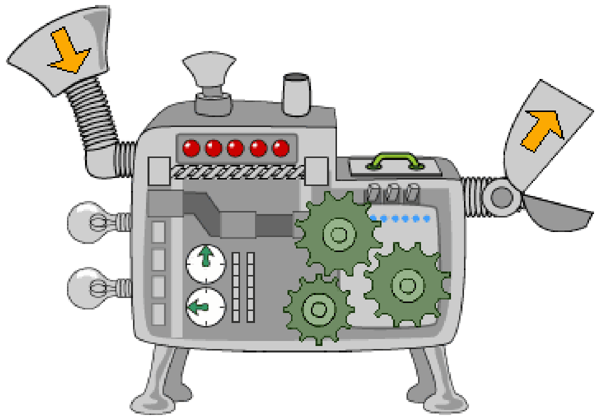
COMP 2021

Unix and Script Programming



Perl Subroutines

Defining & Calling a Subroutine

- General form of a user-defined subroutine is defined in Perl

```
                    sub subroutine_name{
                              body of the subroutine
          }
```

- Typical way of calling that Perl subroutine

```
          subroutine_name(list of arguments);
```

- HelloWorld Example

```
          # Function definition
          sub Hello{
                    print "Hello World!\n";
          }
          # Function call
          Hello();
```

Global Variable in Subroutine

- Subroutine definitions can be anywhere in your program text (they are skipped on execution)but it is most common to put them at the beginning/end of the file.
- Within the subroutine body, you may use any variable from the main program.

```
$cat sub_hello.pl
sub Hello{
 print "Hello $user!\n";
}

$user = "Cindy‟;
Hello();
```

Passing Arguments to a Subroutine

- Various arguments can be passed to subroutine
- The arguments are assigned to a list in a special variable `@_` for the duration of the subroutine (`$_[0], $_[1]` and so on)

```
$ cat sub_max1.pl
#!/usr/local/bin/perl5 -w
sub max{
        if($_[0] > $_[1]){ return $_[0]; }
        else{ return $_[1]; }
}

print "Enter 1st number:\n";
chomp($a = <STDIN>);
print "Enter 2nd number:\n";
chomp($b = <STDIN>);
$max = max($a, $b);
print "max: $max\n";
```

What happens if call `max(3, 4, 5)`?

Passing Arguments to a Subroutine (Cont.)

- A more general way without limitation on the number of arguments:

```
$cat sub_avg.pl
#!/usr/local/bin/perl5 -w
sub Average{
        # total number of arguments passed
        $n = scalar(@_);
        $sum = 0;
        foreach $item (@_){
                $sum += $item;
}
$average = $sum / $n;
print "Average for the given numbers : $average\n";
}

Average(10, 20, 30);
```

More on `@_`

- Don't confuse `$_` and `@_`, they are unrelated.
- Excess parameters are ignored if you don't use them.
- Insufficient parameters simply return `undef` if you look beyond the end of the `@_` array.
- `@_` is local to the subroutine
  - A subroutine can pass argument to another subroutine
  - The nested subroutine invocation gets it own `@_`
  - Works for recursion too!!

Passing Lists to Subroutine

- You can pass a list to subroutine
- A subroutine call *flattens* array contents to a long list

```
$cat sub_listarg.pl
#!/usr/local/bin/perl5 -w
sub double{
foreach my $i (@_){
        $i *= 2;
}
}
my @arr = (1, 2, 3, 4, 5);
double(@arr);
foreach my $i (@arr){
print "$i ";
}
print "\n";
$ sub_listarg.pl
2 4 6 8 10
```

Returning Value

- You can return a value from a function, and use it in any expression

```
$cat sub_avg.pl
#!/usr/local/bin/perl5 -w
sub Average{
        $n = scalar(@_);
        $sum = 0;
        foreach $item (@_){
                $sum += $item;}
        $average = $sum / $n;
        return $average;
}

$num = Average(10, 20, 30);
print "Average for the given numbers : $num\n";
```

Returning Value (cont.)

