Today we'll start talking about things that are specific to Linux and other unix-like operating systems.

We'll come back to these concepts again and again, so this is just a quick first pass.
This is a teletype terminal of the type I used back in the 1970s, when all we had was the command line, and We Liked IT!

Linux is still maybe a little more weighted toward the command line than Windows. Looking at the Linux World, you could say that almost anything you can do at the command line can also be done graphically. In the Windows world, I think it's fair to say that almost anything you can do graphically can also be done from the command line.

The two worlds are converging: Linux's graphical interfaces are continually improving, and Windows keeps improving its command-line interface.

We can all appreciate the value of a graphical interface. It's intuitive (if it's well-designed) and its “discoverable”, in that you can browse around a graphical program's menus and find out what the program can do. But what's the value of a command-line interface? If graphical interfaces are good, why do all major operating systems continue to improve their command-line interfaces?
The Command Line:

```
~/demo> ls
clus.pdf     data-for-everybody.1.dat  phase2
cluster.pdf  ForYourEyesOnly.dat     readme.txt
cpuinfo.dat  phase1                   ReadMe.txt
```

Output of the "ls" command, which lists the files in the current directory.

Why should you do things from the command line?:

* In Linux, graphical tools provide a front-end to help you do tasks, but you can do more from the command line.

* There are several sets of graphical tools available for Linux, so if you learn one of them you may find that it's not available on the next computer you use.

* There's no guarantee that a given computer will have graphical tools installed, or even a monitor.

* Text commands are easily reproduced. It's easy to document what you've done, or to tell someone else how to do it, or to automate what you've done.

The answer is that the command line has its own advantage. Here are some of the things that might make someone choose to use the command line under Linux.

The last item is the most important, I think. This is true for all operating systems, not just Linux, and it's why all major operating systems still have a command-line interface and continue to improve it.
There are graphical shells and command-line shells. When you log in to Microsoft Windows, you're using a graphical shell. In that case, you communicate with the shell by pointing and clicking. Today we'll be talking about command-line shells, where you communicate by typing commands.

The diagram in the corner explains why we call it a “shell”. It's the interface between the inner “kernel” of the operating system and the outer world.
**Command-line Shells:**

Command-line shells accept typed commands, parse them and execute them. They also:

- Expand wild-card expressions.
- Usually store a history of previously-typed commands, and provide a way of recalling these.
- Provide a set of built-in functions that supplement (or sometimes replace) the commands provided by the operating system.
- Provide the user with the ability to define abbreviations for commands (aliases).
- Maintain a set of user-defined variables that can be used in command lines (environment variables and shell variables).

....and.....
Command-line Shells (cont'd):

....

• Often have the ability to auto-complete partially typed commands or file names.

• Provide control structures (if/then/else, while/do) that allow the user to write programs in the shell's language (shell scripts).

• Provide the plumbing to connect commands together with pipes and to redirect the input and output of commands.

The two most commonly used shells used on Linux systems are bash and tcsh, which behave somewhat differently. Bash is Richard Stallman's re-implementation of the original Bourne Shell (sh), which he named the "Bourne Again Shell". Tcsh is an enhanced version of csh, which was an early alternative to the Bourne Shell.
A Few Useful Linux Commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ls</strong></td>
<td>List the contents of a directory.</td>
</tr>
<tr>
<td><strong>pwd</strong></td>
<td>Show the name of the current directory.</td>
</tr>
<tr>
<td><strong>cd</strong></td>
<td>Change the current directory.</td>
</tr>
<tr>
<td><strong>less or more</strong></td>
<td>Show the contents of a file, one page at a time.</td>
</tr>
<tr>
<td><strong>cp</strong></td>
<td>Copy a file.</td>
</tr>
<tr>
<td><strong>mv</strong></td>
<td>Move (rename, relocate or both) a file.</td>
</tr>
<tr>
<td><strong>rm</strong></td>
<td>Delete (remove) a file.</td>
</tr>
<tr>
<td><strong>mkdir</strong></td>
<td>Make a new directory.</td>
</tr>
<tr>
<td><strong>rmdir</strong></td>
<td>Delete (remove) a directory.</td>
</tr>
<tr>
<td><strong>man</strong></td>
<td>Show docs (manual pages) for a command.</td>
</tr>
<tr>
<td><strong>ln</strong></td>
<td>Make a link to a file.</td>
</tr>
<tr>
<td><strong>cat</strong></td>
<td>Spit out the concatenated contents of one or more files, without paging.</td>
</tr>
<tr>
<td><strong>touch</strong></td>
<td>Change the timestamp on a file, or create an empty file.</td>
</tr>
<tr>
<td><strong>which</strong></td>
<td>Find a command in the search path.</td>
</tr>
</tbody>
</table>

This is just a list to get you started.

