $malename       = "bull";
6  $femalename     = "cow";

7  package Horse;
8  use strict;
```

toc

18

19


```
 9  use vars qw($malename $femalename);

10  $malename       = "stallion";
11  $femalename     = "mare";

12  package main;
13  print "a cow male is a: $Cow::malename\n";
14  print "a horse male is a: $Horse::malename\n";
```

toc

19

6 OOP

6.1 a Document

```
1  # Document.pm

2  package Document;

3  use vars qw($DOC_DIR);

4  $DOC_DIR        = "/var/tmp";

5  sub new {
6      my $pkg     = shift
7        or die "Error: constructor requires OO syntax";
8      my $self    = {
9         'name'   => "",
```

toc

```
10          'text'   => "",
11      };
12      return bless $self, $pkg;
13  }

14  sub text {
15      die "Error: parameter missing" if @_ < 1;
16      my ($self, $newval)         = @_;
17      if (defined $newval) {
18          $self->{'text'}  = $newval;
19      }
20      return $self->{'text'};
21  }

22  sub typeset {
23      die "Error: parameter missing" if @_ < 1;
24      my $self    = shift;
25      my $name    = $self->{'name'} || "texdoc";
26      my $texfile         = "$DOC_DIR/$name.tex";
```

toc

```
27      open TEXFILE, "> $texfile"
28        or die "Error: could not open $texfile for writing: $!";
29      print TEXFILE $self->tex;
30      close TEXFILE;

31      my $redir   = " > $name.out 2>&1 < /dev/null";
32      my $error   = system "(cd $DOC_DIR && pdftex $name $redir)";
33      if ($error) {
34         die "Error: pdftex returned exit status of: "
35           . ($error / 256);
36      }

37      return $self->slirp_file("$DOC_DIR/$name.pdf");
38  }

39  sub tex {
40      die "Error: parameter missing" if @_ < 1;
41      my ($self)  = @_;
42      my $tex     = "\% Produced by a Document\n";
43      $tex        .= $self->{'text'};
```

toc

```
44      $tex        .= "\\bye";

45      return $tex;
46  }

47  sub slirp_file {
48      die "Error: parameter missing" if @_ < 2;
49      my ($self, $filename)       = @_;
50      my $contents;
51      {
52         local $/;
53         undef $/;
54         open IN, $filename
55           or die "Error: could not open $filename for reading";
56         $contents        = <IN>;
57         close IN;
58      }
59      return $contents;
60  }
```

toc

ui.pl

```
 1  #! /usr/bin/perl -w

 2  use strict;
 3  use Document;

 4  my $doc                 = new Document();
 5  my $user_text   = "";
 6  while (<>) {
 7      $user_text  .= $_;
 8  }

 9  $doc->text($user_text);

10  my $pdf         = $doc->typeset;

11  open OUT, "> userdoc.pdf"
12    or die "Error: could not open userdoc.pdf for writing";
```

toc

25


```
13  print OUT $pdf;
14  close OUT;
```

toc

25

25

6.2 a Letter

A *Letter* is a *Document*. This is inheritance. Perl's OO features have all their guts available for the programmer to see.

```
1  # Letter.pm

2  package Letter;
3  use Document;

4  @ISA    = qw(Document);

5  sub new {
6      my $pkg     = shift
7        or die "Error: constructor requires OO syntax";
8      my $self    = new Document;

9      # recipient, sender

10     return bless $self, $pkg;
```

toc

```
11  }

12  sub recipient {
13      die "Error: parameter missing" if @_ < 1;
14      my ($self, $newval)         = @_;
15      if (defined $newval) {
16         $self->{'recipient'}     = $newval;
17      }
18      return $self->{'recipient'};
19  }

20  sub sender {
21      die "Error: parameter missing" if @_ < 1;
22      my ($self, $newval)         = @_;
23      if (defined $newval) {
24         $self->{'sender'}        = $newval;
25      }
26      return $self->{'sender'};
27  }
```

toc

```
28  sub tex {
29      die "Error: parameter missing" if @_ < 1;
30      my ($self)  = @_;
31      my $vspace
32        = '\null\vskip 1cm plus .5cm minus .75cm' . "\n";
33      my $header  = <<'END_OF_HEADER';
34  % Produced by a Letter
35  \font\bigfont=cmssbx10 at 16 pt
36  \font\twelverm=cmr12

37  % impress people with a fancy summation
38  \hfil {\bigfont FANCY SUMMATIONS, INC.}
39  \hfil $$\sum_{n=0}^\infty2y+4x^2$$\hfill

