1 Welcome to Unix

Welcome to the unix environment. Here you have the familiar point and click gui, but that's not why you're reading this, you're here for the shell. So what is this mysterious shell you've been told about by upper year students? It is a command line interface to your computer. Where the gui is like pointing and grunting the shell is using language. A beginner might not do much better than pointing and grunting, but with practice one can control a computer with the skill of a poet.

A few useful keys before we begin

As you've probably guessed the shell is a command line interface that you type into. It's essentially an interactive programming language, like the interactions window in DrScheme, but designed to operate on the system and work in a rather diverse environment. Its also designed for convinence, and as you'll discover, quite powerful. Before we get started there are a few keystrokes you'll need to be aware of.

First, the commands that you type in can be editted, similar to a line of text in a text editor. You can move the cursor left and right, erase text (both delete and backspace work as you're familiar with from your word processor, and ctrl+k will delete everything to the right of the cursor), and scroll through your command history (using the up arrow and down arrow for back and forwards respectively).

If you want to view your command history use the command history. To run an old command type !number where number is the number next to the command in history.

Most shells also provide tab completion of commands and file names, to use this just press tab after typing part of a file path or command, as long as it's unique. If it's not the shell will show you all the possibilities (this may be a rather long list).

Getting to the Shell

There are several ways to access the shell remotely and locally. Remote shells can be obtained by connecting to a server with a program called Secure Shell or ssh. If you're local to a linux or unix system just start any of the many terminal programs. (On OS-X, look for Terminal in Applications>Utilities. On Ubuntu look in the Applications menu under Accessories.)
From Windows

Unlike the other operating systems discussed here, Windows does not have a built-in Unix-like shell. There are several applications that provide Unix-like shell features on Windows, but the

There are several utilities that can provide access to ssh from Windows, the main one used by people at Waterloo is PuTTY http://www.chiark.greenend.org.uk/~sgtatham/putty/. There is also a program called Cygwin http://cygwin.org. It provides a Linux-like environment in Windows. You use it inside of a command-prompt window. So it’s just like using PuTTY, except you don’t have to be online to use it. Make sure the ssh package is installed, and then just type ssh.

From Mac OS-X

Mac OS X comes with X11 (more on X11 later) installed, and with X11 comes a unix shell with most of the features discussed here, including an ssh client. There is also a Terminal.app that you can find in /Applications/Utilities. It doesn’t require that you start X11, but then you don’t get the advantage of X11 forwarding. But that’s an advanced topic that will be discussed later. You can also use many of the non student environment commands on your mac. Like most Unix-like operating systems, OS-X comes with a built-in ssh client. All you need to do to connect using SSH is to open up Terminal and type in ssh user@host. (Go on, give it a try.)

From Linux

To connect to the student environment from linux just start a terminal program and use ssh like on OS-X, ssh will be documented in its own section. Also most of the things in this guide are useable on your local terminal as well.

ssh(1)
The ssh utility is invoked from your local command line like this

```bash
$ssh username@hostname
```

At this point you will be prompted for your password. For the student.cs environment hostname is student.cs.uwaterloo.ca or just student.cs from inside the network. Your user name is your quest id.

**ssh keys (optional)** ssh can be configured to log in automatically without prompting you for your password using cryptographic keys to authenticate. To set this up do the following:

```bash
$ssh-keygen -t dsa
```
Generating public/private dsa key pair.
Enter file in which to save the key ("/path/to/identity_dsa"): Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ~/.ssh/id_dsa.
Your public key has been saved in ~/.ssh/id_dsa.pub.
The key fingerprint is:
<some numbers> user@host
$ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (~/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ~/.ssh/id_rsa.
Your public key has been saved in ~/.ssh/id_rsa.pub.
The key fingerprint is:
<some numbers> user@host
$ssh-copy-id user@host
user@host’s password:

This is optional and more detail can be found in the man page. More on those in the next section...
Also, please keep in mind that while the passphrase can be, and too often is, left blank, this poses an enormous security risk and should only be done if you are 100% certain you are the only person who will be using the computer in question, and your hard drive is encrypted.

Getting Help
Now that you can log in easily (or at least login if you skipped the section on keys) you’re probably wondering what you can do, we’ll get to that, but first a way to get help beyond this guide. For most commands (which are just programs) the authors often include manual or man pages (the good ones anyway) documenting how to use the program. To access this information use the man command like this:

$man <command name>

There are also manual entries on various system calls and program libraries, configuration files, and system daemons. These are in different numbered sections, type man man for more information.

