# Subroutines and Arguments

We can demonstrate the use of subroutines by making a script that accepts 3 arguments and calling a subroutine to make our argument check: **vi sub1.pl**

Text

Description automatically generated Text

Description automatically generated

Here we can see that the variable *test* is accessible and modifiable by the subroutine.

Notes:

* if (check\_arg())
  + This calls the subroutine **check\_arg.**
  + Whatever return value that subroutine provides will be used in the Boolean expression for our if statement.
* sub check\_arg {
  + We define a new subroutine via the **sub** keyword, followed by the name of the subroutine, in this case, **check\_arg.**
* return 0 if (@ARGV != 3);
  + This will return 0 (i.e. false) if the number of arguments provided to the script is not equal to 3.
  + This is then used within the **else** block to mention that not enough arguments have been provided.
  + Execution moves to the next line if (@ARGV != 3) is not evaluated to true (i.e. @ARGV is equal to 3).
* return 1;
  + This will return 1 (i.e. true) if the statement above does not return 0.

# File input and output

Let’s create a new Perl script that opens a file called people.txt and writes out each line to stdout.

Text

Description automatically generated

\*\*The people.txt file has 3 lines in it. Each individual element on each line is separated by the “tab” keypress, or “\t” in Perl.\*\*

**vi greetings\_writer.pl**

Text

Description automatically generated

And the output: Text

Description automatically generated

Notes:

* open (my $people\_file, ‘<’, ‘./people.txt’);
  + This opens a file.
  + First parameter is the variable for the file reference
  + Second parameter is the mode in which to open the file (<, >, >>)
    - We can read (<)
    - We can write (>) (destructive)
    - We can append (>>) (non-destructive)
  + Third parameter is the full path to the file
    - ./ represents current working directory
    - So, people.txt needs to be in the same directory as greetings\_writer.pl
* while(<$people\_file>)
  + <$people\_file> reads one line at a time from the $people\_file reference
    - When it runs out of lines to read, it returns false
  + Loops while there is still input from the file remaining
  + Ie: until it reaches EOF
* print “Line: $\_”;
  + $\_ is the default variable for the while loop.
  + Each iteration, it contains the current line from people\_file

The greetings\_writer.pl reads entire lines of input, but we can utilize the **split** keyword to split text input according to a delimiter. In this case, the delimiter is the TAB key (\t), so we can split the individual words on each line and assign them to local while loop variables:

**vi greetings\_writer2.pl**

Text

Description automatically generated

And the output:

Text

Description automatically generated

Notes:

* ($lname, $fname, $address, $city) = split “\t”;
  + Here, we declare the following variables and split the entire line of input from our file according to our \t delimiter (tab key)
    - $lname would the first string of characters on the line until it hits the tab \t
    - $fname would the second string of characters on the line until it hits the tab \t
    - Similar $address and $city
* Notice in the output that there is a \n character after the $city is displayed before the period. This is because our input file has \n (new line, enter key, etc) after each line.
* We could chomp this out of here if we wanted to.

Next, let’s alter the script so that instead of outputting to stdout (ie, the screen), we’ll output to a file. We’ll also make use of the keyword **die** to terminate the script immediately if we have the incorrect number of arguments. (We’ll take 2 arguments: the input file and the output file).

**vi greetings\_writer3.pl**

Text

Description automatically generated

Output:

Text

Description automatically generated

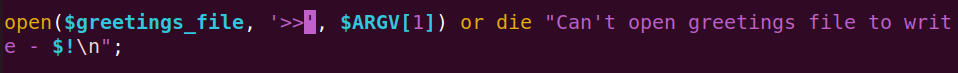
Notes:

* die “USAGE: ./ ……” if @ARGV != 2
  + The **die** keyword will kill the script and perform an action, such as output a string.
  + Can be used in conjunction with if for conditional killing.
* open($people\_file, '<',$ARGV[0]) or die "Can't read people file - $!\n";
  + Opens the file according to the first parameter for reading (<)
  + If we can’t open the file, we’ll kill the script (die) and we’ll display the error message via “$!” keyword.
* open($greetings\_file, '>', $ARGV[1]) or die "Can't open greetings file to write - $!\n";
  + Opens the file according to the second parameter for writing (>)
  + If we can’t open the file, we’ll kill the script (die) and we’ll display the error message via “$!” keyword.

