Linux Systems Administration: Sed, Awk, and Perl

Introduction to Scripting

Before We Start Coding

In order to succeed in this course, you need a solid understanding of Linux system administration practices. If you haven't met this prerequisite yet, the OST Linux Systems Administration course series is a good place to learn those skills. The next step in your development as a systems engineer is to gain a working knowledge of system and network security. After all, if you've spent the time to set up a server environment, the last thing you want is a compromise of your systems to force a rebuild. In addition, programmers with the ability to maintain systems' security are becoming increasingly valued by employers.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take the *useractive* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

```
White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove ~~will look like this~~.
```

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

```
The plain black text that we present in these INTERACTIVE boxes is
provided by the system (not for you to type). The commands we want you to type look lik
e this.
```

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

```
Gray "Observe" boxes like this contain information (usually code specifics) for you to
observe.
```

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

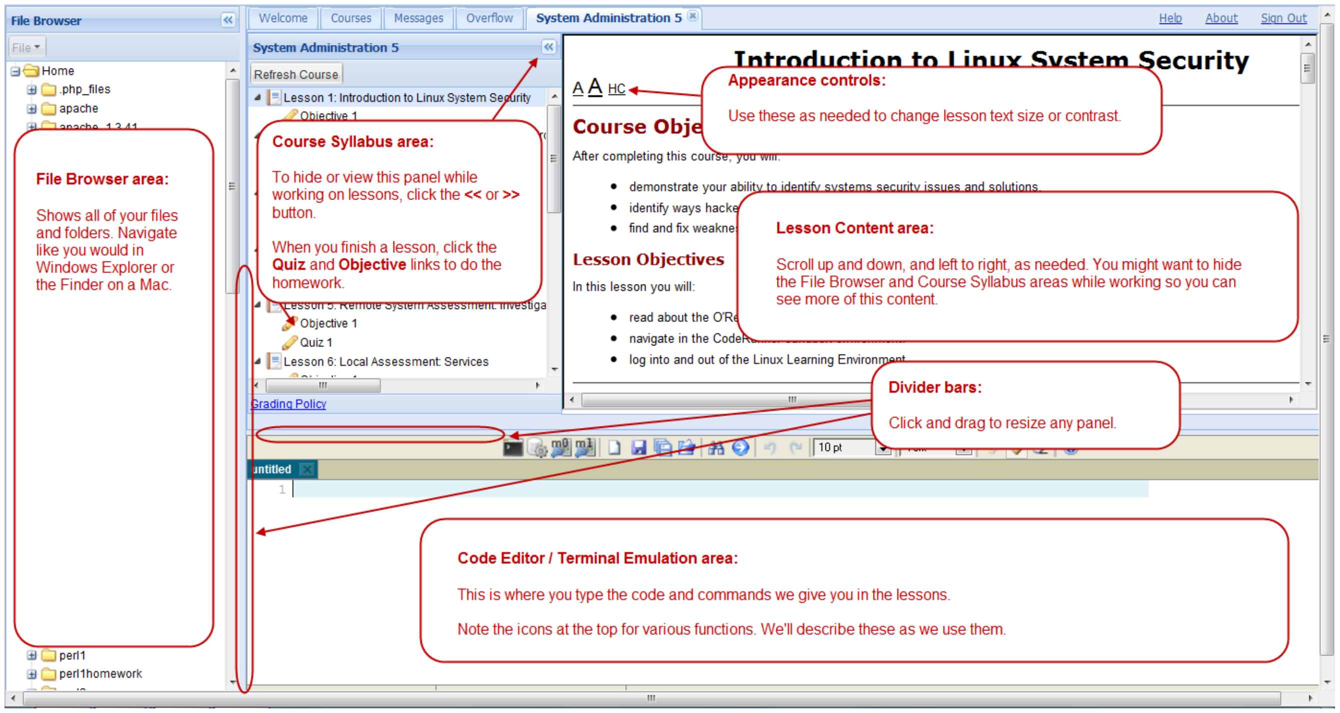
Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:



File Browser area:

Shows all of your files and folders. Navigate like you would in Windows Explorer or the Finder on a Mac.

Course Syllabus area:

To hide or view this panel while working on lessons, click the << or >> button.

When you finish a lesson, click the **Quiz** and **Objective** links to do the homework.

Appearance controls:

Use these as needed to change lesson text size or contrast.

Lesson Content area:

Scroll up and down, and left to right, as needed. You might want to hide the File Browser and Course Syllabus areas while working so you can see more of this content.

Divider bars:

Click and drag to resize any panel.

Code Editor / Terminal Emulation area:

This is where you type the code and commands we give you in the lessons.

Note the icons at the top for various functions. We'll describe these as we use them.

These videos explain how to use CodeRunner:

File Management Demo

Code Editor Demo

Coursework Demo

The Linux Learning Environment

We'll use CodeRunner to interact with the O'Reilly School of Technology *Linux Learning Environment*. The Linux Learning Environment gives you access to one or more Linux virtual machines to use as a tool to learn Linux. The Linux virtual machine that we'll be using is nearly identical in functionality to a physical machine. You'll get your own virtual machine for this course so you'll have complete control over the system.

Using the Linux Learning Environment

In order to use the Linux Learning Environment (LLE), you must connect to one of two LLE machines, m0 or m1. To do that, click the or button in the toolbar above the Code Editor area. Enter the same username and password you use to access this course. For most of our "local" work, you'll log into m0. Click now. In the following session, replace *username* with the name you use to log in to your O'Reilly School of Technology courses, and *password* with your OST password.

INTERACTIVE SESSION:

```
cold1 login: username
Password: password
Attached to CT 1016948 (ESC . to detach) Press Enter
CentOS release 6.4 (Final)
Kernel 2.6.32-042stab076.8 on an x86_64
smiller-m0.unix.useractive.com login: username
Password: password
 [username@username-m0 ~]$
```

What is Scripting?

Scripting is the term applied to a small program that is typically used to make a large job easier or to automate a repetitive task. A script might be run using cron to rotate system logs every few days. You can also use scripts to clean old users out of the password file or to retain statistics about system usage over time.

In this lesson, you will learn how to use **sed** and **awk** on the command line. Beginning with these concepts, you will see how powerful **Perl** can be in the hands of a systems administrator.

Line Editing

Before we get into **sed**, we need to gain a basic understanding of line editing. You are probably very familiar with text editors and word processors that allow you to look at a document while you move around and make changes. This is sometimes called *full text editing* because you can see the entire document.

Line editing, on the other hand, is used to make changes to a single line of a file at a time. This is done using special, editing *commands* rather than making the changes by hand.

Let's use **ed** (a standard Unix line editor) to make this all a little clearer. First we need a file to edit. Copy */httpd/conf/httpd.conf* into your home directory on the cold.

After the command prompt, type the following commands:

```
cold:~$ cp /httpd/conf/httpd.conf .
```

Now we can line edit our copy of the file.

After the command prompt, type the following commands:

```
cold:~$ ed httpd.conf
1168702
p

q
cold:~$
```

The first thing you will notice is that **ed** reports the number of characters in the file. I am not really sure why a line editor reports the number of characters, but it does. After displaying the number of characters, your cursor just sits there without a prompt. **ed** has positioned you at the last line of the file and is waiting for you to tell it what to do. The final instruction used was the **q** command to quit the current editing session. Use **p** to print the current line. In the example above, the last line of the httpd.conf file is empty.

Note The standard Unix method of typing **ctrl+c** to break out of a command will not work with **ed**.

It does not do us much good to just sit on the last line of a file. We can specify a different line by typing a number. For example, if we want to go to line 5, we would just type **5**.

After the command prompt, type the following commands:

```
cold:~$ ed httpd.conf
1168702
5
ServerRoot /httpd
```

Hitting **5** causes **ed** to print out line 5 and make it our current position.

Type the following blue text:

```
d
p
#PidFile /httpd/logs/httpd.pid - moved to /etc/httpd/conf.d/local.inc
w
1168684
q
cold:~$
```

The **d** command deletes the current line. After deleting line 5 we printed out the current line. We are still on line 5, but the old line is gone (we are at what used to be line 6). **w** is typed to save our changes. This *writes* the file. Finally, **ed** displays the updated total number of characters in the file after writing.

Typing one command at a time can be pretty boring though. Lucky for us, **ed** allows us to combine commands into a single string.

After the command prompt, type the following commands:

```
cold:~$ ed httpd.conf
1168684
5dp
ScoreBoardFile /httpd/logs/httpd.scoreboard
wq
1168614
cold:~$
```

The first string says, "go to line 5, delete the line, and then print the current line (the new line 5)." This string can be broken up into two parts: the *address* and the *command*. The address in this case is simply a line number. That is all fine and good, but it is only useful to use a line number if we know exactly where we want to make changes.

Patterns as Addresses

Instead of using line numbers, **ed** allows us to pick lines that match a specified pattern. Patterns are Unix *regular expressions* contained inside two forward slashes (/). Let us find and print the first line that contains "htdocs".

After the command prompt, type the following commands:

```
cold:~$ ed httpd.conf
1168614
/htdocs/
#DocumentRoot /httpd/htdocs
```

Chances are, there are multiple lines that will match that pattern. We can use **g** to run a *global* command.

Type the following blue text:

```
g/htdocs/
#DocumentRoot /httpd/htdocs
#DocumentRoot "/home/webpages/userworld/htdocs"
```

The print command is implied. Let us look at what we have. This is a global pattern that is a regular expression. The default command is to print out the matching lines. Hmmm... **g**lobal **r**egular **e**xpression **p**rint. You remember the **grep** command right? That is no coincidence.

Imagine for a moment that this is not an important file (actually, it is not, it is just a copy). We can add commands to our address pattern.

Type the following blue text:

```
g/htdocs/d
q
?
q
cold:~$
```

We have just deleted every line that contains "htdocs". Oops. If you try to quit **ed** after making changes and without writing the file first, it will give you a question mark "?". If you want to quit without saving changes, just type **q** a second time.

Search and Replace

Another common use for a line editor is search and replace. The **s** (or *switch*) command exists specifically for this purpose. The **s** command requires that you give it a pattern for which to search and something with which to replace the pattern. It looks like this:

OBSERVE:

```
s/pattern/replacement/
```

Recall that we needed to use the **g** flag in order to use a command on every line that matched the address pattern. The same is true for the switch command. It will only replace the first occurrence of the pattern within a line unless we specify otherwise with a trailing **g**. Put this together with what we already know and we can replace every occurrence of "specific" with "precise".

After the command prompt, type the following commands:

```
cold:~$ ed httpd.conf
1168614
g/specific/s/specific/precise/g
p
# you might expect, make sure that you have preciseally enabled it
```

This command will match any line with "specific" in it and then replace every occurrence of "specific" with "precise." The current line becomes the last line where a change took place. Printing the line shows that the results may not be as we intended. Remember, we are matching a pattern, not just words. That is how "specifically" became "preciseally". A better set of commands would include spaces on either side of "specific" like the following:

Observe the following:

```
g/ specific /s/ specific / precise /g
g/ specific$/s/ specific$/ precise/
g/^specific /s/^specific /precise /
```

The first command would replace "specific" found in the middle of a line. Do you remember what the other two would do?

Handing in a Quiz or Objective

After you have read the lesson you have quizzes and objectives to complete that allow you to demonstrate the concepts you have learned. Under the lesson heading there is an objective and/or quiz item. Click on this to reveal the instructions. When you are finished, scroll down the top half of the Coderunner screen and select the button that reads **Hand in** at the right side of the window. You will use the same procedure to hand in objectives. Please do not use the Drop In box but simply click on the Hand In button to hand in any files created. This button will alert your mentor that you are ready to be evaluated.

Sed

Sed

When we want to write scripts that can use the capabilities of **ed** we look to **sed**. You can think of this as **S**cripting with **Ed**. First, let us learn how to use **sed** on the command line.