When you are done reading a manual page, press q to exit from the reader.

Thats all well and good, what can I do?
Like we said earlier, the shell can be used to command a computer. So anything you could do in graphical programs you can do on the shell and more. Many shell tools don’t translate well to pointing and clicking, just like its hard to read a poem to a caveman. Commands don’t function in a vacuum though, they take input a variety of ways. The first is standard input, normally attached to your
keyboard (but it can be moved around), the second is command line arguments and flags. Arguments are just placed after the command, and the man page can explain which flags are available and how to use them. Two common ones are -help and -verbose which have obvious functions. The third input method is reading from various files, some specified on the command line some searched for by default.

We'll start with some basic file tools and command and control abilities.

The file system and what to do about it

Programs are just files on disk and manipulate data which is also on the disk, so it obviously follows that the most important programs are the ones for working with files on the disk. This section will go over the file system and these important programs.

The file system is organized into directories

The file system in the unix environment starts with / or the root, different drives are all attached as directories under the root. For now don't worry about drives and lets chat about directories. In unix everything has a running or "working" directory, including your shell prompt. To see what your current directory is use the print working directory command `pwd` at the prompt. This will give you the absolute path to your working directory. Any relative paths you specify will be relative to this location. To go somewhere else use the change directory command `cd` followed by the path you wish to change to. With no arguments it takes you to your home directory. The path specified can be relative or absolute.

**absolute paths**

Absolute paths are specified starting from the system root and look like `/this/path/is/absolute`. Absolute paths are important when working with other people or when going somewhere completely different on the filesystem, really anywhere precision is needed.

**relative path**

Relative paths are specified relative to your (or a program's) current working directory and are just like they sound. They start without a `/` (remember that `/` specifies the system root) and otherwise work exactly the same. There are a few special directories that can be used in relative paths, the first is ., which stands for the current directory. The second is .., which is the directory above the current directory. Here are some examples:

```
$pwd
/home/user
$cd /etc/games
$pwd
/etc/games
```
Which directories exist isn’t set in stone, you remove directories with the `rmdir` command, and make them with `mkdir`. `rmdir` takes an existing path and removes the directory (you can’t remove one with anything in it). `mkdir` takes a valid path and makes a directory.

Paths may seem confusing, but they are a rather straightforward way of specifying both directory locations and filenames. How do you specify a filename you ask? Just like a directory. If you’re in `/path/to` and want a file called `afile` all you need to do is write `afile`. If you’re somewhere else you can get to it absolutely `/path/to/afile` or use any of the other things mentioned in path specification to name it.

Files, they’re people too

Speaking of files, you probably want to see them. The first step to seeing files is seeing where they are, the command for this is the `list` command `ls`, which shows the files in your current directory, as well as directories. `ls` might be set up for colour coded output, it might not be, the man page can explain how to set that up. `ls` also has a long format, viewed using `ls -l`, which provides a lot of useful information about the files listed. Again the man pages can give you more information.

Now that you know the names of your files you might want to actually get at their contents. The simplest way is `cat` which takes a file name and spits out the contents to the screen. This is nice for short files, but you can’t scroll back. Two commands that give you a little more control are `head` and `tail` which display the start and end of files. `tail` is particularly useful in watching log files using the `follow` flag `-f`, which watches the file for updates and displays the new entries as they appear. A third way to view files, which is perhaps the most convenient, is something known as a `pager` program. There are many pagers, this author’s favorite is `less`, which takes a filename argument and lets you scroll up and down as well as search. To search in less type `/` followed by what you’re looking for. To find more matches press `n` to go forward and `N` to go back. In less `/` actually takes a pattern, to learn about those check the man pages or the text manipulation and editing tutorial.
You might be asking how to create files now. Lots of programs create files as output, there are logs, data files, random crap, so what you probably want is a way to edit text. That's easy, use `pico` or `nano` for simple command line editors. However these editors are rather simple and lack power, for a full introduction to the vast text editing capabilities refer to the text manipulation and editing tutorial.

Files can also be moved around, copied and removed. The commands `mv` and `cp` do moving and copying, taking the original filename as the first argument and the destination as the second. If the second is a directory then the filename is kept the same and the file is placed in that directory, if a name is specified then the file will be moved or copied to the new name. `rm` removes files given to it as arguments, as many as you like.