As you can see, the commands are typically terse.
Command Syntax:

```
$ /demo> ls -l
total 60
lrwxrwxrwx 1 bkwa1 bkwa1 11 Jan 18 11:30 clus.pdf -> cluster.pdf
-rw-r--r-- 1 bkwa1 bkwa1 20601 Jan 18 10:51 cluster.pdf
-rw-r--r-- 1 bkwa1 demo 983 Jan 18 10:53 cpuinfo.dat
-rw-r--r-- 1 bkwa1 bkwa1 29 Jan 18 10:59 data-for-everybody.1.dat
-rw-r--r-- 1 bkwa1 bkwa1 41 Jan 18 10:56 ForYourEyesOnly.dat
drwxr-x-x 3 bkwa1 bkwa1 4096 Jan 18 11:35 phase1
drwxr-x-x 2 bkwa1 bkwa1 4096 Jan 18 10:55 phase2
-rw-r--r-- 1 bkwa1 demo 72 Jan 18 10:52 readme.txt
-rw-r--r-- 1 bkwa1 bkwa1 9552 Jan 18 10:52 ReadMe.txt
```

Linux commands are often modified by the addition of switches or qualifiers like the “-l”, for “long”, switch used in the ls command above. These modifiers will often take one of these forms:

* A dash followed by a letter or number, optionally followed by an argument
  * Two dashes followed by a word, optionally followed by an argument.

For ls, some useful switches are:

- `-l` Gives more information about the files.
- `-T` Combined with `-l`, sorts the files in reverse time order.
- `-S` Combined with `-l`, sorts the files in order of descending size.
- `-a` Lists all files, including hidden files.

Multiple single-letter switches can often be combined, like “ls -IT” instead of “ls -l -T”

In this case, we can change the behavior of the “ls” command by adding the “-l” switch.

Note, though, that Linux commands were all developed independently, and they have a long history. Syntax conventions have evolved over time, and different developers have used different conventions.

We'll see examples of some odd command syntax with commands like “tar” and “ps”.
## Command-Line History:

<table>
<thead>
<tr>
<th>Command</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>~demo&gt; history</td>
<td>16:14</td>
</tr>
<tr>
<td>350 16:15  tar tzvf adobesvg-3.01x88-linux-i386.tar.gz</td>
<td></td>
</tr>
<tr>
<td>351 16:15  tar xzvf adobesvg-3.01x88-linux-i386.tar.gz</td>
<td></td>
</tr>
<tr>
<td>352 16:15  cd adobesvg-3.01</td>
<td></td>
</tr>
<tr>
<td>353 16:15  dir</td>
<td></td>
</tr>
<tr>
<td>354 16:15  cd ..</td>
<td></td>
</tr>
<tr>
<td>355 16:15  rm adobesvg-3.01*</td>
<td></td>
</tr>
<tr>
<td>356 16:15  rm -rf adobesvg-3.01*</td>
<td></td>
</tr>
<tr>
<td>357 16:19  git clone git://people.freedesktop.org/~cworth/svg2pdf</td>
<td></td>
</tr>
<tr>
<td>358 16:19  cd svg2pdf</td>
<td></td>
</tr>
<tr>
<td>359 16:19  dif</td>
<td></td>
</tr>
<tr>
<td>360 16:19  dir</td>
<td></td>
</tr>
<tr>
<td>361 16:19  git pull</td>
<td></td>
</tr>
<tr>
<td>362 16:19  make</td>
<td></td>
</tr>
<tr>
<td>363 16:19  dir</td>
<td></td>
</tr>
<tr>
<td>364 16:20  ./svg2pdf ../drawing.svg</td>
<td></td>
</tr>
<tr>
<td>365 16:20  ./svg2pdf ../drawing.svg junk.pdf</td>
<td></td>
</tr>
<tr>
<td>366 16:20  acroread junk.pdf</td>
<td></td>
</tr>
</tbody>
</table>

The "history" command shows you commands you've recently entered.

You can use the up and down arrow keys to recall previously-typed commands and re-use them. If you know the beginning of a previously-entered command, you can re-run it by entering a "!" followed by the beginning of the command.
The PATH Environment Variable:

```bash
~/demo> echo $PATH
./usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
```

The PATH variable defines a search path for the shell to use when looking for a program. It's composed of a list of directory names, separated by colons. When looking for a program, the shell starts at the left of the list and looks in each directory until it finds a program with the matching name (or fails).

```bash
~/demo> which ls
/bin/ls
```

The "which" command looks through the search path, just as the shell would, and tells you where the shell would find a given program.