Unlike ed, we can not use sed interactively. We have to give sed a file and a list of commands to perform on it. Let us apply some of the same commands we went through with ed to sed to see how sed operates.

After the command prompt, type the following commands:

```
cold:~$ sed -n '/htdocs/p' httpd.conf
#DocumentRoot /httpd/htdocs
#DocumentRoot "/home/webpages/userworld/htdocs"
```

This is the same output we got when we used **g/htdocs/** with ed. One of the major differences between sed and ed is that sed looks at every line in the file automatically. As a result we do not have to include the beginning **g**. Additionally, sed will print out every line in the file regardless of whether it was changed, unless we tell it not to with the **-n** flag. (Printing out every line is useful for output redirection, as we will see later.) Also, distinguishing sed from ed is the inclusion of the **p** command to print out the lines that were changed or, in this case, the lines that matched the pattern of "htdocs".

Let us redo another one of our previous examples.

After the command prompt, type the following commands:

```
cold:~$ sed -n 's/specific/precise/gp' httpd.conf
# Note that from this point forward you must preciseally allow
# you might expect, make sure that you have preciseally enabled it
```

Notice that we did not include the address pattern. Since sed looks at every line of the file, when the address and replacement pattern are the same, it is not necessary to include the address pattern. However, we could replace only on lines that contain "Note".

After the command prompt, type the following commands:

```
cold:~$ sed -n '/Note/s/specific/precise/gp' httpd.conf
# Note that from this point forward you must preciseally allow
```

Multiple Commands

Sed will also execute more than one command as it reads through the lines of a file. There are two ways we can get sed to execute multiple commands simultaneously. One way is to use the **-e** flag for each command we want to execute.

After the command prompt, type the following commands:

```
cold:~$ sed -n -e '/htdocs/p' -e '/Indexes/p' httpd.conf
#DocumentRoot /httpd/htdocs
#DocumentRoot "/home/webpages/userworld/htdocs"
Options FollowSymLinks ExecCGI Indexes Includes
Options FollowSymLinks Indexes Includes
# This may also be "None", "All", or any combination of "Indexes",
Options Indexes FollowSymLinks ExecCGI Includes
Options Indexes MultiViews
```

Let us try another example.

After the command prompt, type the following commands:

```
cold:~$ sed -n -e '/Indexes/p' -e '/ndexes/p' httpd.conf
Options FollowSymLinks ExecCGI Indexes Includes
Options FollowSymLinks ExecCGI Indexes Includes
Options FollowSymLinks Indexes Includes
Options FollowSymLinks Indexes Includes
# This may also be "None", "All", or any combination of "Indexes",
# This may also be "None", "All", or any combination of "Indexes",
Options Indexes FollowSymLinks ExecCGI Includes
Options Indexes FollowSymLinks ExecCGI Includes
Options Indexes MultiViews
Options Indexes MultiViews
# server-generated indexes.  These are only displayed for FancyIndexed
# directory indexes.
```

This proves that both commands get executed on each line. We have already printed the line with the first command, but the second command matches the line as well and ends causing it to be printed again.

Another way to use multiple commands is to separate them with a semi-colon.

After the command prompt, type the following commands:

```
cold:~$ sed -n '/htdocs/s/home/house/g ; /htdocs/p' httpd.conf
#DocumentRoot /httpd/htdocs
#DocumentRoot "/house/webpages/userworld/htdocs"
```

This command replaces "home" with "house" on any line that contains "htdocs." Also, any line with "htdocs" is printed regardless of whether a change took place.

Output Redirection

We can make changes to the lines of a file, but how do we save them? By taking advantage of Unix output redirection. Remember we are using the **-n** option to keep sed from printing out all of the lines. If we remove that option, the entire file, with any changes, will be printed to standard output. We can redirect this output to another file.

After the command prompt, type the following commands:

```
cold:~$ sed '/htdocs/s/home/house/g' httpd.conf > httpd2.conf
```

This command line makes the desired changes to the lines and prints all of the lines to the new file: *httpd2.conf*. When you are not using the **-n** option, you typically do not want to use the print option (**p**) either. Doing so would cause the lines to print out twice.

WARNING Remember, we can not write to the same file from which we are reading. Attempting to do this can cause unpredictable results.

Once the file is written we can move it back to the original file.

After the command prompt, type the following commands:

```
cold:~$ sed '/htdocs/s/home/house/g' httpd.conf > httpd2.conf
cold:~$ mv httpd2.conf httpd.conf
```

Now let us learn how to delete lines that contain "Indexes" or "indexes".

After the command prompt, type the following commands:

```
cold:~$ sed '/[Ii]ndexes/d' httpd.conf > httpd2.conf
```

Instead of deleting lines or replacing whole patterns, we have the option of *translation*. The idea behind translation is that you can replace one set of characters with another set, but they do not have to be in any particular order or next to each other. Here is an example:

After the command prompt, type the following commands:

```
cold:~$ sed -n 'y/DR/dr/ ; /htdocs/p' httpd.conf
#documentroot /httpd/htdocs
#documentroot "/home/webpages/userworld/htdocs"
```

Notice, we are not replacing "DR" with "dr", but instead we are translating any uppercase D to a lowercase D and any uppercase R to a lowercase r. "DocumentRoot" becomes "documentroot". Keep in mind though, the changes are applied on any line with a capital D or R, not just the ones we printed out. You are not restricted to using like letters either. We could have replaced A with z or t with 3.

Sed Script Files

Editing long sed commands at the prompt can be time consuming, especially if we want to use them over and over. To get around this little hassle we write sed scripts. Create a file called *changes.sed* that contains the following sed commands:

Type the following in changes.sed:

```

/Indexes/d
s/ document / letter /g
s/ document$/ letter/
s/^document /letter /
/htdocs/s/home/house/g
w httpd2.conf

```

Here the **w** command writes lines to a file. It allows us create the output file in the script instead of redirecting the output on the command line. Let us add the **-n** flag back into the sed command line.

After the command prompt, type the following commands:

```
cold:~$ sed -nf changes.sed httpd.conf
```

The **-f** flag lets us specify the script file to use when processing our file.

Sed, just like most Unix commands, can read from standard input rather than reading a file. All we have to do is pipe the output from another command into sed.

After the command prompt, type the following commands:

```
cold:~$ ps aux |sed -n 'y/1234567890/abcdefghij/ ; /username/p'
username acdii  j.j  j.b  bcbh acfj pts/j    S    Aprcj   j:jj -bash
username aedge  j.j  j.b  bccf achj pts/b    S    Mayja   j:jj -bash
username cagcj  j.j  j.a  bedh  gdd pts/b    R    ab:df   j:jj ps aux
username cagca  j.j  j.j  acdh  dbh pts/b    S    ab:df   j:jj sed -n y/abcdefgh
```

You can read about many more sed commands on the sed man page. See you at the next lesson!

Awk

Awk is the companion of sed. It works almost the same way, by examining each line of input. The main difference between the two is that awk automatically divides each line into fields, sed does not. Awk divides a line into fields by separating words divided by spaces, but later we will learn how to change the field separator to anything we want. But right now, let us look at how to print out every line that contains "sendmail" using awk.

After the command prompt, type the following commands:

```
cold:~$ awk '/Indexes/ { print $0 }' httpd.conf
Options FollowSymLinks ExecCGI Indexes Includes
Options FollowSymLinks Indexes Includes
# This may also be "None", "All", or any combination of "Indexes",
Options Indexes FollowSymLinks ExecCGI Includes
Options Indexes MultiViews
```

Here we have matched the line and used the **print** command to print out **$0**. Since awk divided the line into fields, we need a way to reference them. **$0** references the entire line. If we wanted to print out just the second field, it would be an easy change.

After the command prompt, type the following commands:

```
cold:~$ awk '/Indexes/ { print $2 }' httpd.conf
FollowSymLinks
FollowSymLinks
This
Indexes
Indexes
```

Cool, huh? But using *httpd.conf* does not really show off the true power of awk. Let us try working with a different file.

After the command prompt, type the following commands:

```
cold:~$ cp /etc/passwd .
```

You are already familiar enough with */etc/passwd* to know that it has colon separated fields. Let us make awk separate by colons instead of spaces.

After the command prompt, type the following commands:

```
cold:~$ awk -F ":" '{ print $1 }' passwd
root
bin
daemon
...
```

This command prints out the first field, which happens to contain the usernames. We can print out whichever field we want though. Having only this basic understanding of awk, we can see immediately how useful it is when combined with other Unix commands. Check out the command below:

Observe the following:

```
kill -9 `ps aux |grep username |awk '{ print $2 }'`
```

The same thing could be done without grep, though it would be slightly slower.

Observe the following:

```
kill -9 `ps aux |awk '/username/ { print $2 }'`
```

The commands inside of the back ticks (`) are executed first and the resulting output is used as a list for **kill -9**. **ps aux** retrieves a list of all of the running processes and the output is piped to awk which prints out the second field of any line that contains "username". The second field happens to be the PID of the processes. The result is that all of the processes owned by that username get killed.

Awk Example

Let us say we want to find out the total number of unique users currently logged into a system. We can retrieve a list of users with the **w** command.

After the command prompt, type the following commands:

```
cold:~$ w
 11:09am  up  1:37,  1 user,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
kbutson  pts/8    cold.useractive. 10:36am  0.00s  0.09s  0.02s  w
```

Hmm...well, that was not very hard. We have just one unique user (there may be more when you try it). Let us look at an example of a server that is a little more active. In your home directory you will find a file called *w.txt*. Let us look at the contents of this file.

After the command prompt, type the following commands:

```
cold:~$ cat w.txt
  3:36pm  up 24 days, 20:17, 22 users,  load average: 0.05, 0.06, 0.08
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
johndoe2 ttyq0    v1740426-a.domai 10:31am  5:03m  0.05s  0.05s  -bash
fsmith   ttypf    work987.useracti  2:01pm  1:20m  4.08s  4.03s  pine
johndoe2 ttyq3    work1.useractive 11:25am 14:12   0.19s  0.19s  -bash
alexandr ttyp6    host.useractive.  8:08am  2:06m  0.08s  0.08s  -bash
johndoe2 ttyq5    work1.useractive 11:28am  3:51m  0.06s  0.06s  -bash
george11 ttypd    work56.useractiv Mon 1pm  0.00s  0.60s  0.07s  w
monkey   ttyq7    work2.useractive  2:02pm 27:21   0.66s   ?     -
george11 ttyq8    work56.useractiv  1:11pm  1:54   1.20s  0.94s  emacs  -l
johndoe2 ttyq2    work1.useractive 11:24am  3:38m  0.22s   ?     -
johndoe2 ttyp0    v1740426-a.domai  1:18am  5:23m  0.29s  0.13s  slogin f
monkey   ttyp1    v1825631-a.domai Mon12pm  3:40   0.27s  0.27s  -bash
george11 ttyp5    v1740426-a.domai  2:08am  6:12m  0.09s  0.09s  -bash
johndoe2 ttyp8    work1.useractive Fri 2pm 18:59   0.23s   ?     -
mouse    ttype    v946166-a.domain 12:41pm 10:59   0.88s  0.84s  /usr/loc
monkey   ttyq9    monkeybar.useract 11:56am 50:37   0.64s  0.51s  emacs i
johndoe2 ttypb    work1.useractive 11:21am  3:23m  0.07s  0.07s  -bash
monkey   ttyp7    v1825631-a.domai Mon12pm 10:58m  0.63s  0.51s  emacs  in
mouse    ttyp3    v946166-a.domain 12:43pm  2:22   1.43s   ?     -
catdog   ttyp9    work52.useractiv  3:06pm  8:31   1.15s  1.13s  pine
monkey   ttyp4    v1825631-a.domai  1:29am 11:33m  0.39s  0.31s  emacs  in
fsmith   ttyqc    work987.useracti  1:19pm  2:06m  0.07s  0.07s  -tcsh
```

If we count these, we will see that there are 7 unique users. Our goal is to retrieve that number directly by using several Unix commands. How do we build something that will accomplish this? Well, think about how you counted them. The first step was to eliminate all of the stuff that did not matter, everything that is not a username. We began by getting rid of the header information provided by **w** so that all of the lines retrieved were of the same format. Lucky for us, **w** had a **-h** flag that suppressed the headers. Output representing **w -h** was stored in a file called *wh.txt*.