**Your home folder (there's no place like it)**

When specifying paths there is another special character `~`, which is short for your home directory. Your home directory is exactly what it sounds like, your place on the file system to keep your crap. If you use `cd` without an argument it will take you to your home dir. What's in your home dir? Well, any files you want to keep. Is that it? No, also kept in your homedir are so-called "dot files" since their filenames start with a ., so they don't normally show up in `ls`, to view them use the `all` option for `ls -a`. These dot files store configuration information for various programs, as well as run commands files with commands to be run on startup of various applications. You can edit these to customise your various preferences for applications, but consult the man pages first to make sure you enter sane values.

Also in your home folder is your web folder, usually called `www` or `public_html`, depending on the configuration of the system. To set up a personal web page all you have to do is place your html (or text or whatever) files in there and make sure they are world readable. Word Readable? What? Its a permission, subject of the next section.

**File permissions, or how to keep people out of your crap.**

In unix every file has an owner, these owners are the users, and an owning group, which is some group that the owner is a member of. You can control who can read, write, and execute your files based on these categories: user, group, everyone. Collectively these permissions are known as the file's (or directory's) *mode*. To view a file's mode use a long listing with `ls -l` and look at the *mode string*. Mode strings are 10 characters, the first character specifies if the file is a file with a - or something special, like a directory d or a character device like a terminal with c. Following the file type are nine characters, they represent the 3 permissions for the 3 groups, the first three characters stand for read, write, and execute for the user, and repeat for the group and everyone. For example `-rw-r-r` provides the owner read-write access and everyone else.

To change *modes*, *owner*, or *group* of a file use the commands `chmod`, `chown`,
and chgrp (actually chown can also change groups). We won’t go into chown since only the superuser (a user with more permissions than normal, for administrative purposes) can use it. To use chmod to change modes first pass in a mode alteration followed by files you own. A mode alteration has the form w[+,-]p where w stands for who the mode change is for, user owning the file, group, others, or all groups, followed a + to grant the permission or a - to revoke it, and p stands for the ability to change, read, write, or execute.

You may be wondering about the execute permission. This grants people the ability to run your files, if you have binaries or executable scripts to permission. Depending on the programs you might want to be careful with this. On directories, where executing makes little sense it is the permission to traverse or change into the directory.

Talking about more than one file at a time

Often its useful to talk about a group of files (like to tell rm about them, or backup your website with cp). The shell has a way to do this. Using a wildcard * you can name all files that match a pattern, for example *.mp3 would be all mp3s in a folder. Multiple wildcards can be used in a single name, but note they don’t match /, that is they don’t traverse directories. There is also the ? wildcard which matches any one character.

Specific characters can also be matched using [..] with the characters or a range of characters (say [a-z] to match all lowercase) to match. Note that this only matches once, so for repeats you must repeat your bracket expression.

For more information on this matching/expansion, which is called file globbing consult man 7 glob (there will probably also be an entry in section 3 of the system manual for glob. This is for a C library function that actually works out the list of files from the glob expression).

If you want to specify a *, ? or [] you must escape it with a \, and to get a literal \ you must escape that as well, using \\ in its place. There are a few other characters that must be escaped: a space, ", . , and ; for reasons that may or may not be obvious now, but will be after the next section.

Power searching: finding files with find

The find command is a flexible tool used to locate files and folders in a given folder (or set of folders). The basic structure of the command is

\[
\text{find [Folder(s)]] [Search Expression]}
\]

Note that the Search Expression is not simply a string that you wish to match - it is more powerful and more complicated than that. As an example, this command:

\[
(1) \text{find } */ -name "cs*"
\]
will locate all files in your home folder that begin with "cs". Find searches recursively, meaning that it will even find matching files when they are located inside subfolders in the home directory. To adjust the number of levels down that 'find' will search, you can use the -maxlevels and -minlevels flags.

You can also use find to execute a command on each file returned using the -exec flag. Everything after the -exec flag will be considered part of the command you wish to run on all files until the sequence \; (an escaped semicolon) is encountered. You can use the sequence {} to represent the name of the current file. As an example, this command will search all files in your home directory for the phrase "Hello World" and print the name of the files that have matches.

(2) find ~/ -exec grep -q Hello\ World {} \; -print

The search expression in find is evaluated left-to-right as a boolean expression based on the success or failure of each parameter. The -exec and -print expressions are implicitly joined by an AND operator.