With the PATH shown above, a local administrator can add local programs to `usr/local/bin` to make them available to users.

Note that:

1. ".", the current directory, is included. This is not generally the case for users with administrative privileges, for security reasons.

2. By putting an alternative program with the same name in `usr/local/bin`, a local administrator can provide a modified version of a program that overrides any version that might already exist in `/bin` or `/usr/bin`. 
Modifying the PATH Variable:

**In the bash shell:**
```
~/demo> echo $SHELL
/bin/bash
~/demo> export PATH=’/home/bryan/bin:$PATH’
```

**In the tcsh shell:**
```
~/demo> echo $SHELL
/bin/tcsh
~/demo> setenv PATH ’/home/bryan/bin:$PATH’
```
Aliases and Shell Built-in Commands:

```bash
~/demo> which echo
echo: shell built-in command.
```

```bash
~/demo> which rm
rm: aliased to rm -i
```

Creating aliases:

**In the bash shell:**
```
~/demo> alias blarg=ls
```

**In the tcsh shell:**
```
~/demo> alias blarg ls
```

```bash
~/demo> blarg
clus.pdf  data-for-everybody.1.dat  phase2
cluster.pdf  ForYourEyesOnly.dat  readme.txt
cpupinfo.dat  phase1 ReadMe.txt
```
Shell Startup Files:

Both bash and tcsh read a set of startup files when the user logs in. These files can be used to automatically set environment variables (like "PATH"), define aliases, or execute other shell commands.

**For bash:**
Add commands to the file ".bash_profile", in your home directory.

**For tcsh:**
Add commands to the file ".login", in your home directory.

Each of the shells actually looks at different startup files under different circumstances, but the files above are a good place to start.
Plugging Commands Together With Pipes (|):

Linux commands are modular, and can be plugged together easily to do complex things. The output of one command can be sent to the input of another, and so on. (We'll see more of this when we start talking about shell scripts.) There are two common ways of doing this, "pipes" and "backticks":

```bash
~/demo/phase1> ls -l | less  (shows output of "ls" one page at a time)
```

```bash
~/demo/phase2> ls -l
```

```
total 16
drwxr-x--- 2 bkw1a bkw1a 4096 Jan 19 10:41 .
drwxr-x--- 4 bkw1a bkw1a 4096 Jan 19 10:39 ..
-rw-r--r-- 1 bkw1a bkw1a 0 Jan 19 10:41 even_more_junk.txt
-rw-r--r-- 1 bkw1a bkw1a 0 Jan 19 10:41 junk1.txt
-rw-r--r-- 1 bkw1a bkw1a 0 Jan 19 10:41 junk2.txt
-rw-r--r-- 1 bkw1a bkw1a 32 Jan 19 10:38 more_stars.txt
-rw-r--r-- 1 bkw1a bkw1a 18 Jan 19 10:38 some_stars.txt
```

```bash
~/demo/phase2> ls -l | grep stars | sed -e 's/star/STAR/' | awk '{print $3,$NF}'
```

```
bkw1a more_STARS.txt
bkw1a some_STARS.txt
```

Here we've introduced the “less” command, which will show you its input one page at a time. You can use it to view the contents of a file one page at a time by typing, e.g., “less file.txt” or “cat file.txt | less”.

Less is the GNU project's successor to the “more” command found in the original Unix. On most Linux systems, “more” is an alternative name for “less”.

We've also mentioned the grep, sed and awk commands, which we'll talk more about when we discuss scripting. For the example above, grep selects only certain lines from its input, sed modifies its input in a specified way, and awk selects only certain columns of its input.
**Stdin, Stdout, and Stderr:**

- **stdout** Channel through which output is printed by a program.
- **stdin** Channel from which a program obtains input data.
- **stderr** Channel through which a program reports errors.

```
ls -l | grep stars | sed -e 's/st/ST/'
```

Stdin, stdout and stderr are the connectors through which commands are plumbed together.

You can think of each command as a little device that has one input (stdin) and two outputs (stdout and stderr). By plugging outputs and inputs together, you can build up a long pipeline of commands that work together to do a complex task.
Redirecting Output to a File:

\[
\text{ls -l | grep stars > output.dat}
\]

Redirecting stdout into a file

\[
\text{ls -l | grep stars >> output.dat}
\]

Appending lines to an existing file

Redirecting Both Stdout and Stderr to a file:

**Under tcsh:**
\[
\text{ls -l | grep stars >& output.dat}
\]

**Under bash:**
\[
\text{ls -l | grep stars > output.dat 2>&1}
\]
Plugging Commands Together With Backticks (`):

```bash
~/demo/phase2> ls
bad_users.txt   laundry_list.txt   recipes.txt
good_users.txt  random_junk.txt   ugly_users.txt
...et cetera.

~/demo/phase2> ls | grep users
bad_users.txt
good_users.txt
ugly_users.txt

~/demo/phase2> grep Bryan `ls | grep users`
ugly_users.txt:Bryan
```

Commands between backticks are evaluated, then their output is inserted into the command line just as though you'd typed it there yourself, directly.