The next step is to take all of the lines (which now look pretty much the same) and figure out how to return only the usernames. This is where **awk** comes in handy.

After the command prompt, type the following commands:

```
cold:~$ cat w.txt |awk '{ print $1 }'
johndoe2
fsmith
johndoe2
alexandr
johndoe2
george11
monkey
george11
johndoe2
johndoe2
monkey
george11
johndoe2
mouse
monkey
johndoe2
monkey
mouse
catdog
monkey
fsmith
```

That is already a lot easier to read. Unix provides us with another useful command called **uniq**. This command will remove duplicates from a sorted list. But how do get your list sorted in the first place? Since sorting is a pretty common task, I bet Unix has a **sort** command.

After the command prompt, type the following commands:

```
cold:~$ cat wh.txt |awk '{ print $1 }' |sort |uniq
alexandr
catdog
fsmith
george11
johndoe2
monkey
mouse
```

Almost there. This is really easy to read, but remember our ultimate goal is to find the actual number of unique users. The **wc** command stands for *word count*. With the **-l (a lowercase L)** option, **wc** will return the number of lines (which is also the number of unique users).

After the command prompt, type the following commands:

```
cold:~$ cat w.txt |awk '{ print $1 }' |sort |uniq |wc -l
      7
```

Awesome.

As this example shows, it is useful to think of all these different Unix commands as tools in a toolbox. In most cases, using only one tool won't do the job, but using many tools together in the right order can solve almost any problem.

Basic Shell Scripting

Shell Scripts

We are already familiar with the concept of a Unix *shell*. It is the environment that allows you interact with the server. We use the shell by issuing it a series of commands at the prompt. Many times, we find ourselves issuing the same set of commands over and over. When this is the case, we can write a *shell script* to speed things up. A *script* is a term often used to refer to a small program.

Shell scripts are mostly just Unix commands with the addition of variables and logical statements. These variables and statements are capabilities built in to your shell. Each shell (sh, tcsh, etc) has a little different syntax, but we will continue focusing on **bash**.

Let us take a look at a very basic shell script. Edit a file called *script.sh*.

Add the following lines to script.sh:

```
#!/bin/bash

uptime
```

Save this file. The first line needs to be in every bash shell script. It indicates the location of **bash** on the server. We need to make this script executable before we can run it.

After the command prompt, type the following commands:

```
cold:~$ chmod 755 script.sh
cold:~$ ./script.sh
  2:09pm  up 28 days,  2:25,  2 users,  load average: 0.00, 0.00, 0.00
```

Edit *script.sh* again.

Change script.sh to contain this line:

```
#!/bin/bash

w -h |awk '{ print $1 }' |sort |uniq |wc -l
```

Save *script.sh* and run it again.

After the command prompt, type the following commands:

```
cold:~$ ./script.sh
      7
```

Now, anytime we want to find out the number of unique users on the system, instead of typing out that whole huge command line, all we have to do is run *script.sh*.

Shell Variables

Just like our shell has environment variables (ex: PATH), our shell scripts may have variables too. These variables are assigned values and used in exactly the same way they are on our shell. Edit a new file called *var.sh*.

Add these lines to var.sh:

```
#!/bin/bash

VAR="hello"
echo $VAR
```

When we want to assign a value to a variable, we simply indicate the variable name. However, when we want to access the variable, we need to use the "$" symbol. Save and execute *var.sh*.

After the command prompt, type the following commands:

```
cold:~$ chmod 755 var.sh
cold:~$ ./var.sh
hello
```

In fact, we can even access existing environment variables.

Add this line to var.sh

```
#!/bin/bash

VAR=hello
echo $VAR
echo $PATH
```

Again, let us save and execute *var.sh*.

After the command prompt, type the following commands:

```
cold:~$ ./var.sh
hello
/users/username/.gemhome/bin:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sb
in
```

The next step is to assign the output of a command (or set of commands) to a variable that we can use later. Edit the previous *script.sh* file again.

Add these lines to script.sh:

```
#!/bin/bash

NUM=`w -h |awk '{ print $1 }' |sort |uniq |wc -l`
echo $NUM
```

Here, we have added backward single quotation marks around our command string to indicate that we want to store the output in **NUM**. By the way, the backward quotation mark key is the same key as the tilde (~) key.

After the command prompt, type the following commands:

```
cold:~$ ./script.sh
2
```

IF Statements

Many times, we will want to execute a certain series of commands only if a variable has a specific value. To test for these *conditions*, we need to use an **if** statement. Let us try it. Edit *script.sh* again.

Add these lines to script.sh:

```
#!/bin/bash

NUM=`w -h |awk '{ print $1 }' |sort |uniq |wc -l`
echo $NUM

if [ $NUM = 2 ]; then
   w
   echo $NUM
fi
```

Adjust the condition depending on the number of users currently on the cold when you try this example. Save and execute *script.sh*.

After the command prompt, type the following commands:

```
cold:~$ ./script.sh
  2:42pm  up 31 days,  2:58,  2 users,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU  WHAT
username pts/1    home.server.co   25Sep01  1:23m  0.12s  0.12s  -bash
username pts/2    home.server.co   25Sep01  0.00s  0.22s  0.02s  bash ./scr
1
```

Here we can see the script in action. First it gathers the number of unique users on the system. If there is one user, it will execute the **w** command and print out the value of **NUM**. This reads as "If NUM is equal to 1, then perform the desired operations".

Note that the syntax (format and spacing) of the **if** statement must be exact. For example, this will not work:

Observe this incorrect syntax:

```
if [ $NUM = 1]; then
   w
   echo $NUM
fi
```

Also, the **if** statement must end with **fi**.

That works well when comparing numerical values, but we also need to be able to compare strings. In order to compare strings, we use double quotation marks.

Add these lines to script.sh:

```
#!/bin/bash

NUM=`w -h |awk '{ print $1 }' |sort |uniq |wc -l`

if [ $NUM = 1 ]; then
   w
   echo $NUM
fi

if [ $USER = "username" ]; then
   ps aux |grep $USER
fi
```

As always, replace "username" with your own login. Here we are checking to make sure the **USER** environment variable is equal to our username.

After the command prompt, type the following commands:

```
cold:~$ ./script.sh
  3:28pm  up 31 days,  3:44,  2 users,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU  WHAT
username pts/1    home.server.co   25Sep01  2:09m  0.12s  0.12s  -bash
username pts/2    home.server.co   25Sep01  0.00s  0.24s  0.02s  bash  ./scr
1
username  8210  0.0  0.2  2336 1376 pts/1    S    Sep25   0:00 -bash
username  8694  0.0  0.2  2332 1364 pts/2    S    Sep25   0:00 -bash
username 30964  0.0  0.1  2000  972 pts/2    S    15:28   0:00 bash  ./script.sh
username 30972  0.0  0.1  2548  744 pts/2    R    15:28   0:00 ps aux
username 30973  0.0  0.1  1520  592 pts/2    S    15:28   0:00 grep  username
```

Wrap Up

Shell scripts are used most often during system startup when other scripting languages might not be available. Take at look at */etc/rc.d/rc.sysinit* for a good shell script example.

With the increasing popularity of Perl, people are using shell scripts much less for day-to-day scripting needs. As a result, we will be focusing on Perl for the rest of this course.

Perl

What is Perl?

The first version of Perl (*Practical Extraction and Report Language*) was written by Larry Wall in 1987. Since its creation, Perl has become popular with systems administrators because of its ease of use and power with text manipulation. It is more versatile than sed or awk and is typically easier to write than other languages. Perl is an immensely popular language for writing CGI scripts, but we will be concentrating on its usefulness to systems administrators.

Perl is an interpreted language, meaning that its instructions are converted into machine code while a Perl script is executed. Conversely, a compiled language is converted to machine language before its execution.

We are going to spend the next few lessons becoming familiar with the fundamentals of Perl syntax before applying it to systems administrator tasks.

Note If you have taken our CGI/Perl class, a lot of the Perl syntax will be familiar to you. However, the application of Perl in a non-CGI environment will be something you are not familiar with yet.

Getting Started

The first thing we need to do is locate the Perl executable. We can find it by using the code below:

After the command prompt, type the following commands:

```
cold:~$ which perl
/usr/local/bin/perl
```

The very first line of our Perl scripts will contain the location of the Perl executable. Let us go ahead and write our first Perl script. Edit a file called *test.pl* so that it contains the following lines:

Add these lines to test.pl

```
#!/usr/local/bin/perl

print "blaa";
```

This may sound a tad obvious, but it is important to make sure that the first line starts at the very beginning on the far left side. Also, notice that the print statement ends in a semi-colon. Most lines in Perl will end in a semi-colon because that denotes the end of a statement. Save the script and exit your editor.

After the command prompt, type the following commands:

```
cold:~$ ./test.pl
bash: ./test.pl: Permission denied
```

Hmm...that is not good. We have to make the script executable before we can use it.

After the command prompt, type the following commands:

```
cold:~$ chmod 755 test.pl
cold:~$ ./test.pl
blaacold:~$
```

We just printed out "blaa", but it would be a little nicer if it were on a line by itself.

Make this change to test.pl

```
#!/usr/local/bin/perl

print "blaa\n";
```

The backslash (\) is called the *escape character* in Perl. This indicates to Perl that the character right after it should be treated specially. For instance, the combination of **\n** represents a new line. The escape character can also be used to print quotes that would otherwise interfere with our print statement.

Make this change to test.pl:

```
#!/usr/local/bin/perl

print "\"blaa\"\n";
```

Save test.pl and try it out.

After the command prompt, type the following commands:

```
cold:~$ ./test.pl
"blaa"
```

There are many more escape characters that we will learn about as we go.

Variables

Just printing out lines of text does not do us much good on its own though. Let us go over some different variable types in Perl. A *variable* can be thought of as a box that contains a piece of data. When storing something in a variable, it does not matter if it is a number or a string.

Make the following changes to test.pl:

```
#!/usr/local/bin/perl

$var = 3;
$var2 = "This is a string";

print "$var\n";
print "$var2\n";
```

Save and test.

After the command prompt, type the following commands:

```
cold:~$ ./test.pl
3
This is a string
```

Regular variables in Perl always start with **$**. Here, we have created a couple of simple variables and printed them out. They are not particularly useful individually though. We need to be able to change the values of those variables and use them in conjunction with each other.

Operators

In the previous example, when we created our variables we gave them values with the *assignment operator* or equals sign (=). We can also perform mathematical operations on them. Let us create a new file called *operator.pl* with the following lines:

Add these lines to operator.pl:

```
#!/usr/local/bin/perl

$var1 = 3;
$var2 = 5;

$var3 = $var1 + $var2;             # $var3 equals 8  (3+5)
$var4 = $var3 * $var2;             # $var4 equals 40 (8*5)
$var5 = ($var4 / $var2) - $var3;   # $var5 equals 0  ((40/5) - 8)

print "$var3 $var4 $var5\n";
```

Here we can see that basic math operations work just fine. Test this script to make sure you get the right values. Remember, you will have to make the file executable before you can run it.

Note Comments in Perl start with a pound sign (#). Anything after a pound sign is ignored when the script is executing. Comments are used to explain the code to other programmers who look at your code. It is always a good idea to comment your code especially as your code gets more complex.

