Perl Library Functions

Perl has literally hundreds of functions for all kinds of purposes:

- file manipulation, database access, network programming, etc. etc.

It has an especially rich collection of functions for strings.

E.g. lc, uc, length.

Consult on-line Perl manuals, reference books, example programs for further information.

Defining Functions

Perl functions (or subroutines) are defined via **sub**, e.g.

```
sub sayHello {
    print "Hello!\n";
}
```

And used by calling, with or without **\&**, e.g.

```
&sayHello;  # arg list optional
sayHello(); # better: show empty arg list explicitly
```

Defining Functions

Function arguments are passed via a list variable `@\_`, e.g.

```
sub mySub {
    @args = @_;
    print "I got ",@#args+1," args\n";
    print "They are (@args)\n";
}
```

Note that @args is a global variable.

To make it local, precede by `my`, e.g.

```
    my @args = @_;
```

Defining Functions

Can achieve similar effect to the C function

```
int f(int x, int y, int z) {
    int  result;
    ...
    return result;
}
```

by using array assignment in Perl

```
sub f {
    my ($x, $y, $z) = @_;
    my $result;
    ...
    return $result;
}
```

Defining Functions

Lists (arrays and hashes) with any scalar arguments to produce a single argument list.

This in effect means you can only pass a single array or hash to a Perl function and it must be the last argument.

```
sub good {
    my ($x, $y, @list) = @_;
```

This will not work (x and y will be undefined):

```
sub bad {
    my (@list, $x, $y) = @_;
```

And this will not work (list2 will be undefined):

```
sub bad {
    my (@list1, @list2) = @_;
```

References

- are like C pointers {(refer to some other objects)}

Parameter Passing

Scalar variables are aliased to the corresponding element of @_. Allows a function to change them, this code sets x to 42.

```
sub assign {
    $_[0] = $_[1];
}
assign($x, 42);
```

Arrays & hashes are passed by value.

If a function needs to change an array/hash pass a reference.

Also use references if you need to pass multiple hashes or arrays.

```
%h = (jas=>100,eric=>95,andrew=>50);
@x = (1..10)

mySub(3, \%h, \@x);
mysub(2, \%h, [1,2,3,4,5]);
mysub(5, {a=>1,b=>2}, [1,2,3]);
```

Perl Prototypes

- Prototypes declare the expected parameter structure for a function.
- In other languages, main purpose of prototypes is type checking.
- The main purpose of prototypes is to allow more convenient calling of functions.
- Prototypes allow users to define functions that are called like builtins.
- Prototypes also provide some error checking - sometimes useful, sometimes less so.
- Some programmers recommend against using prototypes.
- Use in COMP(2041|9044) optional.

Perl Prototypes

Prototypes can cause a reference to be passed when an array is given as a parameter. If we define our version of push like this:

```
sub mypush {
        my ($array_ref,@elements) = @_;
        if (@elements) {
                @$array_ref = (@$array_ref, @elements);
        } else {
                @$array_ref = (@$array_ref, $_);
        }
```

It has to be called like this:

```
mypush(\@array, $x);
```

But if we add this prototype:

```
sub mypush2(\@@)
```

It can be called just like the builtin push:

Recursive example

```
sub fac {
    my ($n) = @_;

    return 1 if $n < 1;

    return $n * fac($n - 1);
}
```

which behaves as

```
    print fac(3);   # displays 6
    print fac(4);   # displays 24
    print fac(10);  # displays 3628800
    print fac(20);  # displays 2.43290200817664e+18
```

The Perl builtin function eval evaluates (executes) a supplied string as Perl.

For example, this Perl will print 43:

```
$perl = '$answer = 6 * 7;';
```