Notice if we run the perl script multiple times, we are being destructive to our output file (ie, we clear the file of any existing data and start over). We can change “>” to “>>” so that we append to the file instead:

**cp greetings\_writer3.pl greetings\_writer4.pl**

**vi greetings\_writer4.pl**



Will result in the following output, assuming you run the script multiple times

Text

Description automatically generated

# Using System Commands

Let’s write another script that parses the output from our **ps** linux command. Maybe not the best way to do it, but we’ll live. We’ll read the output of the command, then use split and Dumper to get the parts we want – usernames and commands.

**vi ps\_boss.pl**

Text

Description automatically generated

And the output:

Text

Description automatically generated

Notes:

* This perl script will run the command ps -Ao “%p,%U,%a”
  + This formats the output of the ps command so that it displays the following separated by a comma (,) delimiter
    - PID
    - User
    - Command
  + We split each line of our ps output via the delimiter “,”
  + We then Dump that line to stdout.
* Try **perldoc -m Data::Dumper** to learn more about this module.

Output of ps\_boss.ps: Output of ps -Ao “%p,%U,%a”

Text

Description automatically generated 

Altering our script to use hashes / nested hashes **{key:value}**.

This will help us parse and clean up our input: **vi ps\_boss2.pl**

Text

Description automatically generated

Notes:

* ($pid,$user,$command) = split ",";
  + Recall our output for our ps command forces the delimiter to be a comma “,”
  + We split each line encountered by our input via “,” and assign the text values to the variables $pid, $user, and $command respectively.
* $pid =~ s/^\s+|\s+$//g;
  + This cleans up our $pid value via regex so that trailing and preceding spaces are trimmed.
    - =~ is an operator to apply a regex to a string.
    - s means that this is a substitution regex, not just matching.
    - / will be used to separate the components – the first component is the pattern to match, the second is the replacement, and the third is options.
    - ^ represents the start of the string, and \s+ represents one or more whitespace characters.
    - | is “or”
    - \s+ is one or more whitespace characters, and $ is the end of the string.
    - So this expression matches one or more whitespace characters at the beginning of the string, or one or more whitespace characters at the end of the string.
    - The replacement is blank – it will remove whatever matches and replace it with nothing.
    - The only option is “g” which means that Perl should keep repeating the substitution until it gets no more matches.
    - Regexes are very powerful, but they do look like gobbledygook.
* next if ($pid eq "PID");
  + If our variable $pid is equal to “PID”, that means we’re reading the first line of our input (with the column headings)
  + We’ll skip this if that’s the case (**next**)
* $procs{$pid}{"user"}=$user;
  + Our %procs hash at our index given by the current $pid value will have an additional value with index “user”. This will have the value of whatever is in the “$user” variable
* $procs{$pid}{"command"}=$command;
  + Our %procs hash at our index given by the current $pid value will have an additional value with index “command”. This will have the value of whatever is in the “$command” variable
* foreach (keys %procs)
  + The **keys** keyword extracts the keys (indexes – in this case, the PID) from the hash and returns them as a list
  + The **foreach** keyword then iterates through this list
* print "$\_ \t $procs{$\_}{user} \t $procs{$\_}{command}\n";
  + We have several occurrences of the default variable $\_
  + We display the first $\_ as our “key”, which is the PID
  + Then we display a tab character (\t)
  + Then we display the user in our corresponding list entry
  + Another tab character
  + Display the command in our corresponding list entry.

Further enhancing the script so that it accepts a process id input and displays user and group information via the **id** command.