Make the following changes to operator.pl:

```
#!/usr/local/bin/perl

$var1 = 3;
$var2 = 5;

$var1 = $var1 ** $var2;    # $var1 equals 243
$var3 = $var1 % $var2;     # $var3 equals 3

print "$var1 $var3\n";
```

Notice that we can use a variable in the operation to assign it a new value. Here we have used the current value of **$var1** to come up with a new one. The exponent operator (**) raises **$var1** to the power of **$var2**.

An operator you might not be familiar with is the *modulus* (%). This is not a percentage operation. The modulus acts like a division operation except that, instead of returning the answer, it returns the remainder. For example: 5 % 2 = 1 (5 divided by 2 equals 2 with a remainder of 1).

All of these operations deal with numbers, but we know that a variable can store a string as well. Two strings can be "added" together with the *concatenation* operator (which happens to be a period).

Make the following changes to operator.pl

```
#!/usr/local/bin/perl

$var1 = 3;
$var2 = 5;
$string1 = "This is my ";
$string2 = "sentence.";

$var1 = $var1 ** $var2;
$var3 = $var1 % $var2;
$string1 = $string1.$string2."  Neat, huh?";

print "$var1 $var3\n";
print "$string1\n";
```

The *concatenation* operator simply pushes all the strings together into one string. Save *operator.pl* and test it out.

After the command prompt, type the following commands:

```
cold:~$ ./operator.pl
243 3
This is my sentence.  Neat, huh?
```

Perl provides us with shortcuts for a lot of operations. Here is a table of operator shortcuts and their expanded versions.

Shortcut	Fully Expanded
$var += 3;	$var = $var + 3
$var -= 3;	$var = $var - 3
$var /= 3;	$var = $var / 3
$var *= 3;	$var = $var * 3
$var **= 3;	$var = $var ** 3
$var++;	$var = $var + 1
$var--+= 3;	$var = $var - 1 + 3

If Statements and Loops

If Statements

Often times we will need to perform a test on a variable before executing some code. We do this using an *if* statement. These statements are of the form "if, then":

Observe the following:

```
if ( condition ) {
      operations;
}
```

If the *condition* turns out to be true, then the *operations* are executed. If the *condition* turns out to be false, then the *operations* are not executed. In programming, **true** is also represented as and interpreted as **1**, while **false** is represented and interpreted as **0**.

Let us look at a couple examples to see this in action. Create a new file called *if.pl*.

Add the following code to if.pl

```
#!/usr/local/bin/perl

$var1 = 3;
$var2 = 4;

if($var1 < 5){
   print "first test true\n";
}

if($var1 >= -2){
   print "second test true\n";
}

if($var1 == 4){
   print "third test true\n";
}else{
   print "third test false\n";
}
```

The first **if** statement tests to see if the variable is less than five. If so, the **print** statement is executed, otherwise the script continues after the closing bracket. The second **if** statement is checked regardless of the result of the first one. The same is true for the third **if** statement. In this last case, we are testing to see if the variable is equal to four. Notice that the test for equality is a double equals sign (==); this is not the same as the assignment operator (=). Additionally, if the variable is not equal to four, everything inside of the **else** statement is executed.

Let's save *if.pl* and try it out. Be sure to change the file permissions first.

After the command prompt, type the following commands:

```
cold:~$ chmod 755 if.pl
cold:~$ ./if.pl
first test true
second test true
third test false
```

Here is a table of comparison operators and their uses:

Comparison	Operator	Example
equal	==	($var1 == $var2)
not equal	!=	($var1 != $var2)

less than	<	($var1 < $var2)
greater than	>	($var1 > $var2)
less than or equal	<=	($var1 <= $var2)
greater than or equal	>=	($var1 >= $var2)
not	!	(!($var1 < $var2))

All of the above operators deal with numbers (with the exception of ! (not) which will invert any conditional statement). But what about when we are comparing strings? When comparing strings we have to use a different set of operators.

Comparison	**Operator**	**Example**
equal	eq	($var1 eq $var2)
not equal	ne	($var1 ne $var2)
less than	lt	($var1 lt $var2)
greater than	gt	($var1 gt $var2)
less than or equal	le	($var1 le $var2)
greater than or equal	ge	($var1 ge $var2)
not	!	(!($var1 lt $var2))

So, you are probably wondering how one string can be less than another string? When Perl compares two strings this way, it tests strings for how they rank alphabetically. For example, "abc" would be less than "xyz."

Embedded Ifs and Multiple Conditions

How would we go about testing for more than one thing before executing a block of code? The first solution to this problem is to include multiple if statements inside of each other. Change your *if.pl* to look like the following:

Make the following changes to if.pl:

```
#!/usr/local/bin/perl

$var1 = 1;
$var2 = 0;
$var3 = 5;

if($var1 == 1){
   if($var2 == 0 ){
      print("\$var2 is 0\n");
   }
   if($var2 == 1){
      print("\$var1 is 1 and \$var2 is 1\n");
   }else{
      if($var3 == 2){
         print("\$var1 is 1, \$var2 is not 1 or 0, and \$var3 is 2\n");
      }
   }
}
```

Here we have *embedded* **if** statements. The second **if** would not be checked unless the first one was true. Notice that the **if** statements are indented to match up with their closing brackets and the code inside is indented even farther. This is not a requirement for Perl syntax, but it helps to keep things straight. If every statement was written justified to the left, it would be much harder to tell where one **if** statement ended and the next one began. Note also that we have used the escape character (\) in order to print "$". Without the escape character, Perl would return the value of the variable in its place.

We can make the above code run a little faster by using **elsif**, which stands for *else if*.

Make the following changes to if.pl:

```
#!/usr/local/bin/perl

$var1 = 1;
$var2 = 0;
$var3 = 5;

if($var1 == 1){
   if($var2 == 0 ){
      print("\$var2 is 0\n");
   }elsif($var2 == 1){
      print("\$var1 is 1 and \$var2 is 1\n");
   }else{
     if($var3 == 2){
         print("\$var1 is 1, \$var2 is not 1 or 0, and \$var3 is 2\n");
     }
   }
}
```

This code has the same output as before. Save and test *if.pl* to see the results.

Embedded **if** statements are not the only way to test for multiple conditions. We can use several tests for the same condition.

Make the following changes to if.pl:

```
#!/usr/local/bin/perl

$var1 = 1;
$var2 = 0;
$var3 = 5;

if(($var1 == 1) && ($var2 == 1)){
   print("\$var1 is 1 and \$var2 is 1\n");
}

if(($var1 == 1) && ($var2 != 1) && ($var3 == 2)){
   print("\$var1 is 1, \$var2 is not 1, and \$var3 is 2\n");
}

if(($var1 == 1) || ($var2 == 1)){
   print("\$var1 or \$var2 equals 1\n");
}
```

The **if** statements are still testing to see if the conditions are true, but the conditions are a little more complicated. In the first case, the **print** statement will only be executed if **$var1** and **$var2** are equal to 1. A double ampersand (&&) represents "AND", while a double pipe (||) represents "OR". The condition for the last **if** statement returns true if either one (or both) of the tests is true.

Let's test this code.

After the command prompt, type the following commands:

```
cold:~$ ./if.pl
$var1 or $var2 equals 1
```

String Conditions

We mentioned before that strings do not use the normal conditional operators. Let us do a quick example to show this in action. Edit a new file called *ifstring.pl*.

Add the following code to ifstring.pl:

```
#!/usr/local/bin/perl

$var1 = "hello";
$var2 = "goodbye";
$var3 = "hello";

if($var1 eq $var3){
  print "These two strings are the same: $var1 $var3\n";
}
if($var1 ne $var2){
  print "These two strings are not the same: $var1 $var2\n";
}
if($var1 lt $var2){
  print "$var1 comes before $var2\n";
}else{
  print "$var1 comes after $var2\n";
}
```

Save and run *ifstring.pl*.

After the command prompt, type the following commands:

```
cold:~$ chmod 755 ifstring.pl
cold:~$ ./ifstring.pl
These two strings are the same: hello hello
These two strings are not the same: hello goodbye
hello comes after goodbye
```

Loops

While Loops

An extension of the **if** statements in the previous lesson is the ability to keep checking the condition and keep executing the code over and over. This is done with *loops*. let us start by looking at **while** loops. Create a new file called *while.pl* to use for these examples.

Add the following code to while.pl:

```
#!/usr/local/bin/perl

$i = 0;

while($i < 3){
  print "$i\n";
}
```

Save and test this example. Be sure to change the file permissions before testing it.

$i starts out as zero which causes the condition of the **while** statement to be true. The value of **$i** is printed out and the condition is checked again. The condition is still true so the **print** statement is executed yet again. This is what's known as an *infinite loop*. It will go on forever until we stop it. If you're testing out a program and it seems to be stalled, there's a good chance you've got an accidental infinite loop.

Note Remember, we can hit **Ctrl+c** to break out of a program in Unix.

let us modify our program slightly so that it will not run forever.

Make the following changes to while.pl:

```
#!/usr/local/bin/perl

$i = 0;

while($i < 3){
  print "$i\n";
  $i++;
}
```

Excellent. Now, **$i** will be incremented by one every time through the loop. The value will be printed out when it is 0, 1, and 2, but when **$i** equals 3 the condition will be false and the loop will end. let us check it out.

After the command prompt, type the following commands:

```
cold:~$ ./while.pl
0
1
2
```

Pretty simple, right? Later we will see that we are not limited to simple math tests in our conditions. Lots of Perl functions will return true or false values based on the success of what they were trying to do. There are many ways to provide useful test conditions for a loop.

For Loops

Using a variable such as **$i** to loop a specific number of times is very common. There is another type of loop that is built perfectly for this situation. The difference between the two is that with a **for** loop the initial value, condition, and change are all contained in one spot. let us create a script called *for.pl* to demonstrate the workings of a "for loop".

Make the following change to while.pl:

```
#!/usr/local/bin/perl

for($i=0 ; $i < 3 ; $i++){
  print "$i\n";
}
```

Notice that the three parts of the **for** statement are separated by semi-colons. And we aren't limited to using such simple increments, any expression will do.

Last and Next

There are many times that we don't want to execute all of the code within a loop. Maybe we've met the first condition, but if another condition is met we want to exit the loop immediately. Perhaps we don't want to exit the loop, but just start over at the beginning. Both of these situations are made possible by **last** and **next**.

last works just how its name implies. It makes the current iteration of the loop the last one. The loop will not even finish executing the rest of its code block. Go back and modify your *while.pl* script.

Make the following change to while.pl:

```
#!/usr/local/bin/perl

$i = 0;

while($i < 10){
  print "$i\n";
  if($i > 5){
    print "Oh no!  \$i is greater than 5!\n";
    last;
  }
  $i++;
}
```

After the command prompt, type the following commands:

```
cold:~$ ./while.pl
0
1
2
3
4
5
6
Oh no!  $i is greater than 5!
```

Is this the output you expected?

Now let us take a look at **next**.

After the command prompt, type the following commands:

```
cold:~$ ./while.pl
0
1
2
3
4
5
9
```

When **$i** was equal to 5, we added 4 to it and started the loop over. Notice that it skipped the last part of the loop, where it would have normally incremented **$i** by one.

Arrays and Hashes

Arrays

So far, all we have been using is regular variables to store a number or a string. What if we have a whole set of data we want to store? Do we have to create a bunch of variables? No, luckily we can store them in an *array*. An array is simply a list of variables that are usually used to store a series of related data. Whether it contains a bunch of numbers or lines from a file, arrays are used the same way. Edit a new file called *array.pl*.

Make the following changes to array.pl:

```
#!/usr/local/bin/perl

@myarray = (1,2,3,4,5);
@words = ("one","two","three","four","five");

print "$myarray[0] $myarray[2]\n";
print "$words[0] $words[2]\n";
```

When we reference an element of an array we have to use the dollar symbol ($) again, just as if it were a regular variable. Square brackets are used to indicate the *index* or position in the array. Arrays start with an index of zero (0). This means that the first element of the array is at 0, the second at 1, and so on. Save *array.pl* and try it out. Be sure to change the file permissions before testing it.