Here we see another way to use the grep command. If it's given a list of filenames as arguments, it will operate on the content of those files instead of on its input. (In this mode it prints the name of the file in addition to the matching line.)

This is typical of many Linux commands. If given an input file name, they'll work on that file. Otherwise they'll wait for input to be piped into them.
Part 2: Files and Directories:
Listing the Files in the Current Directory:

```
~/demo> ls
clus.pdf  data-for-everybody.1.dat  phase2
cluster.pdf  ForYourEyesOnly.dat  readme.txt
cpinfo.dat  phase1
ReadMe.txt
```

```
~/demo> ls -l
total 60
lrwxrwxrwx  1 bkw1a bkw1a  11 Jan 18 11:39 clus.pdf -> cluster.pdf
-rw-r----  1 bkw1a bkw1a  20601 Jan 18 10:51 cluster.pdf
-rw-r-----  1 bkw1a demo  983 Jan 18 10:53 cpinfo.dat
-rw-r--r--  1 bkw1a bkw1a  29 Jan 18 10:59 data-for-everybody.1.dat
-rw-------  1 bkw1a bkw1a  41 Jan 18 10:56 ForYourEyesOnly.dat
drwxr-x---  3 bkw1a bkw1a  4096 Jan 18 11:35 phase1
drwxr-x--  2 bkw1a bkw1a  4096 Jan 18 10:55 phase2
-rw-r-----  1 bkw1a demo  72 Jan 18 10:52 readme.txt
-rw-r-----  1 bkw1a bkw1a  9552 Jan 18 10:52 ReadMe.txt
```

The first thing a user will probably want to do is look around. We can do this with the “ls” command. Note that “ls” and “ls -l” do different things.

Let's spend some time dissecting the output of the `ls -l` command.
Case-sensitive File Names:

```
$ ls -l
total 60
lrwxrwxrwx 1 bkw1a bkw1a 11 Jan 18 11:30 clus.pdf -> cluster.pdf
-rw-r----- 1 bkw1a bkw1a 20601 Jan 18 10:51 cluster.pdf
-rw-r----- 1 bkw1a demo 983 Jan 18 10:53 cpuinfo.dat
-rw-r--r-- 1 bkw1a bkw1a 29 Jan 18 10:59 data-for-everybody.1.dat
-rw------- 1 bkw1a bkw1a 41 Jan 18 10:56 ForYourEyesOnly.dat
drwxr-x--- 3 bkw1a bkw1a 4096 Jan 18 11:35 phase1
drwxr-x--- 2 bkw1a bkw1a 4096 Jan 18 10:55 phase2
-rw-r----- 1 bkw1a demo 72 Jan 18 10:52 README.txt
-rw-r----- 1 bkw1a bkw1a 9552 Jan 18 10:52 ReadMe.txt
```

Under Linux, files are typically **case-sensitive**. This means that “readme.txt” is a completely different file from “ReadMe.txt”. This is unlike Windows or OS X, where a filename’s case is preserved, but ignored.

We see that Linux file names are usually case-sensitive.

This is a feature of the filesystem, not the operating system, per se. We'll talk about filesystems at another time. Some filesystems used under Linux are not case-sensitive, but the most common ones are.
Here are some other features of the output of our “ls” command.

For now, we'll ignore the third column. This is the number of “hard links”. For files, it will almost always be 1. For directories, it will be 2 + the number of subdirectories inside the directory. We'll talk about “why” later.

The “file size” column reports what you'd expect for files. For directories, though, it reports the the size of the directory excluding its contents. For a directory, this number is the size of all of the “metadata” associated with this directory: the file names, permissions, et cetera. This isn't generally very useful.
Symbolic Links:

```bash
$ /demo> ls -l
 total 60
 lrwxrwxrwx 1 bkw1a bkw1a 11 Jan 18 11:39 clus.pdf -> cluster.pdf
-rw-r----- 1 bkw1a bkw1a 20601 Jan 18 10:51 cluster.pdf
-rw-r----- 1 bkw1a demo 983 Jan 18 10:53 cpuinfo.dat
-rw-r----- 1 bkw1a bkw1a 29 Jan 18 10:59 data-for-everybody.1.dat
-rw------- 1 bkw1a bkw1a 41 Jan 18 10:56 ForYourEyesOnly.dat
drwxr-x-- 3 bkw1a bkw1a 4096 Jan 18 11:35 phase1
drwxr-x-- 2 bkw1a bkw1a 4096 Jan 18 10:55 phase2
-rw-r----- 1 bkw1a demo 72 Jan 18 10:52 readme.txt
-rw-r----- 1 bkw1a bkw1a 9552 Jan 18 10:52 ReadMe.txt
```

Symbolic links are like alternative names for a file or directory. In the example above, “clus.pdf” is a symbolic link pointing to a real file called “cluster.pdf”. You can have as many symbolic links as you like pointing to a given real file. (Symbolic links can even point to other symbolic links, but there’s a limit on how many levels of this you can do.)