- A subroutine can also return a list of values

```
return ($a,$b);
```

- You can choose among values to return

```
sub max_of_a_and_b{
          if($a > $b){ return $a; }
          else{ return $b; }
    }
```

A recursion Example

```
#!/usr/local/bin/perl5 -w
sub fact {
     my $val = $_[0];
     if ($val > 1) {
        return $val * fact($val-1);
     } else {
        return 1;
     }
}

$value = $ARGV[0];
$fv = fact($value);
print "factorial $value is $fv\n";
```

Private Variables in Subroutine

- You can create private variables (also called *lexical variables*) with the `my` operator.

```
sub somefunc {

    # $variable is invisible outside
    my $variable;

    # declare many variables at once
    my ($another, @an_array);

    # initialize local variable
    my ($i, $j) = (0, 0);
}
```

use strict

- You can force all variables to require declaration with `my` by starting your program with `use strict;`

```
$ cat sub_max_v2.pl
#!/usr/local/bin/perl5 -w
use strict;
sub max{
 my ($n1, $n2) = @_;
 if ($n1 > $n2) {return $n1;}
 else {return $n2;}
}

print "Enter 1st number:\n";
chomp(my $a = <STDIN>);
print "Enter 2nd number:\n";
chomp(my $b = <STDIN>);
my $max = max($a, $b);
print "max: $max\n";
```

You can load local variables directly from @_

`use strict` (cont.)

- `use strict` effectively makes all variables local.
- Typing mistakes are easier to catch with `use strict`, because you can no longer accidentally reference `$billl` instead of `$bill`.

- For these reasons, many Perl programmers automatically begin every Perl program with `use strict`.
- It is up to you which style you prefer.

Perl Reference

- A Perl reference is a scalar data type. It holds the location of another value, and is used anywhere a scalar can be used.
- Create a reference by prefixing it with a backslash \

```
    $scalarref = \$foo;
    $arrayref = \@ARGV;
    $hashref = \%ENV;
    $coderef = \&handler;
```

Dereferencing

- Dereferencing returns the value a reference points to the location
- Simply use `$`, `@` or `%` as prefix of the reference variable depending on whether reference is pointing to a scalar, array or hash

```
$var = 10;
$r = \$var;  # reference to scalar variable
print "Value of $var is : ", $$r, "\n";

@var = (1, 2, 3);
$r = \@var; # reference to array
print "Value of @var is : ",  @$r, "\n";
```

Reference to Subroutines

```
# Function definition
sub PrintArr{
   my (@arr) = @_;

   foreach $item (@arr){
      print "Item : $item\n";
   }
}
@arr = qw (C, BASIC, Java, Perl) ;

# Create a reference to above function.
$cref = \&PrintArr;

# Function call using reference.
&$cref(@arr);
```

Brian Storming 1

- In a language (e.g. C), a function definition takes the form

```
function myFunc(arg1, arg2) {
// do something with arg1 and arg2 here }
```

- Yet in Perl it's just

```
sub mySub {
 # @_ holds all arguments passed }
```

- What if I want to restrict the caller to only pass 2 arguments?
- Does Perl has pass-by-reference and pass-by-value?

- Perl was written by a linguist, who knew that human languages are not strongly typed and have no fixed argument lists, so he did not impose these brain damages on Perl users.
- Perl doesn't manage your argument handling for you. Instead, it provides a minimal, flexible abstraction and allows you to write code that fits your needs.

```
sub mysub {
  my ($p1, $p2) = @_;
  ... etc.
}

sub fancy_listy {
    my ($positional, $args, @bad) = @_;
    die "Extra args" if @bad;
}
```

- By default, Perl sticks an alias to each argument in @_. This implements basic, **pass by reference** semantics.

```
my $num = 1;
foo($num);
print "$num\n"; # prints 2.
sub foo { $_[0]++ }
```

- If you want pass by copy semantics, you need to make the copies yourself.

```
sub shifty { my $foo = shift; }
sub listy { my @foo = @_; }
```

Brian Storming 2

- Pass array and scalar to Perl subroutine

```
sub ifin {
    my (@array, $str) = @_;
    for my $i (@array) {
        if ($i eq $str) {
            return 1;
        }
    }
    return 0;
}
```

- Try to check whether a string exists in an array
- But can’t return the correct result
- `ifin` always return 0

```
my @f = qw (apple, banana, pineapple, grape);
my $k = “grape”;
print ifin(@f, $k);
```

Observations from 2 Brainstorms

- You can't pass 'arrays' to subroutines. They're flattened to be a list of scalars.

`ifin(@f, $k);` is the same as `ifin($f[0], $f[1], $f[2], $f[3], $k);`

- One way to pass an array to a subroutine is pass a reference!

```
sub ifin {
  my ($array, $str) = @_;
  for my $e (@$array){
     return 1 if $e eq $str;
   }
     return 0;
}
my @f = (1,2,3,4);
my $k = 1;
print(ifin(\@f, $k), "\n");
```