Perl defines an array called @ARGV for us automatically. It contains command line arguments passed to the script when it is executed.

Make the following changes to array.pl:

```
#!/usr/local/bin/perl

print "Maximum index: $#ARGV\n";
print "Last argument: $ARGV[$#ARGV]\n";
```

$#ARGV will contain the number of the last index in the array. This turns out to be the total number of elements minus one. Let us test this script out a few times to see how it works.

After the command prompt, type the following commands:

```
cold:~$ ./array.pl one
Maximum index: 0
Last argument: one
cold:~$ ./array.pl one two three
Maximum index: 2
Last argument: three
cold:~$ ./array.pl
Maximum index: -1
Last argument:
```

Here we can see that it always prints out the last argument passed to the script.

Printing Array

We have seen how to print out individual elements of an array, but what if we wanted to print out the whole thing? We could try this:

Make the following changes to array.pl:

```
#!/usr/local/bin/perl

@words = ("one","two","three","four","five");

print @words;
```

But that probably is not the result we want.

After the command prompt, type the following commands:

```
cold:~$ ./array.pl
onetwothreefourfivecold:~$
```

Yuck. That is horrible. It would be nice if we could print each element, one at a time, and add a newline or something. We need a way to start at the beginning of the array and stop at the end. Using a **while** loop solves this problem.

Make the following changes to array.pl:

```
#!/usr/local/bin/perl

@words = ("one","two","three","four","five");

$i=0;
while($i <= $#words){
   print "$words[$i]\n";
   $i++;
}
```

The index of the array starts at zero so that is where we will start our variable. We increment **$i** every time through the loop (progressing through the elements of the array) as long as it is less than or equal to the maximum index. The results look a little better than before.

After the command prompt, type the following commands:

```
cold:~$ ./array.pl
one
two
three
four
five
```

This method is obviously not just useful for printing an array. It allows us to perform operations on each element of the array separately. We will get lots of practice with this concept in the later lessons.

Foreach

A **foreach** loop allows us do something *for each* element in a list. Check this out:

Make the following changes to array.pl:

```
#!/usr/local/bin/perl

@words = ("one","two","three","four","five");

foreach $elem (@words){
  print "$elem\n";
}
```

Pretty nice, huh? The output is exactly the same as before, plus the code is a little easier to understand.

Hashes

Hashes are a lot like arrays in that they store a list of data. Instead of a numbered index, hashes associate values with a *key*. These keys are used to reference the values much like the index of an array. Hashes are sort of like small databases. let us take a look at a pre-defined hash that stores environment variables by creating a script called *hash.pl*.

Make the following changes to hash.pl:

```
#!/usr/local/bin/perl

print %ENV;
```

To reference a whole hash we use **%**. Test this script to see what happens. It printed out a big mess, right? Referencing the whole hash turns out not to be very useful. Instead we can print out a hash value by using its key.

Make the following changes to hash.pl:

```
#!/usr/local/bin/perl

print "$ENV{'USER'}\n";
```

The "USER" environment variable stores your username. To get the value from a hash we have to change the **%** to a **$** and include the key inside braces after the hash name.

After the command prompt, type the following commands:

```
cold:~$ ./hash.pl
username
```

We can also use a variable to reference the value.

Make the following changes to hash.pl:

```
#!/usr/local/bin/perl

$key = "USER";
print $ENV{$key}."\n";
```

Save and test.

After the command prompt, type the following commands:

```
cold:~$ ./hash.pl
username
```

Notice the use of the period (concatenation operator) to push two strings together.

Using Foreach with Hashes

The **foreach** loop is not useful only for arrays. Since hashes do not have an index like arrays do, we can not traverse them with a while loop very easily. If we could grab the keys from the hash and do work for each one of them, things would be a lot easier. Guess what? We can. One of the great things about Perl is that manipulating data in any specific way is likely possible.

Make the following changes to hash.pl:

```
#!/usr/local/bin/perl

%names = ("john","smith",
          "sally","johnson",
          "bill","peterson");

foreach $key (keys(%names)){
  print "$key $names{$key}\n";
}
```

Here we defined a hash by giving it a list of key/value pairs. We used the **keys** function to get a list of the keys from our hash. Each key was then used within **foreach** to print out our hash table.

After the command prompt, type the following commands:

```
cold:~$ ./hash.pl
john smith
sally johnson
bill peterson
```

If we did not want to use the keys at all, we could use the **values** function instead of the **keys** function.

Perl also provides us with the **sort** function that will let us alphabetize our list with ease.

Make the following changes to hash.pl:

```
#!/usr/local/bin/perl

%names = ("john","smith",
          "sally","johnson",
          "bill","peterson");

foreach $key (sort(keys(%names))){
  print "$key $names{$key}\n";
}
```

After the command prompt, type the following commands:

```
cold:~$ ./hash.pl
bill peterson
john smith
sally johnson
```

Input

Getting Input

Up until this point we have really just been learning about basic Perl syntax. Now it is time to get some real work done. Let us try to do the same sort of things we were doing with **sed** by searching for a pattern in a file and printing out the line. First, we have to get input.

One simple way of getting input is to capture the output of another Unix command. We do this by using back ticks. Edit *input.pl*.

Add the following code to input.pl:

```
#!/usr/local/bin/perl

@input = `cat /etc/protocols`;

foreach $line (@input){
  if($line =~ /ICMP/){
    print $line;
  }
}
```

Some of this we have seen before and some we have not. Getting our input takes place as an assignment operation for an array. **@input** will contain an element for each line of the contents from */etc/protocols*. We already know that we can use **foreach** to look at each line of an array. All we need to do now is to search for a pattern. In the **if** statement we have another condition operator that we have not seen before. Using **=~** allows us to match our variable against a pattern. In this case, we want to see if **$line** contains "ICMP". If a line matches the pattern it is printed out, otherwise the **foreach** loop just goes on to the next line.

Note The companion operator to **=~** is **!~**. It allows us to match all lines that *do not* contain a specific pattern.

After the command prompt, type the following commands:

```
cold:~$ chmod 755 input.pl
cold:~$ ./input.pl
icmp     1       ICMP            # internet control message protocol
ipv6-icmp        58      IPv6-ICMP       # ICMP for IPv6
```

In More Ways Than One

With Perl is that there is almost always more than one way to perform a particular task. Instead of **cat**ing a file inside of the Perl script, why not just use a Unix pipe? We know how to pipe output to a command, so we only need to learn how to read from standard input (STDIN) with our script. Not a problem.

Make the following changes to input.pl:

```
#!/usr/local/bin/perl

@input = <STDIN>;

foreach $line (@input){
  if($line =~ /ICMP/){
    print $line;
  }
}
```

The only difference here is that we are getting our input from STDIN instead of from a Unix command directly. Let us see it in action.

After the command prompt, type the following commands:

```
cold:~$ cat /etc/protocols | ./input.pl
icmp     1        ICMP            # internet control message protocol
ipv6-icmp        58       IPv6-ICMP       # ICMP for IPv6
```

This method of grabbing the contents of the file allows us to change the input without modifying the script.

Still More Ways

A variation on the above technique is to eliminate the use of the **@input** array altogether. We can use a **while** loop to perform operations on the lines of STDIN while they are coming into the script. This takes advantage of the fact that STDIN is a *stream*. A stream can be thought of as a continuous flow of data instead of a static amount that you might read from a file. The condition of the **while** loop will continue to be true as long as it hasn't reached the end of the input stream.

Make the following changes to input.pl:

```
#!/usr/local/bin/perl

while(<STDIN>){
  if($_ =~ /ICMP/){
     print $_;
  }
}
```

What is **$_** all about? This is a special variable that Perl uses to represent a default pattern matching space. It will be equal to each line of STDIN and can be used just like any other variable. The neat part about **$_** is that since it is the default space, it is not always necessary to include it. Check this out:

Make the following changes to input.pl:

```
#!/usr/local/bin/perl
while(<STDIN>){
  if(/ICMP/){
     print;
  }
}
```

That looks a lot simpler, correct? Instead of piping in the contents of a file as input, let us use this script to observe the **while** loop treating the input as a stream. If we run the script by itself, it is still expecting input, so we have to type it ourselves.

After the command prompt, type the following commands:

```
cold:~$ ./input.pl
hello
what is happening?
ICMP
ICMP
we are creating the stream as we go
to end the stream, hit Ctrl+d
```

We can see what we are typing, but behind the scenes our script is using it as input. This becomes apparent when we type ICMP. As soon as we hit enter on that line it is used as the next pattern space in the **while** loop. It matches the pattern and our script prints out ICMP. Then we continue on with the stream. To end the stream we have to hit **Ctrl+d**. In Unix, this sends an EOF (*end of file*) which indicates the end of the input stream. The loop will end and Perl will continue on with the rest of the script.

Opening a File for Reading

Good grief. How many ways are there to get input? Lots.

Instead of grabbing stuff from standard input or capturing the output of a command, we can open a file directly.

Make the following changes to input.pl:

```
#!/usr/local/bin/perl

open(FILE,"/etc/protocols");

while(<FILE>){
  if(/ICMP/){
     print;
  }
}

close(FILE);
```

Save and test this script to make sure you get the same output as before. The **open** function creates a *filehandle* from which to read. It is used just like STDIN was used before. We need to **close** the file when we are done with it.

If we do not want to keep the file open the whole time, we can store the contents in an array and close the file before doing any work on its contents.

Observe the following:

```
#!/usr/local/bin/perl

open(FILE,"/etc/protocols");
@lines = <FILE>;
close(FILE);

foreach $line (@lines){
  if($line =~ /ICMP/){
     print $line;
  }
}
```

We will discuss later how we can open files for writing; either by creating a new file or appending to an existing one.

Data Manipulation

Split

Recall that we used **awk** to break up the fields of the password file. Perl uses the **split** command to let us to the same thing. Let us edit a new script called *passwd.pl*.

Add the following code to passwd.pl:

```
#!/usr/local/bin/perl

open(FILE,"/etc/passwd");
@lines = <FILE>;
close(FILE);

foreach $line (@lines){
   @fields = split(/:/,$line);
   print "$fields[0]\n";
}
```

Just like before, we are reading in a file (*/etc/passwd*) and putting the contents into an array. Then we use **foreach** to perform an operation on each line of the array.

The **split** command takes a string, in this case **$line**, and divides it into an array of strings. It separates the elements by whatever pattern is given as its first argument. Each time through the loop, **@fields** contains data for the next line.

Save and test the above script.

Push

Currently, *passwd.pl* prints out a list of usernames as they appear in the */etc/passwd* file. But we want an alphabetized list instead. We know we can easily **sort** a list, that is not the issue. Somehow we have to create a list of usernames that can be sorted later. One way to do this would require a variable to "increment" each time through the **foreach** loop, like so:

Observe the following:

```
$i=0;
foreach $line (@lines){
   @fields = split(/:/,$line);
   print "$fields[0]\n";
   $users[$i] = $fields[0];
   $i++;
}
```

That seems like an awful lot of work just to add a new element into an array. It would be nice if we could just **push** something onto an array.

Add the following code to passwd.pl:

```
#!/usr/local/bin/perl

open(FILE,"/etc/passwd");
@lines = <FILE>;
close(FILE);

foreach $line (@lines){
   @fields = split(/:/,$line);
   push(@users,$fields[0]);
}
```

The **push** function adds a list to the end of an array. Instead of a list, we are just adding a single element. The only thing left to do is to print out the sorted list.

Add the following code to passwd.pl:

```
#!/usr/local/bin/perl

open(FILE,"/etc/passwd");
@lines = <FILE>;
close(FILE);

foreach $line (@lines){
   @fields = split(/:/,$line);
   push(@users,$fields[0]);
}

foreach $user (sort(@users)){
  print "$user\n";
}
```

When you test this script, you will see that the usernames are now printed out in alphabetical order.