To create a symbolic link, use the “ln” command:

```
ln -s RealFile SymlinkFile
```

Be careful of the order! It’s easy to get the file names reversed.

Symbolic links can be a wonderful way to solve otherwise thorny problems. Many Unix administrators, when asked what’s the most useful Unix command, will say “ln -s”, the command used for creating symbolic links.
**Directory files:**

```
~/demo> ls -l
total 68
lrwxrwxrwx 1 bkw1a bkw1a  11 Jan 18 11:39 clus.pdf -> cluster.pdf
-rw-r----- 1 bkw1a bkw1a 20601 Jan 18 10:51 cluster.pdf
-rw-r----- 1 bkw1a demo  983 Jan 18 10:53 cpuinfo.dat
-rw-r--r-- 1 bkw1a bkw1a  29 Jan 18 10:59 data-for-everybody.1.dat
-rw------- 1 bkw1a bkw1a  41 Jan 18 10:56 ForYourEyesOnly.dat
drw-r-x--- 3 bkw1a bkw1a 4096 Jan 18 11:35 phase1
drwxr-x--- 2 bkw1a bkw1a 4096 Jan 18 10:55 phase2
-rw-r----- 1 bkw1a demo  72 Jan 18 10:52 readme.txt
-rw-r----- 1 bkw1a bkw1a 9552 Jan 18 10:52 ReadMe.txt
```

Directories can contain other directories, and these subdirectories show up in the output of "ls -l". In the example above, "phase1" and "phase2" are two subdirectories of the directory we're looking at. Each of these subdirectories may contain its own files and more subdirectories.
The “Current Directory”:

You can see what directory you're currently working in by using the "pwd" command:

```bash
~/>demo> pwd
/home/bryan/demo
```

Note that the path to a file or directory is given as a list of parent directories, separated by slashes, starting with the root directory ("/"). In this case, the current working directory is "/home/bryan/demo".

You can change your current directory by using the "cd" command, like:

```bash
~/>demo> cd phase1
```

Or, equivalently:

```bash
~/>demo> cd /home/bryan/demo/phase1
```

In the first case, we specify the name of a directory relative to the current directory, and in the second case we explicitly give the full path name (the complete name of the directory we're interested in.)

Let's pause for a minute to look at the way directories are laid out on a typical Linux system.

First, there's the concept of a “current directory”.
The “Home Directory”:

Each user has a “home directory”. This directory will be your current directory right after you log in.

```bash
~demo> echo $HOME
/home/bryan
```

You can use the $HOME environment variable in commands, to refer to your home directory.

```bash
~demo> ls $HOME/demo
```

You can also refer to your home directory as “~”, in most shells.

```bash
~demo> ls ~/demo
```

If you just type “cd” by itself, it will take you to your home directory.

$HOME is an “environment variable”. These are similar to the variables used in computer programs. In fact, you can write programs in the shell language, too. These are usually called “scripts” or “shell scripts”. They can be used to automate shell tasks you do often.

The $HOME variable is set for you automatically when you log into the computer. We'll talk later about how modify the values of environment variables, or create new ones of your own.
Here's a graphical representation of a highly simplified Linux directory tree. One of the basic principles of Linux (and other varieties of Unix) is that there's only one directory tree. Everything lives somewhere under the “/” (root) directory.

This is unlike Windows, for example, where each device has a separate directory tree. In Windows we have a directory tree on drive C:, a different one on drive D:, and so on. As we'll see a little later, under Linux all files on all devices show up somewhere in the same directory tree, with “/” at its top.

Note that, whereas Windows uses “\” as the directory separator, Linux uses “/”.