**vi ps\_boss3.pl**

Text

Description automatically generated

Text

Description automatically generated

# Fork and Exec

Fork

The **fork** command spawns a child process exactly the same as the parent. The only way to tell them apart initially is by the return value from fork:

* Return value of 0
  + Returned if you are the child
* Return value of <SOME PID>
  + You are a child with <SOME PID>

**vi fork1.pl**

Text

Description automatically generated

Text

Description automatically generated

Notes:

* When we first run **fork1.pl**, Linux will assign the PID of 3861 to that process.
  + It will be different on your VM.
* When we run **fork**, we spawn a child process the same as the current process.
  + Fork can return an integer value, which will be the following:
    - 3861
      * Signifies you are a parent and have a child of 3862.
    - 0
      * Signifies you are a child with no children.

Change it up into an if and then do an “if the child, if the parent type block”.

**vi fork2.pl**

Text

Description automatically generated

Output:

Text

Description automatically generated

Notes:

* Again, $$ prints our current process id (PID)
* fork spawns a copy process as a child to our current process
* Initial process when executing (perl fork2.pl) is **3875**
* The fork command creates a child process with process id (PID) of **3876**

Often, we’ll want to fork off to do something totally independently: **vi fork3.pl**

Text

Description automatically generated

Notes:

* Executing “perl fork3.pl” assigns a process id (PID) of **5051** to the parent process
* Forking results in a child process of **5052** to be spawned
  + This child process loops 10 times and outputs the respective sentence
  + Sleeps for 1 second
* Once execution is done of the child process, we have no way currently of checking if the child process is completed.

Output:

Text

Description automatically generated

Presumably this child is doing something important. Don’t we want to know what?

**vi fork4.pl**

Text

Description automatically generated

Text

Description automatically generated

Notes:

* Executing “perl fork4.pl” spawns a parent process with PID **3891**
* **Fork** will return a PID of **3892**, signifying a child process has been created
* **3892** will repeat and do something important 10 times, with a sleep of 1 second between each.
* **wait**
  + Returns a value to a variable $wait\_pid of what it waited for (in this case child process 3892)
  + Once $wait\_pid = wait; is finished executing:
    - We can read the values returned from and continue execution
    - Wait return 3892
      * Process 3892 is finished
    - Exit code of 99
      * process 3892 finished by the following line: exit 99

Having just one child seems like a waste, doesn’t it?

**vi fork5.pl**

Text

Description automatically generated

Text

Description automatically generated

Notes:

* This script spawns 3 children (CHILD1,CHILD2,CHILD3) with 3 distinct PIDS (3900, 3901, 3902
* Remember that our parent PID is 3899, which, hierarchical representation would like this:

3899 (parent process, executed from perl fork5.pl)

|----> 3900 (first fork, first for loop iteration)

|----> 3901 (second fork, second for loop iteration)

|----> 3902 (third fork, third for loop iteration)

* The last print statement doesn’t really indicate all of our children, is it really waiting for **all** of our children to die? No

What happened to the other pids? **vi fork6.pl**

Text

Description automatically generated

Text

Description automatically generated

Notes:

* Parent process is 5862
* Spawns three children:
  + 5863
    - When done executing returns exit code 1
  + 5864
    - When done executing returns exit code 2
  + 5865
    - When done executing returns exit code 3.

**Exercise**: What if I only care about a particular child? Use **waitpid** to create fork7.pl as follows:

|  |
| --- |
| print "I only care about my youngest child, $fork\_pid. I'll wait for him.\n";  waitpid $fork\_pid;  $error\_code =$? >> 8;  print "My youngest child has died with exit code $error\_code.\n"; |

# Using exec

Usually we’ll actually want to do something other than just do something in the script – that’s what exec is for. Exec runs a command and **never returns**. It essentially replaces the current process with whatever thing you run with exec.

Text

Description automatically generated

Our bash script looks like this:

Text

Description automatically generated

Output:

Text

Description automatically generated

Notes:

* Our endless\_script.bash is no longer endless just to demonstrate a return code
* Once endless\_script.bash is done, it will exit with code 5, which our parent process should detect.
* exec(<some command>)
  + This executes, from Perl, a system command, script, or whatever you want really.
* Overall, this exec1.pl script does the following:
  + Spawns a child process with PID 4352
  + Within that child process, exec the endless\_script.bash Bash script
  + The process exec is executing will never return, so we need to listen to the child process that script is running under instead of the exec process.
  + The bash script returns exit code 5, which is detected by the parent process.