40  \nopagenumbers
41  \advance\baselineskip by 1ex
42  \twelverm

43  END_OF_HEADER
```

toc

```
44      my $tex     = $header . $vspace;
45      $tex        .= "{\\obeylines\n" . `date` . $self->sender;
46      $tex        .= "\n$vspace\n" . $self->recipient
47        . "}\n$vspace\n";
48      $tex        .= $self->{'text'};
49      $tex        .= "\\bye";

50      return $tex;
51  }
```

toc

7 modules

7.1 CGI

Here we use the Letter class, that we just developed, in conjunction with a popular perl module, CGI.

Perl modules are distributed in a very convenient distributed archive called CPAN, the Comprehensive Perl Archive Network. It even has a search engine.

CPAN means that for perl, code reuse is not a promise but an everyday reality.

```
1  #! /usr/bin/perl -w

2  use strict;
3  use CGI;
4  use CGI::Carp;
5  use Letter;
```

toc

31


```
 6  use vars qw($TMP_DIR);
 7  $TMP_DIR        = "/var/tmp";

 8  &do_something(new CGI());

 9  sub do_something {
10      die "Error: missing parameter" if @_ < 1;
11      my $cgih    = shift;

12      if ($cgih->param()) {
13         &print_feedback_page($cgih);
14      } else {
15         &print_welcome_page($cgih);
16      }
17  }

18  sub print_welcome_page {
19      die "Error: missing parameter" if @_ < 1;
20      my $cgih    = shift;
```

toc

31

```
21      print $cgih->header;
22      print $cgih->start_html(-title => "webletter -- Welcome");

23      # -- stick the name of the document onto the "action"
24      # -- attribute of the form HTML tag
25      my $action
26        = $cgih->url(-relative    => "yes") . "/product.pdf";
27      print $cgih->startform(-action => $action);

28      foreach my $part ("Sender", "Recipient", "Text") {
29         print "<p>$part:<br>\n";
30         print $cgih->textarea(-name      => lc($part),
31                               -wrap      => "hard",
32                               -rows      => 5);
33         print "\n";
34      }
35      print $cgih->submit(-name   => "action",
36                          -value   => " make me a form ") . "\n";
37      print $cgih->endform() . "\n";
38      print $cgih->end_html;
```

toc

```
39  }

40  sub print_feedback_page {
41      die "Error: missing parameter" if @_ < 1;
42      my $cgih    = shift;
43      my $letter  = new Letter();
44      $letter->sender($cgih->param("sender") or die "no sender");
45      $letter->recipient($cgih->param("recipient")
46                         or die "no recipient");
47      $letter->text($cgih->param("text") or die "no text");
48      my $pdf     = $letter->typeset;

49      print $cgih->header(-type   => "application/pdf");
50      print $pdf;
51  }
```

toc

34

8 More

- **DBI**

  Interacting with relational databases from perl—this one and CGI are the two most popular modules.

- **Regular Expressions**
  - Regular expressions have been integrated with the language syntax
  - Extended but familiar syntax
  - Easy to use parenthetical matches
  - Possible to format complex regular expressions with whitespace
- **GUI Programming**
  - Perl/Tk allows the use of the Tk library, best known from TCL/Tk, for building GUI applications. It works on Windows.
  - Perl/GTK allows perl programmers to use the GTK (Gimp Toolkit) GUI widgets from perl code. I haven't tried it, but it sounds good.

toc

34

35

- **Resources**
  - "The Camel Book" *Programming Perl, 2nd Edition*, Larry Wall, Tom Christiansen, and Randal L. Schwartz. O'Reilley

    This is the book for programmers looking to get a handle on the whole language quickly, written by Wall himself and other perl gurus.
  - "The Lama Book" *Learning Perl, 2nd Edition*, Randal L. Schwartz and Tom Christiansen. O'Reilley

    This popular book is more of a friendly introduction to perl.
  - *Advanced Perl Programming*, Sriram Srinivasan.

    Srinivasan investigates perl further, illuminating areas that make perl a language for building serious applications.
  - `http://www.perl.com/`
  - `"man perl"` Perl has pretty extensive online documentation in familiar formats generated from its own *POD* format.

toc

35

36

Colophon

This document was produced using the ConTEXt document preparation system, a macropackage based on the famous TEX typesetting program by Donald Knuth, that offers a coherent and powerful interface while providing modern features.

see `http://www.pragma-ade.nl/`

toc

36

37

toc

37

37