Note The **push** function has a counterpart called **unshift** which does exactly the same thing except the new elements are added to the beginning of the array instead of to the end.

Reverse and Join

Pretend for a moment that we need a printout of the password file with its entries reversed; one where all of the fields are in the opposite order. We know for a fact that there are seven fields in the password file so we could reverse them by hand.

Add the following code to passwd.pl:

```
#!/usr/local/bin/perl

open(FILE,"/etc/passwd");
@lines = <FILE>;
close(FILE);

foreach $line (@lines){
   @fields = split(/:/,$line);
   push(@users,$fields[0]);
   print "$fields[6]:$fields[5]:$fields[4]:$fields[3]:$fields[2]:$fields[1]:$fields[0]\
n";
}

foreach $user (sort(@users)){
  print "$user\n";
}
```

This would work, but it is really a rather rough way of doing it. As always with Perl, there's probably a better way.

Add the following code to passwd.pl:

```
#!/usr/local/bin/perl

open(FILE,"/etc/passwd");
@lines = <FILE>;
close(FILE);

foreach $line (@lines){
   @fields = split(/:/,$line);
   push(@users,$fields[0]);
   @rfields = reverse(@fields);
   $newline = join(":",@rfields);
   print "$newline\n";
}

foreach $user (sort(@users)){
  print "$user\n";
}
```

The **reverse** function does exactly what its name implies. It takes an array, reverses the elements, and outputs a new array. We could still use a large print statement to output the new lines, but that's an awful lot of work. What if we were not dealing with a known number of fields? Fortunately, Perl gives us **split** to break up the fields and it gives us **join** to put them back together. To use **join**, all we have to do is specify a string to use as a separator and a list of elements to join. Be sure and save this script before trying it out.

After the command prompt, type the following commands:

```
hottub:~$ ./passwd.pl |head
/bin/bash
:/root:root:0:0:x:root

:/bin:bin:1:1:x:bin

:/sbin:daemon:2:2:x:daemon

:/var/adm:adm:4:3:x:adm

:/var/spool/lpd:lp:7:4:x:lp
```

Whoa! Hold on a second. I thought we were supposed to have reversed lines, so why is it all broken up like this? Let us examine what is happening a little closer by looking at the first line of */etc/passwd*.

```
root:x:0:0:root:/root:/bin/bash
```

This is the whole line, correct? Well, not really. The key to understanding this problem is realizing that the line has a newline character at the end of it.

```
root:x:0:0:root:/root:/bin/bash\n
```

This is what the line actually looks like. Notice that we have that newline character at the end.

```
                 split
root   x   0   0   root   /root   /bin/bash\n
```

Our line is split up at all of the colons, giving us seven separate elements. Then we reverse the order of the elements.

```
                   reverse
/bin/bash\n   /root   root   0   0   x   root
```

Notice that the newline remains with the last field.

```
               join
/bin/bash\n:/root:root:0:0:x:root
```

After rejoining the line back together with colons, we can see how the newline is embedded into the new line. When we print this line out, the newline causes a line break even though we might not have intended it.

Chomp

This trailing newline problem is so common that Perl has a special function specifically designed to get rid of it when it occurs. There is a function called **chop** that removes the last character from all of the elements in a list. That's good, but we could accidentally remove characters that are not newlines. Fortunately we have the **chomp** function. It removes all of the newlines from each item in a list (or nothing if there is not a newline). Let us check this out in action.

Add the following code to passwd.pl:

```
#!/usr/local/bin/perl

open(FILE,"/etc/passwd");
@lines = <FILE>;
close(FILE);

foreach $line (@lines){
   @fields = split(/:/,$line);
   chomp(@fields);
   push(@users,$fields[0]);
   @rfields = reverse(@fields);
   $newline = join(":",@rfields);
   print "$newline\n";
}

foreach $user (sort(@users)){
  print "$user\n";
}
```

Save this script and try it out. Notice that this time we get the results we expected to get the first time.

Instead of giving **chomp** a whole list, we could have just given it the last field (which we know had the newline).

OBSERVE:

```
chomp($fields[$#fields]);
```

Since we hsve removed the newline, we need to re-add it before printing it out. It is simple enough just to add on the newline as part of the **print** statement.

Note Some of the entries in */etc/passwd* do not have a shell listed. If we were to remove the newline from **$line** before using **split**, we would lose the last field. Since nothing would exist after the last colon, an element is not created. Thus, when we use **join** the extra colon is lost. Try it out and see. See you at the next lesson!

Fun with Regular Expressions

Regular Expressions

So up until now we have claimed that Perl makes text manipulation easy. Regular expressions are what gives Perl a lot of its power. Unfortunately, working with regular expressions is not easy. Simple word matching patterns aren't difficult, but regular expressions can get very complex. For this reason, we're going to spend some extra time on regular expressions.

Unix regular expressions and Perl regular expressions aren't identical. They have very similar, but a lot of the things that can be done with Perl will not work on the Unix command line.

Quick Review

Let us briefly go over what we already know about regular expressions. We know that matching a simple string is fairly easy. We just include it inside of a pair of forward slashes.

/string/

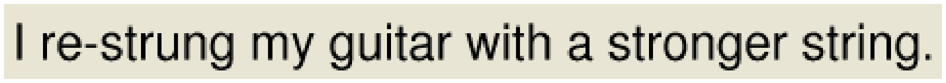
A group of characters can be matched by using brackets. For example

/[abc]/

will match **a**, **b**, or **c**. We can combine these to come up with something like this:

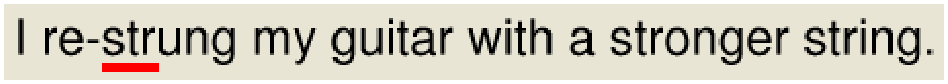
/str[io]ng/

Let's look carefully at the way Perl would use this regular expression to match a string. Say we're searching for this pattern in the string below:



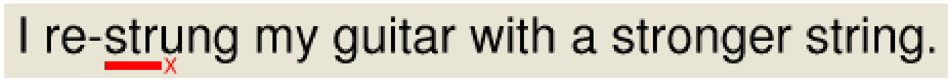
I re-strung my guitar with a stronger string.

We start at the first character and continue searching until we match the first thing in the pattern.



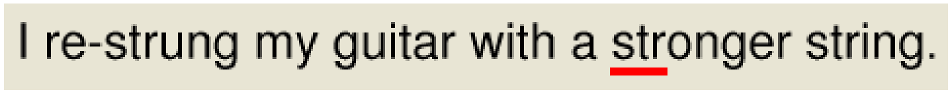
I re-strung my guitar with a stronger string.

After we see an **S**, we look for **t** and **r**. If both of those are found, we try to match the group **[io]**.



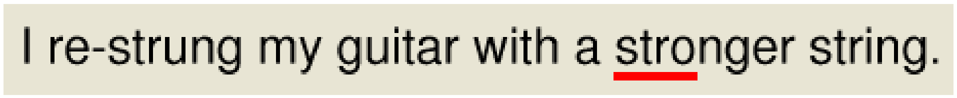
I re-strung my guitar with a stronger string.

The "u" is not an **i** or an **o** so we stop matching. We continue through the string looking for the beginning of our pattern again.



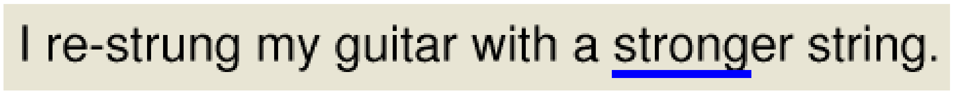
I re-strung my guitar with a stronger string.

Here it is. Does the **[io]** group match?



I re-strung my guitar with a stronger string.

It sure does. So now all we have to do is check for **ng**, if that matches we are done.



I re-strung my guitar with a stronger string.

We have found our match! We can stop searching for the pattern now.

We can even match whole classes of data by using a dash.

/[A-Za-z]tr[a-z]ng[0-9]/

Multipliers

Multipliers allow us to match part of a pattern more than once. For instance, what if I wanted to match "Ned" and "need" with the same expression? Somehow I have to match multiple **e**'s.

/[Nn]e+d/

The plus sign (**+**) stands for *one or more*. This expression will match an upper or lowercase **n**, followed by one or more **e**'s and a **d**. But what if we needed to match "Nd" as well?

/[Nn]e*d/

The star (*****) works in the same way except that it stands for *zero or more*. The last simple multiplier we should talk about is the question mark (**?**). The question mark lets us match *zero or one* of something. It is kind of like asking, "Is there one of these?"

/ab?c/

This would match **abc** and **ac**. Another way to use multipliers is to specify the minimum and maximum values directly using braces. There are three main ways to define a multiplier using braces. First, a single number in braces, such as **{4}**, means it will match exactly 4 times. If we follow that up with a comma, as in **{4,}**, it would match 4 or more times. The third and final way would be to specify a maximum value. To match 4 to 6 times we would use **{4,6}**. Let us see how this method can be used to make equivalents to the simple multipliers we already talked about.

+ can be represented by **{1,}**
***** can be represented by **{0,}**
? can be represented by **{0,1}**

Special Characters

If there is one thing Perl has a lot of, it is special characters. The **+**, *****, and **?** from above are examples of special characters (characters that have special meanings depending on how they are used).

Two very useful special characters are the karat (**^**) and the dollar sign (**$**). When used in a regular expression, the **^** symbol will match the beginning of a line and **$** matches the end. For example, if we wanted to match all lines that contain "The" at the beginning we would use this:

/^The/

Matching the end of a line works much the same way.

/the end$/

$** matches before any newline characters so you do not have to worry about those getting in the way. Let us not forget that these two are special characters. Remember that **$ is used when we reference variables as well. The **^** also has another use. If it is included at the beginning of a character group with the brackets, it not match the list. Let us look at an example:

/str[^io]ng/

In this case, the grouping will match anything *but* **i** and **o**. Another special character is the period (**.**). We have already seen that it can be used as the concatenation operator for strings. But inside of a regular expression it is a whole different story. A period represents *any character*.

/str.ng/

This expression will match "string", "strong", "str9ng", etc. The only thing a period will not match is a newline. Periods are often used in conjunction with multipliers.

/^The.*nice/

Here we are matching any line that starts with "The" and contains any number of characters followed by "nice". That is all fine and good, but you are probably wondering how we would match an actual period in the text.

The Escape Character

The *escape character*, represented by a backslash (\), indicates to Perl that the character right after it should be treated differently than it normally would. This is used for two main purposes. First, it can make special characters have no special meaning or it can make ordinary characters take on a new meaning.

Say we wanted to match a period inside of our text. Let us take it a little further and try to match a dollar amount. The decimal point and the dollar sign are both special characters.

/\$[0-9]*\.[0-9]*/

The escape character has been used here to disregard the normal meaning of the dollar sign and decimal point. Can you spot any problems with the expression? Is it really useful? Are there any cases where the expression would not match a dollar amount? Aha! This expression will not match things like $1,000.00 and $5. Here is a better way (changes are in **blue**):

/\$[,0-9]*\.?[0-9]{0,2}/

The escape character functions with quotes, semi-colons, at symbols (@), etc. As mentioned, the escape character not only removes the special meaning from some symbols, it also adds meaning to others. Below is a table of the most commonly used of these:

Character	Matches
\w	alphanumeric, including underscore (_)
\s	whitespace
\d	numeric
\n	a newline
\W	non-alphanumeric
\S	non-whitespace
\D	non-digit

These special characters are used in a regular expressions just like any other character or group of characters. You are doing great! See you at the next lesson.