To find out how much space is used by all the files underneath a given directory, you can use the “du” command, like “du -s -k phase1”, which would tell you the total size of all files under the subdirectory “phase1”, in kilobytes.
The directory tree of each physical device is grafted onto the same tree, with the root directory ("/") at the top. There are no "C:" or "D:" drives under Linux. Every file you have access to lives in the same tree, and you don't need to care what device the file lives on.
Viewing Mounted Filesystems with “df”:

<table>
<thead>
<tr>
<th>Filesystem</th>
<th>1K-blocks</th>
<th>Used</th>
<th>Available</th>
<th>Use%</th>
<th>Mounted on</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/mapper/LogVol0</td>
<td>73545144</td>
<td>37268984</td>
<td>32479988</td>
<td>54%</td>
<td>/</td>
</tr>
<tr>
<td>/dev/sda2</td>
<td>101105</td>
<td>45519</td>
<td>50365</td>
<td>48%</td>
<td>/boot</td>
</tr>
<tr>
<td>/dev/sdb1</td>
<td>721075720</td>
<td>630461080</td>
<td>53986040</td>
<td>93%</td>
<td>/data</td>
</tr>
<tr>
<td>tmpfs</td>
<td>2008536</td>
<td>0</td>
<td>2008536</td>
<td>0%</td>
<td>/dev/shm</td>
</tr>
<tr>
<td>home.private:/home</td>
<td>721075744</td>
<td>621413088</td>
<td>63034048</td>
<td>91%</td>
<td>/home</td>
</tr>
<tr>
<td>mail.private:/var/spool/mail</td>
<td>721075744</td>
<td>621413088</td>
<td>63034048</td>
<td>91%</td>
<td>/var/spool/mail</td>
</tr>
</tbody>
</table>

For now, just notice the last column of this output, which shows several different filesystems mounted within the directory tree. We'll talk about the details of this when we discuss filesystems, in a later meeting.
The Linux Directory Tree:
From the Linux Filesystem Hierarchy Standard: http://proton.pathname.com/fhs/

/  Top directory of the entire file system hierarchy.
/bin  Essential programs that need to always be available for all users.
/boot  Boot loader files.
/dev  Special files representing various devices.
/etc  System-wide configuration files specific to this computer.
/home  Users' home directories.
/lib  Libraries essential for the binaries in /bin/ and /sbin/.
/media  Mount points for removable media such as CD-ROMs.
/mnt  Temporarily mounted filesystems.
/opt  Optional application software packages.
/proc  Virtual filesystem, documenting kernel and process status as text files.
/root  Home directory for the root user.
/sbin  Essential system programs.
/srv  Site-specific data which is served by the system.
/tmp  Temporary files (see also /var/tmp).
/usr  Secondary tree, containing the majority of user utilities and applications.
/var  Tertiary tree for local data, specific to this computer.

Two important principles in Linux are:
1. There's only one directory tree.
2. Everything is a file.

Although it's possible to arrange the directories on a Linux system in many ways, there's an evolving standard layout called the Linux Filesystem Hierarchy Standard (fhs). All major Linux distributors use this now.
File Types ("Everything is a File"):

**Regular files (-):**
- `-rw-r--r-` 1 bkw1a bkw1a 20601 Jan 18 10:51 cluster.pdf

**Directories/folders (d):**
- `drwxr-x---` 3 bkw1a bkw1a 4096 Jan 18 11:35 phase1

**Symbolic Links (l):**
- `lrwxrwxrwx` 1 bkw1a bkw1a 11 Jan 18 11:39 clus.pdf -> cluster.pdf

**Block or Character Device Special Files (b or c):**
- `/dev/sda1` 1 root disk 8, 1 Dec 26 10:23
- `/dev/ttyS0` 1 root uucp 4, 64 Dec 26 10:22

**Unix Domain Sockets or Named Pipes (s or p):**
- `/var/lib/mysql/mysql.sock` 1 mysql mysql 0 Jul 1 2008
- `/var/xdmctrl/xdmctrl` 1 root root 0 Dec 26 10:24

For now, we'll only talk about regular files, symbolic links and directories. We'll save the other types for another time.
File Permissions:

```
-rwxrwxrwx  1 bkw1a demo 20601 Jan 18 10:51 myprogram
```

For each file, three sets of permission bits can be set, referring to three different classes of people: The user who owns the file (u), the group that owns the file (g) and everybody else (“other”, or o). For each of these classes, several permission bits can be set (or not), including read (r), write (w) and execute (x).

For the file above, the user (bkw1a) has permission to read, write or execute the file. The owning group (demo) has permission to read the file, but not to write it or execute it. Other users have no permission to do anything with the file.

Note that permissions are interpreted as though they were read from right to left. For example, if the user permissions give no write access, but the “other” permissions grant write access, then the user still won’t be able to write the file, even though others can. This is true since the user has been explicitly denied access in the user permissions.

Looking back at the output of “ls”, let's examine the file permissions column.

There are other things besides permissions that control access to files. We'll talk about attributes, ACLs and other things later.
**File Timestamps:**

**mtime:** The file's “modification time”. This is the time that the file's contents were last modified. This is the time that “ls” displays by default.

