Object Orientation Basics

Lesson 1

References and Data Structures

References

- References are Pointers to (the content of) "things" (scalars, arrays, hashes, subroutines, ...)
- To create a reference from a "thing" it is prefixed with the \ character
- All References fit into scalar variables

```
$a=1; @b=(2,3); %c=(a=>1, b=>2, c=>3);
sub hello { print "Hello\n" }
$scalarref = \$a;  $refref = \$scalarref;
$arrayref = \@b;
$hashref = \%c;
$coderef = \&hello;
```

Dereferencing - Prefix notation

- Dereferencing with the proper data type prefix **($ @ % &)**

```
$a1 = $$scalarref;$a2 = $$$refref;
@b1 = @$arrayref; $b2 = $$arrayref[1];
%c1 = %$hashref;  $c2 = $$hashref{a};
print "a=$a1 b[1]=$b2 c{a}=$c2\n";
&$coderef; # call subroutine hello
```

- precedence rules can be modified by {} **$scalar=${$ref}[0]; @array=@{$refs[0]};**
  - formally this means dereferencing a block

Dereferencing with ->

- Dereferencing using the prefix notation hard to read

```
${$aref}[0] = ${$href}{key0};
```

- Array or Infix operator prevents humps of $@{$%

```
$b3 = $arrayref->[1];
$c3 = $hashref->{a};
```

- Especially useful for subroutine references:

```
$coderef->($arg1, $arg2);
```

instead of `&{$coderef}($arg1, $arg2);`

```
$coderef->(); # call hello without args
```

Dereferencing

- The arrow notation cannot be used to dereference whole arrays or hashes
- The arrow operator can be stacked, i.e

```
$b = $refref->{a}->{b};
```

means that refref is a reference to a hash, the element a of that hash contains again a reference to another hash, whose element b gets assigned to the variable `$b`. The name of the second hash is not required at all

The type of a reference

- The function **`ref`** determines the reference type

  `$type = ref $ref_var;`

- the following return values are possible
  - **`undef`** if the argument **`$ref_var`** is not a reference
  - the strings **`SCALAR, ARRAY, HASH, CODE`** if **`$ref_var`** points to the appropriate thing
  - the string **`REF`** if the reference points to a reference
  - the name of an object class, if **`$ref_var`** is an object (see later)

The type of a reference (2)

- The reference type can be printed using **`ref`**
- Interpolating a reference in a double quoted string yields both the reference type and the address

```
print ref $_, "\n" for ($scalarref,
  $arrayref, $hashref, $coderef, $refref);
print "\$hashref in a string becomes
  $hashref\n";
```

the output of the last statement reads

`$hashref in a string becomes` `HASH(0x8784040)`

Reference counting

- For all data in perl a reference count is kept
- Data are kept as long as there is at least one reference to that data left
  - For string `$a="str"; $b=\$a;` the refcount is 2
  - Even if `$a` goes out of scope `$b` remains intact
  - For reference counting there is no difference between ordinary variables and references
- This concept is used to protect unintended access to data (see later)

Anonymous Arrays

- An array can be filled with a list: `@array=(1,2,3);`
- A reference to that array: `$ptr=@array;`
- Array name not required for access: `$a=$ptr->[1];`
- Arrays without names can be defined by using references only: **anonymous arrays**

```
$a_array=[1, 2, 3]; #$a_array is a reference
@n_array=(1, 2, 3); #a normal array
```

- Access to the elements by dereferencing

`$a = $a_array->[1];` but `$a = $n_array[1];`

Multidimensional Arrays

- In perl arrays (or lists) in a list get flatted out
  - `((1,2,3),(4,5,6))` equivalent to `(1,2,3,4,5,6)`
- Can be avoided using references for inner lists
  - anonymous array yields a reference, therefore
  - `@matrix1 = ( [1,2,3], [4,5,6] , [7,8,9] );`
- Access to elements
  - `$m23 = $matrix1[1]->[2]` or
  - `$m23 = $matrix1[1][2]`

Multidimensional Arrays (2)

- More elegant notation using anonymous arrays
  - `$matrix2 = [ [1,2,3], [4,5,6], [7,8,9] ];`
- Access to elements using array notation
  - `$m23 = $matrix2->[1]->[2];` or
  - `$m23 = $matrix2->[1][2]; #matrix notation`
- Differences to previous notation
  - `@matrix1` is an array, `$matrix2` is a reference
  - Outer parentheses `()` versus outer brackets `[]`
  - Array operator `->` required for `$matrix2`

