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 with &
sayHello(); # more common, 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  res;
    ...
    return res;
}
```

by using array assignment in Perl

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

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

References

- are like C pointers (refer to some other objects)
- can be assigned to scalar variables
- are dereferenced by evaluating the variable

Example:

```
$aref = [1,2,3,4];
print @$aref;     # displays whole array
... $$aref[0];    # access the first element
... ${$aref}[1]; # access the second element
... $aref->[2];  # access the third element
```

Parameter Passing

Scalar variable are aliased to the corresponding element of `@_`. This means a function can changed them. E.g. 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,arun=>95,eric=>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]);
```

Notation:

- `[1,2,3]` gives a reference to `(1,2,3)`
- `{a=>1,b=>2}` gives a reference to `(a=>1,b=>2)`

Perl Prototypes

Perl prototypes declare the expected parameter structure for a function.
Unlike other language (e.g. C) Perl prototype's main purpose is not type checking.
Perl prototypes' main purpose is to allow more convenient calling of functions.
You also get some error checking - sometimes useful, sometimes less so.
Some programmers recommend against using prototypes.
Use in COMP2041/9041 optional.
You can use prototypes to define functions that can be called like builtins.

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:

```
mypush @array, $x;
```

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
```

Eval

The Perl builtin function eval evaluates (executes) a supplied string as Perl.
For example, this Perl will print 43:

```
$perl = '$answer = 6 * 7;';
eval $perl;
print "$answer\n";
```

and this Perl will print 55:

```
@numbers = 1..10;
$perl = join("+", @numbers);
print eval $perl, "\n";
```

and this Perl also prints 55:

```
$perl = '$sum=0; $i=1; while ($i <= 10) {$sum+=$i++}';
eval $perl;
print "$sum\n";
```

and this Perl could do anything:

```
$input_line = <STDIN>;
eval $input_line;
```

Pragmas and Modules

Perl provides a way of controlling some aspects of the interpreter's behaviour (through *pragmas*) and is an extensible language through the use of compiled modules, of which there is a large number. Both are introduced by the *use* keyword.

- `use English;` - allow names for built-in vars, e.g., `$NF` = `$.` and `$ARG` = `$_`.
- `use integer;` - truncate all arithmetic operations to integer, effective to the end of the enclosing block.
- `use strict 'vars';` - insist on all variables declared using `my`.

Standard modules

These are several thousand stndard Perl modules into packages.
The package name is prefixed with `::`
Examples:

- `use DB_File;` - functions for maintaining an external hash
- `use Getopt::Std;` - functions for processing command-line flags
- `use File::Find;` - find function like the shell's find
- `use Math::BigInt;` - unlimited-precision arithmetic
- `use CGI;` - see next week's lecture.

CPAN

Comprehensive Perl Archive Network (CPAN) is an archive of 150,000+ Perl packages.
Hundreds of mirrors, including
http://mirror.cse.unsw.edu.au/pub/CPAN/
Command line tools to quickly install packages from CPAN.

CPAN

Comprehensive Perl Archive Network (CPAN) is an archive of 150,000+ Perl packages.
Hundreds of mirrors, including
http://mirror.cse.unsw.edu.au/pub/CPAN/
Command line tools to quickly install packages from CPAN.