**ctime:** Somewhat confusingly, this is the “change time”. This is the time at which the file's properties (excluding contents) were last changed. For example, if the file's permissions are changed, or its ownership is changed, the ctime will change. To see the ctime of files, use "ls --time=ctime"

**atime:** This is the file's “access time”, showing the last time the file was looked at. Many administrators currently disable the updating of atime stamps, since they entail some I/O overhead and are seen as being of little value. To see the atime of files, use "ls --time=atime"

Typically, Linux filesystems store several different timestamps for each file.

Ctime is useful because hackers will often modify the mtime stamps of any files they've changed, to hide the hackers' activities. The almost always forget to change the ctime stamp, though.
“Hidden” Files:

```
~/.demo> ls -l
total 60
lrwxrwxrwx 1 bkwa bkwa 11 Jan 18 11:39 clus.pdf -> cluster.pdf
-rw-r----- 1 bkwa bkwa 20681 Jan 18 10:51 cluster.pdf
-rw-r----- 1 bkwa demo 983 Jan 18 10:53 cpuinfo.dat
-rw-r-r-- 1 bkwa bkwa 29 Jan 18 10:59 data-for-everybody.1.dat
-rw------- 1 bkwa bkwa 41 Jan 18 10:56 ForYourEyesOnly.dat
drwxr-x--- 3 bkwa bkwa 4096 Jan 18 11:35 phase1
drwxr-x--- 2 bkwa bkwa 4096 Jan 18 10:55 phase2
-rw-r----- 1 bkwa demo 72 Jan 18 10:52 readme.txt
-rw-r----- 1 bkwa bkwa 9552 Jan 18 10:52 ReadMe.txt
~/.demo> ls -a
```

Another switch to use with “ls”. Look at the difference between the output of the two commands.

Any file whose name begins with a dot is a “hidden” file. These aren't hidden for any security reason, they're just not shown so things will be less cluttered. Generally, these files contain configuration information of one type or another.

Two particular hidden files will always appear in every directory. These are “.” and “..”. These files give you a way to refer to the current directory (.) and its parent directory (..). For example, the command “ls .” does the same thing as “ls”. To move up one directory with cd, you could type “cd ..”. The properties of the “.” and “..” directories, as shown by ls, tell about permissions, ownership, etc. of the respective directories.

Note that the root directory also contains a “..” entry. In this case, since there's nothing above the root directory, the “..” entry just points back to root itself. (”/” is its own parent.)
Wild-cards ("globbing"):

The commands “ls -l junk.txt” or “ls /home/bryan/demo/phase1” will tell us about the named file or directory. We can also use wild-cards to specify filenames that match a pattern. This process is called "globbing" in Linux:

* Match any string of characters.

? Match any single character.

[abc] Match a single a, b or c.

[a-zA-Z] Match any character in the range a-z or A-Z.

Note that these patterns won't match filenames beginning with a dot ("hidden" files).

It's important to remember how wild-cards work on the Linux command line. If we type a command like “ls *.txt”, here's what happens internally:

“ls *.txt” is expanded to read “ls a.txt b.txt c.txt....” by inserting the names of any files that match. Then the resulting expanded command line is executed.

The “ls” command itself never sees the wild-card characters. The command-line interpreter (the “shell”) expands the wild-card expression before invoking “ls”. Because of this, wild-cards work for any command, since wild-card support doesn't have to be built into the command itself. Commands just get this functionality for free. This is another example of the modular philosophy behind the design of Linux and other varieties of Unix.

There are two types of pattern-matching in Linux, “globbing” and “regular expressions”. For most things you'll do on the command-line, globbing is used. We'll talk about regular expressions, which are much more powerful but also more difficult to understand, when we discuss scripting.
Some Examples of Wild-card Matching:

```bash
~/demo> ls -l
total 2
-rw-rw-- 1 bkw1a bkw1a 0 Aug 6 18:42 a.1
-rw-rw-- 1 bkw1a bkw1a 0 Aug 6 18:42 b.1
-rw-rw-- 1 bkw1a bkw1a 0 Aug 6 18:42 c.1
-rw-rw-- 1 bkw1a bkw1a 466 Aug 6 17:48 t2.sh
-rw-rw-- 1 bkw1a bkw1a 750 Jul 30 09:02 test1.txt

~/demo> ls -l t?.sh
-rw-rw-- 1 bkw1a bkw1a 466 Aug 6 17:48 t2.sh

~/demo> ls -l [ab]*
-rw-rw-- 1 bkw1a bkw1a 0 Aug 6 18:42 a.1
-rw-rw-- 1 bkw1a bkw1a 0 Aug 6 18:42 b.1

~/demo> ls -l [a-c]*
-rw-rw-- 1 bkw1a bkw1a 0 Aug 6 18:42 a.1
-rw-rw-- 1 bkw1a bkw1a 0 Aug 6 18:42 b.1
-rw-rw-- 1 bkw1a bkw1a 0 Aug 6 18:42 c.1
```

Here are a few examples of glob-style wild-card matching.
Part 3: Documentation
Documentation: Command-line help:

Many commands will tell you about themselves if you give them a "-h" or "--help" switch on the command line. For example:

~/demo> ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftvSUX nor --sort.