Anonymous Hashes

- Using references to access hash elements the name of the hash is not required (as for arrays)
- Therefore in analogy to anonymous arrays there is syntax for defining **anonymous hashes**
- An anonymous hash is a reference to a hash

```
$a_hash = {num=>1, str=>'a'};
%n_hash = (num=>1, str=>'a'); #normal hash
```

- Access to the elements by dereferencing

`$a = $a_hash->{num};` but `$a = $n_hash{num};`

Data Structures

- Multidimensional hashes can be built analogous to multidimensional arrays
- Arrays and hashes keep only a flat list of items
  - structuring happens by storing pointers instead of data
  - to be useful pointers have to point to different data
  - but: named arrays and hashes have a fixed address
  - each anonymous thing gets a new location
- data structures usually built from anonymous things
- see also **`perldoc perllol`** and **`perldoc perldsc`**

List of lists

Data structures cookbook

Data structure design

- Data structures can hold a variety of data
  - Analogy to records in C
- Data structure usually identified by a reference
  - occupies only a scalar value
  - can contain data (strings, numbers) and
  - references to scalars, arrays, hashes, code (and more)
  - references to references (other data structures)
- For readability anonymous hash best suited
  - keys describe what gets stored, values contain data (refs)

A sample Data Structure

```
$name='Friebel'; @names=('W.');
$record = {familyname=>$name,                    #scalar
           firstnames=>\@names,                 #arrayref
           accounts  =>[$name, 'guest'],        #arrayref
           access    =>{linux=>'accept',        #hashref
                        irix =>'deny'
                       },
           room      =>'1R06',                  #constant
            };
print
"$record->{accounts}->[0]:$record->{access}->{irix}\n";
```

final comma within data structure o.k.

Anonymous Subroutines

- Reference to a piece of code is possible
- Code can be executed knowing only the reference
- This is the idea of **anonymous subroutines**
- Definition of an anonymous subroutine

```
$sub_ref = sub {block};
```

- Call by

```
$sub_ref->(args);
```

- Other subroutine topics (prototypes) not covered

Parameter passing in Subroutines

- Passing by reference in array `@_`
- If `@_` gets modified, so will the original in the caller
- Avoided by making copies of the arguments

```
my ($arg1, $arg2) = @_;# see lesson 2
```

- Passing Arrays as references, otherwise flattening

```
matrix(@row1, @row2); #all Elements in @_
matrix(\@row1,\@row2);#ref to @row1 in $_[0]
```

- Call with `&name; @_` gets passed from caller to subroutine body (implicit parameter passing)

Passing a Data Structure

- A subroutine call can be written in two ways

```
mysub($arg1, $arg2); or mysub $arg1, $arg2;
```

- Passing an anonymous array or hash as the only parameter looks like using wrong parentheses

```
mysub[$arg1, $arg2]; # passing anon array
mysub{$arg1, $arg2}; # passing anon hash
```

- Anonymous hashes can be used to name args in parameter lists and to pass it in arbitrary order

```
mysub{arg2=>'val2', arg1='val1'};
```

A Subroutine Example

```
# Definition of an anonymous subroutine
$sub_ref1 = sub { print "Hello $_[0]\n" };
# Definition of a normal subroutine
sub printit { print "Parameter1: $_[0]\n" };

$sub_ref1->('World'); #call with one parameter

# Parameter passing via @_ with &name form!
$sub_ref2 = sub { &printit };
$sub_ref2->('passed');
```

Closures

- Access to scoped local variables by subroutine

```
{ my $count=0; #access only to end of block
  sub incr { $count++; print "Count is now $count\n"}
  sub decr { $count--; print "Count is now $count\n"}
}
incr;incr;decr; print "No access to count: $count\n";
```

- After end of block still access to variable **`$count`** using the subroutines **`incr`** and **`decr`**
- Variable remains, as reference count != 0
- no chance of manipulating **`$count`**

Suggestions for Practicing

- Try to get familiar with references and especially with the array notation
- Try to write a subroutine that acts differently corresponding to the type of the argument passed
- Try to build a data structure and put the contents of named and unnamed arrays into it. What happens?
- Rewrite scripts using references, anonymous arrays and hashes
- Read the man pages perlref, perllol and perldsc