Some Advanced Perl

UVic SEng 265

Daniel M. German
Department of Computer Science
University of Victoria

September 30, 2002 Version: 1.00

dmgerman@uvic.ca

References

- By design hash and array elements should be scalars
- To support complex data structures we need to use references (which are scalars)
- Perl then supports complex data structures by using references
- Like arrays of arrays, or assoc. arrays of arrays, etc.

dmgerman@uvic.ca

Making References: Rule 1

- If you put a \ in front of a variable, you get a reference to that variable.

```
$aref = \@array;          # $aref now holds a reference to @array
$href = \%hash;           # $href now holds a reference to %hash
```

- Once the reference is stored in a variable like $aref or $href, you can copy it or store it just the same as any other scalar value:

```
$xy = $aref;              # $xy now holds a reference to @array
$p[3] = $href;            # $p[3] now holds a reference to %hash
$z = $p[3];               # $z now holds a reference to %hash
```

Making References: Rule 2

- `[ ITEMS ]` makes a new, anonymous array, and returns a reference to that array.
- `{ ITEMS }` makes a new, anonymous hash. and returns a reference to that hash.

```
$aref = [ 1, "foo", undef, 13 ];
# $aref now holds a reference to an array

$href = { APR => 4, AUG => 8 };
# $href now holds a reference to a hash
```

- The references you get from rule 2 are the same kind of references that you get from rule 1:

```
# This:
$aref = [ 1, 2, 3 ];
# Does the same as this:
@array = (1, 2, 3);
$aref = \@array;
```

Using References: Rule 1

- If `$aref` contains a reference to an array, then you can put `$aref` anywhere you would normally put the name of an array. For example, `@{$aref}` instead of `@array`.
- For arrays:

```
@a              @{$aref}              An array
reverse @a      reverse @{$aref}      Reverse the array
$a[3]           ${$aref}[3]           An element of the array
$a[3] = 17;     ${$aref}[3] = 17      Assigning an element
```

- For references:

```
%h              %{$href}              A hash
keys %h         keys %{$href}         Get the keys from the hash
$h{'red'}       ${$href}{'red'}       An element of the hash
$h{'red'} = 17  ${$href}{'red'} = 17  Assigning an element
```

Using References: Rule 2

- `${$aref}[3]` is too hard to read, so you can write `$aref->[3]` instead.

```
@a                  @{$aref}                  An array
$a[3]               $aref->[3]               An element of the array
$a[3] = 17;         $aref->[3] = 17          Assigning an element
```

- `${$href}{red}` is too hard to read, so you can write `$href->{red}` instead.

```
%h                  %{$href}                  A hash
$h{'red'}           $href->{'red'}           An element of the hash
$h{'red'} = 17      $href->{'red'} = 17      Assigning an element
```


dmgerman@uvic.ca

Passing Variables by Reference to Functions

- An example of how to do it:

```
#!/usr/bin/perl
@a = (1, 2, 3);
print "Original values: @a\n";
Modify_Array_Value(@a);       # passed by value
print "After modify value @a\n";
Modify_Array_Ref(\@a);        # passed by reference
print "After modify ref @a\n";
sub Modify_Array_Value
{
    local (@array) = @_;
    $array[0] = 99;
    print "Inside modify array value @array\n";
    return;
}
sub Modify_Array_Ref
{
    local ($arrayRef) = @_;
    $arrayRef->[0] = 77;
    print "Inside modify array reference @array\n";
    return;
}
####################Output#############
Original values: 1 2 3
Inside modify array value 99 2 3
After modify value 1 2 3
Inside modify array reference
After modify ref 77 2 3
```


dmgerman@uvic.ca

Return values from Functions

- Perl has simple semantics: you either return a scalar or a list
- Example:

```
#!/usr/bin/perl

@a = My_Function_Array();
$b = My_Function_Scalar();
print "@a\n";
print "$b\n";

sub My_Function_Array
{
    return (1, 2, 3);
}
sub My_Function_Scalar
{
    return 99;
}
```

- Output:

```
1 2 3
99
```


dmgerman@uvic.ca

References Allow Complex Data structures

- Multidimensional Arrays
- Records
- Arrays of hashes and vice-versa
- Read the man page `perldata`

dmgerman@uvic.ca

Using Parameters

- The array @ARGV holds all the parameters to the program
- Example:

```
#!/usr/bin/perl

foreach (@ARGV) {
    print "$_\n";
}
```

- Run:

```
#Run as:
./parms.pl 1 2
#Output:
1
2

#Run as:
./parms.pl --x --y --z "abcde xyz" --prefix='^abc'
#Output:
--x
--y
--z
abcde xyz
--prefix=^abc
```


dmgerman@uvic.ca

Try the following

- Create a program called test.pl

```
#!/usr/bin/perl

$argument = shift @ARGV;

print "Argument: $argument\n";

while (<>) {
    print;
}
```

- Run it as:

```
./test.pl --option test.pl
```


dmgerman@uvic.ca

Scope

- Variable declaration comes into play when you need to limit the scope of a variable's use. You can do this in two ways:
  - **Dynamic scoping: `local`** creates a variable that is lexically scoped, i.e. it is visible to functions called from within the block in which they are declared.
  - **Lexical scoping: `my`** creates private constructs that are only visible within their scopes. They are totally hidden from the outside world.

Scope

- Example:

```
#!/usr/bin/perl
First();

sub First
{
    my $b = 4;
    local $a = 3;
    print "Inside First: a->[$a] b->[$b]\n";
    Second();
}

sub Second
{
    print "Inside Second: a->[$a] b->[$b]\n";
}
```

- Output:

```
Inside First: a->[3] b->[4]
Inside Second: a->[3] b->[]
```


dmgerman@uvic.ca

undef and defined

- **undef**: A variable holds the undefined value `undef` until it has been assigned a defined value, which is anything other than `undef`.
  - When used as a number, `undef` is treated as 0;
  - when used as a string, it is treated as the empty string, `""`;
  - and when used as a reference that isn't being assigned to, it is treated as an error.
- Sometimes we need to know if a variable has a *value*
- `defined EXPR`

  returns a Boolean value telling whether EXPR has a value other than the undefined value `undef`

dmgerman@uvic.ca

strict

- A pragma that restricts **unsafe** constructs
- add the line `use strict;` to your program:

```
#!/usr/bin/perl
use strict;

$a = 4;
b($a);
```

- When running it, generates the following error:

```
Undefined subroutine &main::b called at extrict.pl line 5.
```

- Or:

```
#!/usr/bin/perl
use strict;

$a = $c;
print "$a\n";


Global symbol "$c" requires explicit package name at extrict2.pl line 4.
Execution of extrict2.pl aborted due to compilation errors.
```

For more information

- OO Perl
- Perl Modules
- Perl FAQ
- Complex data structures
- Anything about perl
- do **man perl**

dmgerman@uvic.ca