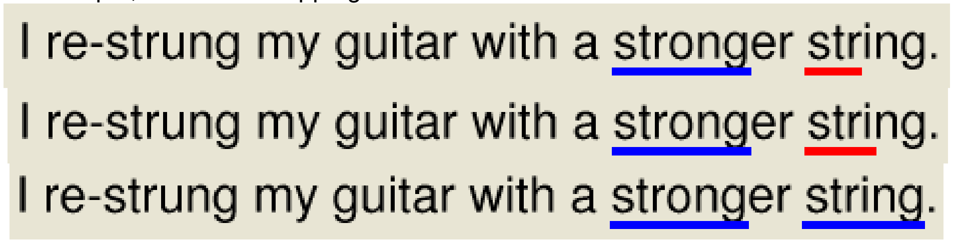
More Fun with Regular Expressions

Regular Expression Commands

Remember at the beginning of this course we were working with **sed** and we used **g** to force a pattern to match as many times as possible? Perl lets us do the same thing with the same command.

/str[io]ng/g

Continuing with our example, instead of stopping after the first match we can continue until the end of the string.



I re-strung my guitar with a stronger string.

I re-strung my guitar with a stronger string.

I re-strung my guitar with a stronger string.

Perl, like Unix, is case sensitive. This means that it treats upper and lower case letters differently. This can cause problems if we are looking for a specific word, but we do not know if it is at the beginning of a sentence or maybe even in all upper case letters. We could use character groupings like this:

/[Ww][Oo][Rr][Dd]/

That is a little ridiculous, though, don't you think? Instead we can use a regular expression command.

/word/i

That is a lot easier to look at. The **i** tells Perl to be case insensitive.

Search and Replace

Alright, so we know how to match patterns in strings which helps us identify lines and gather information. But how can we change those lines quickly and easily? Search and replace with Perl looks a lot like it did with Sed. Using the **s** command we can give Perl a pattern to look for and something to replace that pattern. Imagine for a moment that we need to change some of our users' default shells. A lot of them are using **sbin** but we have renamed it **sysbin**.

Add the following code to passwd.pl:

```
#!/usr/local/bin/perl

$oldfile = "/etc/passwd";
$newfile = "/users/username/passwd";

open(FILE,"$oldfile");
open(NEW,">$newfile");

while(<FILE>){
  $_ =~ s/sbin/sysbin/;
  print NEW;
}

close(FILE);
close(NEW);
```

Most of this program works by setting up the files and reading in the lines that we need to modify. The search and replace is done on a single line. That is why regular expressions are so powerful. However, this is not the best expression we could have used. Keep in mind that Perl will replace the first instance of "sbin" that it finds. What if one of the user names was "sbinder?" We would have a problem then. The following regular expression would be a little better:

s/\/sbin\//\/sysbin/;

Nobody is going to have a login name of "/sbin/", that is for sure. Regular expressions are powerful, but they can also be dangerous. For those having a difficult time reading the above regex here is a reader-friendly view:

s/ \/sbin\/ / \/sysbin /; #We are escaping the slashes of "/sbin/" and "/sysbin"

Translate

The translation command, **tr**, looks like search and replace when you first see it, but it acts a lot differently. Instead of replacing an entire pattern with something else, translation takes a list of characters and replaces them with corresponding characters from a replacement list. Let us write a script called *translate.pl* that converts all of the upper case letters from input into lower case letters.

Add the following code to translate.pl:

```
#!/usr/local/bin/perl

while(<STDIN>){
  $_ =~ tr/[A-Z]/[a-z]/;
  print;
}
```

Save this script so we can test it out.

After the command prompt, type the following commands:

```
cold:~$ chmod 755 translate.pl
cold:~$ echo "ABC" |./translate.pl
abc
```

Here we have translated entire groupings of characters. We do not have to use only single letters. Here are more possible translations.

tr/123/abc/
tr/\'/\"/

Not-so Regular Expressions

So far we have learned how to match patterns in strings and even replace those patterns with other strings. We know we can pull parts of the string into other variables if we use functions like **split**. However, sometimes the string is not going to be divided up nicely by spaces or colons. If this is the case, how can we grab parts out of the string easily? The answer: parentheses.

Parentheses can be used as part of a pattern to store a match in a temporary variable. Create a file called *grab.pl*.

Add the following code to grab.pl:

```
#!/usr/local/bin/perl

while(<STDIN>){
  if(/str([io])ng/){
    $vowel = $1;
    print "We matched $vowel\n";
  }
}
```

This script tests each line of input for the given regular expression. The difference is that when there is a match, the vowel that was part of the match gets stored in **$1**.

After the command prompt, type the following commands:

```
cold:~$ ./grab.pl
string
We matched i
Ctrl+c
```

The entire word "string" was matched, but only the "i" was stored in a variable. Try it again with "strong" and see what happens. We can use as many sets of parenthesis as we want and the variables will be named in numerical order.

Perl allows us to use the stored matches in the same regular expression in which it was matched. This is done by using the escape character and is known as a *back reference*.

Make the following changes to grab.pl:

```
#!/usr/local/bin/perl

while(<STDIN>){
  if(/str([io])ng\1/){
    $vowel = $1;
    print "We matched $vowel\n";
  }
}
```

We have only added one thing, but let us see how it changes the functionality of the script.

After the command prompt, type the following commands:

```
cold:~$ ./grab.pl
stringi
We matched i
strongo
We matched o
stringo
Ctrl+c
```

Search and Replace with Back References

When doing a search and replace, we can include part (or all) of the search pattern in the replace string, or even modify the matched string before using it as a replacement. Let's write a script that will switch the first two fields of colon delimited lines. Copy *translate.pl* to *switch.pl* and make the following changes:

Make the following changes to switch.pl:

```
#!/usr/local/bin/perl

while(<STDIN>){
  $_ =~ s/^([^:]*:)([^:]*:)/\2\1/;
  print;
}
```

Let's save this and make sure it works before we try to figure it out.

After the command prompt, type the following commands:

```
cold:~$ tail /etc/passwd |./switch.pl
...
x:username:507:507::/users/username:/bin/bash
```

That regular expression sure looks complicated, but the trick is to break it down into pieces. Here we have the whole thing:

s/^([^:]*:)([^:]*:)/\2\1/

The first **s** indicates that this is a search and replace operation. That gives us two parts to deal with: the pattern and the replacement string.

^([^:]*:)([^:]*:)

Examining the pattern more closely, we can see three main parts. The first karat matches the beginning of the string. Now we've got two parts that look exactly the same.

([^:]*:)

This expression matches zero or more characters that aren't colons followed by a single colon. Then the match is stored in a temporary variable. In this case, we do not want to use a period to match all characters because it would have matched colons too. That is not so bad. Now let us look at the replacement string.

\2\1

This is simply two back references to temporary variables. We've reversed the order to switch them around. Let us put it all back together again.

s/^([^:]*:)([^:]*:)/\2\1/

Do you see how we are making two matches and reversing them?

Evaluation of the Replacement String

Sometimes it is not enough just to use a back reference as part of the replacement string. We may want to grab the matched string, perform some operations on it, and then use it as a replacement. Here is one method of doing that:

Observe the following:

```
#!/usr/local/bin/perl

while(<STDIN>){
  if(/([ts]?he)/){
    $new = ucfirst($1);
    $_ =~ s/$1/$new/;
    print;
  }
}
```

The **ucfirst** function takes a string and capitalizes the first character. This script finds a line that contains "the", "she", or "he". Then it capitalizes the first letter and replaces the original match with the new and improved one.

Of course, there is an easier way. We do not have to do so much work. The **e** command will evaluate the replacement string as normal Perl code before using it as a replacement. Here is a better way to do the previous example:

Observe the following:

```
#!/usr/local/bin/perl

while(<STDIN>){
  $_ =~ s/([ts]?he)/ucfirst($1)/e;
  print;
}
```

Note that when the replacement string is evaluated we can not use the escape character back reference. We have to use the temporary variable name.

Wrap Up

The trickiest part about regular expressions is taking a problem and figuring out how to apply a regular expression to help solve it. This is where lots of practice comes in handy.

Functions

Functions

We use *functions* in our scripts to perform specific tasks. We have used lots of them already, including **print**, **split** and **sort**. These are each pre-defined methods for doing specific tasks. We could write the code ourselves using other parts of Perl, but we choose to use these functions because they help to clean up the overall look of our script (that and it's easier than writing our own).

Sometimes we have scripts that use the same section of code over and over. Other times we may have a huge piece of code that we only use once, but it still manages to clutter up our script. In these cases (and a few others) we may want to define our own functions or subroutines. Let's start a new script named **function.pl**:

CODE TO TYPE:

```
#!/usr/local/bin/perl

open(PASSWD,"/etc/passwd");
@passwd = <PASSWD>;
close(PASSWD);

foreach $line (@passwd){
  $line =~ s/^([^:]*:)([^:]*:)/\2\1/;
  $line =~ tr/[A-Z]/[a-z]/;
  @fields = split(/:/,$line);
  chomp(@fields);
  @rfields = reverse(@fields);
  $newline = join(":",@rfields);
  print "$newline\n";
}
```

This script takes lines from the */etc/passwd* file and displays them scrambled in the console. It is not a particularly useful script, but we are going to play with it so we can experiment with the functions. Let's remove the guts of this script into a separate Perl function.

CODE TO TYPE:

```
#!/usr/local/bin/perl

open(PASSWD,"/etc/passwd");
@passwd = <PASSWD>;
close(PASSWD);

foreach $line (@passwd){
  $newline = messup($line);
  print "$newline\n";
}

sub messup {
  my $line = shift(@_);

  $line =~ s/^([^:]*:)([^:]*:)/\2\1/;
  $line =~ tr/[A-Z]/[a-z]/;
  @fields = split(/:/,$line);
  chomp(@fields);
  @rfields = reverse(@fields);
  $newline = join(":",@rfields);

  return $newline;
}
```

The top part of the script is unchanged except that we have replaced most of the code with this line:

$newline = messup($line);

The new **messup** function takes a line and converts it into a new, messed up line. At the bottom of the script we have defined the function. You can see all of the lines that we removed from our original script, but there are a few new parts now.

sub messup{

This line defines the beginning of our function (subroutine). There is a matching closing brace (}) at the end. The next line might be the most important part:

my $line = shift(@_);

Functions would not be nearly as useful if they could not receive input and return results. When a function is invoked, the arguments that are passed to it are stored in a list called **@_**. It works sort of like the temporary variable **$_**, but it works like an array instead of a single variable. The **shift** function returns the first element from the list, which in this case is the only variable we gave as an argument. Next we assign this value to **$line**. The addition of **my** makes the variable private to the subroutine so it will not get confused with other variables of the same name found in our main script.

return $newline;

This returns our new line back to the main script.

But let's get back to functions making programming easier—so far, this new function just seems like a lot of extra work. That's true at first, but we can perform the same operation again with very little extra effort. Make the following change to **function.pl**:

CODE TO TYPE:

```
#!/usr/local/bin/perl

open(PASSWD,"/etc/passwd");
@passwd = <PASSWD>;
close(PASSWD);

foreach $line (@passwd){
  $newline = messup($line);
  $newline = messup($newline);
  print "$newline\n";
}

sub messup {
  my $line = shift(@_);

  $line =~ s/^([^:]*:)([^:]*:)/\2\1/;
  $line =~ tr/[A-Z]/[a-z]/;
  @fields = split(/:/,$line);
  chomp(@fields);
  @rfields = reverse(@fields);
  $newline = join(":",@rfields);

  return $newline;
}
```

Here we performed the same set of operations a second time. All we had to do was add an additional call to the function instead of writing out all of that code again. Save and test this script to see the results.

Libraries

Another benefit to writing functions is that we can reuse them in future scripts. We just copy them into new scripts or put them into a *library*. A library is a set of pre-written functions that are set aside in a separate file for use in other scripts. Let's copy **messup** from **function.pl** into a library. To remove the code ~~marked for deletion~~, highlight it and press [**Ctrl+x**].