Mandatory arguments to long options are mandatory for short options too.
-a, --all         do not ignore entries starting with .
-A, --almost-all  do not list implied . and ..
-@, --author      with -l, print the author of each file
-b, --escape      print octal escapes for nongraphic characters
 --block-size=SIZE use SIZE-byte blocks
-B, --ignore-backups do not list implied entries ending with -
-c                with -lt: sort by, and show, ctime (time of last
                 modification of file status information)
                 with -l: show ctime and sort by name
                 otherwise: sort by ctime
-0, --classify    append indicator (one of *~=>@) to entries
-C                list entries by columns
 --color=[WHEN]   control whether color is used to distinguish file
                 types. WHEN may be 'never', 'always', or 'auto'
-d, --directory   list directory entries instead of contents,
                 and do not dereference symbolic links
-D, --dired       generate output designed for Emacs' dired mode
-f                do not sort, enable -au, disable -lst
--classify
.....

Note that this is just a convention, and not all commands will honor it. As we noted before, these commands have a long history, and were written by many authors.
For information about using the man command, don't hesitate type type “man man”.

Man pages are the most common type of online documentation for Unix-like operating systems.
Some commands have only info pages. These commands will typically have a minimal man page that only refers you to the info page.

For information about navigating around inside info, try typing “info info” at the command line.
Part 4: Text Editors
It's important to know how to use a text editor because most Linux configuration files are text files. For example, the “.login” or “.bash_profile” shell startup files are just plain text files.
## Some Common Linux Text Editors:

<table>
<thead>
<tr>
<th>Editor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vi</strong></td>
<td>The original Unix “visual editor”. Found on all Unix-like computers. Rather non-intuitive.</td>
</tr>
<tr>
<td><strong>emacs</strong></td>
<td>Developed later by RMS, emacs is a very powerful, extensible editor. Emacs and vi are the two most commonly-used Unix editors.</td>
</tr>
<tr>
<td><strong>pico</strong></td>
<td>A small, intuitive editor included with the “pine” mail program. Found on systems where pine is installed.</td>
</tr>
<tr>
<td><strong>nano</strong></td>
<td>A standalone clone of pico, with enhancements.</td>
</tr>
</tbody>
</table>

I recommend you learn to use emacs, but any of these are fine. It may be easiest to start out by learning pico/nano.
Using nano:

~/demo> nano purplecow.txt

I've never seen a purple cow.
I never hope to see one.
But I can tell you anyhow,
I'd rather see than be one.

Instructions at the bottom show you how to do basic operations, like saving the file and exiting nano. Note that "^" is just shorthand for "hold down the CTRL key".
Some Emacs Commands:

<table>
<thead>
<tr>
<th>Starting Emacs</th>
<th>Quitting</th>
<th>Help</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; emacs</td>
<td>C</td>
<td>M T</td>
</tr>
<tr>
<td>-s filename</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Q</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cursor Positioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward right char.</td>
</tr>
<tr>
<td>Back left char.</td>
</tr>
<tr>
<td>Up char.</td>
</tr>
<tr>
<td>Down char.</td>
</tr>
<tr>
<td>Left word</td>
</tr>
<tr>
<td>Right word</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Search and Replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>search pattern; cursor moves as you type.</td>
</tr>
<tr>
<td>search for pattern; cursor moves as you type.</td>
</tr>
<tr>
<td>Search backwards for pattern; cursor moves as you type.</td>
</tr>
<tr>
<td>Search for the next occurrence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>set mark at cursor</td>
</tr>
<tr>
<td>kill region</td>
</tr>
<tr>
<td>copy region to kill ring</td>
</tr>
<tr>
<td>paste region to kill ring</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loading and Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>create new filename for editing (clear workspace)</td>
</tr>
<tr>
<td>load file for editing</td>
</tr>
<tr>
<td>save (also for editing)</td>
</tr>
<tr>
<td>copy (also for editing)</td>
</tr>
<tr>
<td>copy (also for editing)</td>
</tr>
</tbody>
</table>


This is copied from the Stanford site, above.
The End

Thanks!