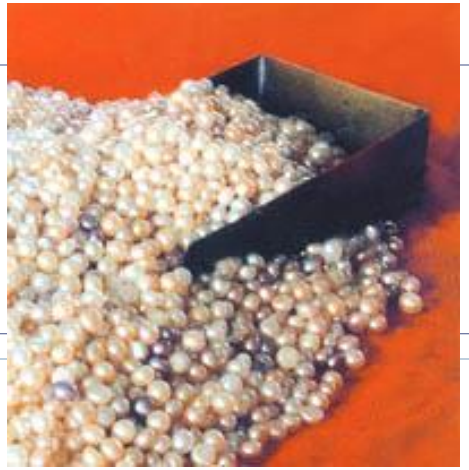
COMP 2021

Unix and Script Programming



More Control Flow

Control Flow

- We have already seen several Perl control flow statements:
  - `if`
  - `while`
  - `for`
  - `last`
- Other control flow statements:
  - `unless`
  - `until`
  - `do while`
  - `do until`
  - `foreach`

unless

- The Perl `unless` statement is like an `if` statement with the condition negated

```
unless($temperature > 10){
     print "wear down jacket!\n";
}
```

- `unless` can have `else`, just like `if`

```
unless($temperature > 10){
     print "wear down jacket!\n";
}else{
     print "wear skirt!\n";
}
```

- `unless` can have `elsif`, just like `if`

```
unless elsif else
```

until

- `until` statement is like a `while` statement with the condition negated.
- It loops indefinitely, until the condition is true
- Sometimes is easier to say "until something is true" rather than "while not this is true"

```
while(!$endoffile){
      ...
}
```

```
until($endoffile){
      ...
} # same thing
```

- `Until` example

```
$resp = "no";
until($resp eq "yes"){
     print "Wakeup [yes/no]? ";
     chomp($resp = <STDIN>);
}
```

do while

- The `do while` statement is like the C++ `do while` statement.
- It loops indefinitely, while the condition is true
- `do while` always executes the body of the loop at least once.

```
do{
        print "Wakeup [yes/no]? ";
        chomp($resp = <STDIN>);
}while($resp ne "yes");
```

do until

- The `do until` statement loops indefinitely, until the condition is true.
- `do until` always executes the body of the loop at least once.

```
do{
        print "Wakeup [yes/no]? ";
        chomp($resp = <STDIN>);
}until($resp eq "yes");
```

foreach

- `foreach` takes a list of values and assigns them one by one to a scalar variable.
- The body of the loop is executed once for each successive assignment.
- `foreach` is similar to the shell programming's `for` statement.

```
foreach $i (@some_list){
        ...
}
```

Access Array Elements with `foreach`

- The following example sums the elements of an array

```
$ cat foreach_sum.pl
#!/usr/local/bin/perl5 -w
@a = (21,32,3,44,75,16,19);
$sum = 0;
foreach $b (@a){
     $sum += $b;
}
print "The array sum is: $sum\n";
$ foreach_sum.pl
The array sum is: 210
```

Reversely Access Array Elements

- The following example prints the numbers in reverse order without changing the array

```
$ cat foreach_rev.pl
#!/usr/local/bin/perl5 -w
@a = (1,2,3,4,5);
foreach $i (reverse @a){
     print "$i  ";
}
print "\n";
$ foreach_rev.pl
5  4  3  2  1
```

- `reverse @a` is the same as writing `reverse(@a)`. Parenthesis are always optional on Perl functions.

foreach with $_

- If you omit the scalar variable in `foreach`, Perl will use `$_` automatically

```
$ cat foreach_rev2.pl
#!/usr/local/bin/perl5 -w
@a = (1,2,3,4,5);
foreach (reverse @a){
        print; # print "$_\t" if you want formatting
}
print "\n";
$ foreach_rev2.pl
54321
```

- `print` (and other Perl functions) use `$_` as the default if nothing is specified.
- `$_` is a global variable that happens to be used by default by many operators.
- It's also `for`'s default iterator

More on `foreach`

- The scalar variable in `foreach` is an alias for each variable in the list, not just a copy. (Tricky!)
- If you modify the scalar variable in `foreach`, the aliased element in the list is also changed

```
$ cat foreach_double.pl
#!/usr/local/bin/perl5 -w
@a = (1,2,3,4);
foreach $b (@a){
        $b *= 2;
}
print "@a\n";
$ foreach_double.pl
2 4 6 8
```

Backward `if`

- A simple way to write "if this, then that" is

```
chomp($user = `whoami`);
print("Hi Bill!\n") if ($user eq "gates");
```

- Backward `if` avoids having to write the curly braces `{ }`.
- There can only be one statement inside the block.
- Backward `if` is a natural and tidy way to exit from a loop

```
 while(1){
        print "Wakeup [yes/no]? ";
        chomp($resp = <STDIN>);
        last if ($resp eq "yes");
};
```

Backward `unless, while, until`

- You can also use backward `unless, while,` and `until` (if there is only one statement in the block)

```
$ cat backward_while.pl
#!/usr/local/bin/perl5 -w
print "Enter numbers to sum (0 to quit): \n";
$sum = 0;
$n = 1;
$sum += $n = <STDIN> while($n != 0);
print "The sum is: $sum: \n";
$ backward_while.pl
Enter numbers to sum (0 to quit):
1
3
0
The sum is: 4
```

&& if

- Another simple way to write "if this, then that" is:

```
chomp($user = `whoami`);
$user eq "gates" && print("Hi Bill!\n");
```

is the same as:

```
chomp($user = `whoami`);
if($user eq "gates"){
      print("Hi Bill!\n");
}
```

&& if

```
this && that;
```

- Why does this work?
- Isn't `&&` the logical-and operator?
- Consider what happens when *`this`* and *`that`* take on values of true and false:
  - If *`this`* is true, then the value of the entire expression is still not known, because it depends on the value of *`that`*. So *`that`* has to be evaluated.
  - If *`this`* is false, there is no need to look at `that`, because the value of the whole expression must be false. Since there is no need to evaluate *`that`*, Perl skips it.

&& if and || unless

- && if is also a tidy way to exit from a loop

```
while(1){
        print "Wakeup [yes/no]? ";
        chomp($resp = <STDIN>);
        $resp eq "yes" && last;
};
```

- Similarly, another simple way to write an unless statement is

```
$temperature > 20 || print "too cold!\n";
```

is the same as:

```
unless($temperature > 20){ # same thing
        print "too cold!\n";
}
```