Make the following changes to function.pl:

```
#!/usr/local/bin/perl

require 'mess-lib.pl';

open(PASSWD,"/etc/passwd");
@passwd = <PASSWD>;
close(PASSWD);

foreach $line (@passwd){
  $newline = messup($line);
  $newline = messup($newline);
  print "$newline\n";
}

~~sub messup{~~
~~  my $line = shift(@_);~~

~~  $line =~ s/^([^:]*:)([^:]*:)/\2\1/;~~
~~  $line =~ tr/[A-Z]/[a-z]/;~~
~~  @fields = split(/:/,$line);~~
~~  chomp(@fields);~~
~~  @rfields = reverse(@fields);~~
~~  $newline = join(":",@rfields);~~

~~  return $newline;~~
~~}~~
```

The new **require** line tells Perl to include the contents of **mess-lib.pl** (which we'll create in a moment) when it runs. We could write lots of other scripts that use the **mess-lib.pl** library as well. All we need to include is the **require** line. Now, create the **mess-lib.pl** file. The name of this file doesn't really matter (as long as our **require** statement above matches it), but it is helpful if the name indicates what the file contains—but you knew that. Add the following code to mess-lib.pl (first, press [**Ctrl+v**] to insert the code we removed from function.pl):

CODE TO TYPE:

```
sub messup{
  my $line = shift(@_);

  $line =~ s/^([^:]*:)([^:]*:)/\2\1/;
  $line =~ tr/[A-Z]/[a-z]/;
  @fields = split(/:/,$line);
  chomp(@fields);
  @rfields = reverse(@fields);
  $newline = join(":",@rfields);

  return $newline;
}

1;
```

This is the exact same function we worked on before with the exception of that last line. Perl library files must return "true", so every library file needs to end with a **1;**. Save this file.

Also, if we have to change how **messup** works, we would only have to change one file instead of all of our scripts. Good job! Let's move on to the next lesson.

Directories and Files

Reading Directories

In order to perform operations on a directory full of files, it becomes necessary to be able to list the directory's contents. To do this we would use **ls** on the command line. We could do the same in Perl by using back ticks to capture the output. But it would be better to use a few of Perl's built-in functions to read the directory. Write a script called *list.pl* that will list the contents of the current directory.

Add the following code into list.pl:

```
#!/usr/bin/perl

opendir(CURRENT,".");
@list = readdir(CURRENT);
closedir(CURRENT);

foreach $item (@list){
  print "$item\n";
}
```

These three new functions are fairly self-explanatory. We open the directory, read its contents into an array, and finally, close the directory when we are finished. Save this script and test it.

After the command prompt, type the following commands:

```
cold:~$ ./list.pl
.
..
cgi
.emacs
html_css
.ncftp
.bash_login
.bash_history
.error_log
linux4
.ssh
```

There are a couple things to notice about this output. First, the contents are not listed alphabetically. They are listed in the order that they were created. In most cases you do not really need an alphabetical list so there is no point in Perl wasting the time to do it. The second thing to note is that dot files show up automatically. Let us change our script so it does not list files that begin with a dot.

Make the following changes to list.pl:

```
#!/usr/bin/perl

opendir(CURRENT,".");
@list = readdir(CURRENT);
closedir(CURRENT);

foreach $item (@list){
  if($item !~ /^\./){
    print "$item\n";
  }
}
```

Instead of simply printing out the file names, we could open the files and perform operations on them.

File Tests

Once we have a filename, either by supplying it ourselves or reading it from a directory, we can use it to do a number of tests. *File test operators* look a lot like Unix command line options and are often used within conditional statements.

Create a script called *filetest.pl*.

Add the following code to filetest.pl:

```
#!/usr/bin/perl

$username = "yourusername";
$filename = "/users/".$username."/.bash_login";

if(-e $filename){
  open(FILE,"$filename");
  while(<FILE>){ print };
  close(FILE);
}else{
  print "Error:  $filename doesn't exist\n";
}
```

Replace "username" with your username when creating this script. Save and test the script. If you have a *.bash_login* file, the script will display its contents, otherwise you will get an error saying the file does not exist.

Another method enables us to make sure the file has a non-zero size. Using the **-s** file test operator instead of **-e**, we can check for a file and determine its size at the same time.

Make the following changes to filetest.pl

```
#!/usr/bin/perl

$username = "yourusername";
$filename = "/users/".$username."/.bash_login";

if($size = -s $filename){
  print "$filename has a size of $size\n";
}else{
  print "Error:  $filename doesn't exist or has zero size.\n";
}
```

Change the **$filename** variable to a file that you know exists. Save and test this script. Additionally, the **-z** can be used to test for a file that exists and has zero size.

We can test to see if a file exists and find out its size, and we can also use file tests to determine a lot of additional information. Here is a table of a few of the different file test operators available.

File Tests	Description
-e	Exists
-s	Non-zero size (returns size)
-z	Zero size
-r	Readable by effective uid/gid
-w	Writable by effective uid/gid
-x	Executable by effective uid/gid
-o	Owned by effective uid/gid
-f	Plain file
-d	Directory
-l	Symbolic Link
-M	Age of file (in days) since last modification

The *effective* user or group id is typically going to be the uid/gid of the user executing the script.

File Manipulation

We now know quite a bit about gathering information on a file. Now let us learn how to modify things. We can change the permissions using the **chmod** function.

Observe the following:

```
chmod(0755,$filename);
```

This works just the way you would expect, except for one important difference. The numerical permissions must start with a zero in order for them to be interpreted correctly.

A file can be renamed with a function called, logically enough, **rename**. Using it is pretty simple.

Observe the following:

```
rename($oldname,$newname);
```

Not everything is named quite so intuitively though. For instance, to delete a file we use **unlink**. Creating hard links and symbolic links is done with **link** and **symlink** respectively.

There seems to be something missing here though. How would we copy a file? There is not a command specifically for this purpose built into Perl. This is one of many cases where you find it might be easier to use a simple Unix command instead. Fortunately this is not a problem.

Running System Commands

Perl provides two ways to run a Unix command directly. (These are in addition to using back ticks to capture the output of a command in a variable.) The first method is the **system** function. We simply give **system** a string or list to run as a Unix command.

Observe the following examples:

```
system("cp $filename $newfilename");

system("find",@ARGV);
```

When **system** executes its command, the Perl script will wait for it to finish before continuing. This is necessary when the results of your system command are needed by the rest of the script. In case you do not want to wait for the external command to finish, you can always background it with an ampersand (&) as part of the command string.

A second, similar method to use is the **exec** function. The main difference between **exec** and **system** is that **exec** will terminate the originating Perl script and replace it with the new command. Therefore, it is only useful at the end of a script, typically for running a shell or something like that.

WARNING Reusing user input on the command line with a system call can be dangerous in cases when the Perl script is running as a different user than the person executing it (CGI scripts and suid scripts). For example, if you grab user input and they input "; some command" as an argument, the semicolon would represent the end of the first command. This would allow the second arbitrary command to run on the system. User input should always be parsed for special characters with a regular expression before being used in a system call.

Recursive Directory Search

Recursion

You have probably heard of "recursion" even if you are not entirely familiar with the concept. The idea behind *recursion* is to focus on a problem, divide it into increasingly small problems of the same type and find solutions for the smaller problems. At this point, you can back up and see that the entire big problem has been solved.

In Perl, a recursive sub-routine would call itself over and over as it makes progress towards a *base case*. The base case is the smallest part of the problem with a "known" solution. Recursion can be a difficult concept to get your mind around at first, so let us look at an example.

Recursive Directory Search

Let us say we want to get a list of all the directories underneath a given starting point. Essentially we want the same results that **find ./ -type d** would return, but we sre going to build a Perl script ourselves to make the output look better.

Where do we start? Well, before we write any code at all, let's make a plan. We know how to open a directory and read its contents. Using file tests we can also find the name of other directories within the first one. Once we get a list of those directories we can list their contents as well. The problem is that we do not know how far this is going to go. But do you see the pattern? Even if we are five directories deep, we are doing the same thing: reading a directory and getting a list of other directories within it.

Let's write a function that does just that. Create a script file called *rd.pl*.

Add the following code to rd.pl:

```
#!/usr/local/bin/perl

$startdir = "/lib";

list_dirs($startdir);

sub list_dirs{
  my $dir = shift(@_);

  opendir(TOP,$dir);
  my @files = readdir(TOP);
  closedir(TOP);

  foreach $file (@files){
    if(-d "$dir/$file"){
        print "$file\n";
    }
  }
}
```

The **list_dirs** function reads the directory it is given to get a list of files. Then it tests those files to see which ones are directories. Notice that when this happens we have to include the first directory as part of the test (**-d "$dir/$file"**) so that the path is correct. Save and test this script:

After the command prompt, type the following commands:

```
cold:~$ ./rd.pl
kbd
security
rtkaio
modules
terminfo
.
i686
..
udev
firmware
```

Excellent. Now the idea here is to take these new directories and do the same thing over and over again until there are no more directories left. There is one problem, however. What happens if we run our function on the **.** and **..** directories? These are the current and previous directories. If our script is allowed to list the contents of these directories as well, it will never end. It will keep going back and forth while the problem grows ever larger. To prevent this, we prevent **.** and **..** from being added to the list of new directories.

Note Remember, if you find yourself in an infinite loop, you can quit your process by hitting Ctrl+c at the command line.

Removing . and ..

Once we have listed all of the files, we will sort them so we end up with **.** and **..** at the top of our array. It is simple enough to remove them before doing any more processing.

Make the following changes to rd.pl:

```
#!/usr/local/bin/perl

$startdir = "/lib";

list_dirs($startdir);

sub list_dirs{
  my $dir = shift(@_);

  opendir(TOP,$dir);
  my @files = readdir(TOP);
  closedir(TOP);

  @files = sort(@files);
  shift(@files);
  shift(@files);

  foreach $file (@files){
    if(-d "$dir/$file"){
        print "$file\n";
    }
  }
}
```

That is all there is to it. We no longer have to worry about those two special directories. Save and test the script to make sure they are gone.

Making the Search Recursive

Our script lists all of the directories in the starting directory. But that is only the first step. It seems like it would be much harder to make our script keep searching all the way down through each of the directories. But wait a second, the **list_dirs** function will find directories in any directory we give it. So if we pass it one of the newly found directories, it will continue on as if that is the new starting point. And, as it turns out, this can be done with a single line of code.

Make the following changes to rd.pl:

```
#!/usr/local/bin/perl

$startdir = "/lib";

list_dirs($startdir);

sub list_dirs{
  my $dir = shift(@_);

  opendir(TOP,$dir);
  my @files = readdir(TOP);
  closedir(TOP);

  @files = sort(@files);
  shift(@files);
  shift(@files);

  foreach $file (@files){
    if(-d "$dir/$file"){
        print "$file\n";
        list_dirs("$dir/$file");
    }
  }
}
```

All we have done is add a line that continues the search down the next branch of the directory tree. In the next branch, the same thing will happen and the search will continue all the way down until there are no more directories.

Save and test the script. You should see a lot of directory names all in a big column. So now we have got the list of directories, but the only problem is that it is really hard to tell what goes where. It would be nice if the subdirectories were indented slightly from their parent directories.

Making the Output Better

Look over and make these changes:

Make the following changes to rd.pl:

```
#!/usr/local/bin/perl

$startdir = "/lib";

$level = 0;

list_dirs($startdir,$level);

sub list_dirs{
  my $dir  = shift (@_);
  my $lev = shift (@_);


  opendir(TOP,$dir);
  my @files = readdir(TOP);
  closedir(TOP);

  @files = sort(@files);
  shift(@files);
  shift(@files);

  foreach $file (@files){
    if(-d "$dir/$file"){
        spaces($lev);
        print "$file\n";
        list_dirs("$dir/$file",$lev+1);
    }
  }

}

sub spaces{
  my($num) = shift(@_);
  for($i=0;$i<$num;$i++){
    print " ";
  }
}
```

Test the script to see the new output. As part of a quiz you will be describing how this works. Good luck!