Example 2 can have this implicit AND written explicitly as:

(3) find ~/ -exec grep -q Hello\ World {} \; -and -print

Whereas:

(4) find ~/ -exec grep -q Hello\ World {} \; -or -print

would show only files that "didn't" have matches.

Find will stop processing the expression once the final result is known. In (3), -exec returning false means that the whole expression is false, since false AND anything is false. In (4), if -exec returns true, -print will not be processed, since the exit state is already guaranteed to be true.

What the shell really gives you.. power.

Up until now we've only really covered features that you can find in your familiar point and click environment. Going back to the language analogy think of that as boring vocabulary work and what's about to be covered grammar, rhyme, and meter work. This section discusses the features of the shell that separate it from the gui, the programmable loops, variables that affect program execution, and io redirection. This section describes the machinery the shell provides to put words together to form ideas.

Every command you run is a process: how to run more than one.

Every command you start from the shell (well there are a few exceptions we'll go in to in a bit) runs as a process. In Unix processes represent separate discrete running programs, for example you can have several processes which are all cp with different arguments (or the same arguments, but that is likely to break and
not advised). In playing around with what you've learned in the last chapter you may have been frustrated by only being able to do one thing at a time. The shell does not limit you to running one thing at a time, and on the contrary gives you powerful tools for managing processes, both those started by you and those already running on the system.

For the purposes of job control, processes can be viewed in two groups: those under the control of your shell (your jobs), and those not. We'll first discuss the ways of starting your own jobs and interacting with them.

Your jobs and what to do about them The first thing you're probably wondering is how to start more than one job. To do this you first need to know the difference between a foreground process and a background process. A foreground process is running attached to your terminal, your command prompt goes away and is replaced by its output and any input goes to the program. With a program running in the background it doesn't receive input (unless you have an io redirect, more on that later) and output goes to your terminal, but you can still enter other commands.

Before we can get to fancier features you probably want to know how to stop something in the foreground. To interrupt or cancel the job in the foreground press CTRL+C (hereafter control will be abbreviated with ^ followed by a character. So CTRL+C is ^C and so on and so forth). If you want to pause the job instead of terminating it press ^Z. This stores the process in a stopped state and gives you your command prompt back. You'll also see a message that looks like

\[[1]+ Stopped some command with args\]

The number in square brackets is the job number, + means the state has changed. This is followed by the new state and what job is being run. If you want to bring your job back to the foreground type \texttt{fg \%1} and your job will come back just as you left it. Not very interesting, but the next part is. If you want to let your job finish running in the background while you do something else type \texttt{bg \%1} and your job will resume running, but you still have a prompt to use. If you know you want a command to run in the background and don't want to futz with ^Zing it and then typing bg you can just put an & at the end of your command like so:

\$\texttt{cp ~/giant\_file.tar.gz /mnt/usbdrive &}

Note that the number 1 in \%1 isn't special, its just the usual job number of your first job. It will be different if you have more jobs running. To get a list of your jobs type \texttt{jobs}. This will have job numbers, states and what command is running.

To terminate a job early or put one of your background in a "Stopped" state (and often to communicate with it in other ways) unix provides signals. A signal is a numbered code sent to a process with a meaning defined by the system.
Commonly used signals are terminate, `TERM`; kill, `KILL`; and stop, `STOP`. `TERM` instructs a process to stop what it’s doing and quit, `KILL` ends it, and `STOP` is equivalent to `^Z`. Yes, `^C` has a corresponding signal: interrupt, `INT`. To send signals to jobs use the `kill` command. By default `kill` sends the `TERM` signal, to specify other ones use `-s` followed by the signal name. You also need to tell `kill` which job to send the signal to.

**Other processes** You may be wondering about the mention of processes not started by your shell. These all have associated numbers, called a *process id*. Your jobs also have pids, and `kill` will take a pid instead of a job id. To see a listing of current processes use the command `ps`. This will give all sorts of information: the pid, who started it, the attached terminal, what’s running, the ram usage. `ps` gets more useful when you’re using more than one terminal or performing system administration.

**Input, output, and the story of a plumber**

As stated earlier processes running take input from standard input (your keyboard) and send it to standard output (your terminal) or errors to standard error (also your terminal). Sometimes this can be really inconvenient or downright aggravating. Fortunately the shell has an answer, its called *IO redirection*. This is the process of sending output somewhere other than your terminal and getting it from somewhere other than your keyboard.

**Simple redirection** Simple redirection sends program output to a file or brings input in from a file. To connect input to a file use the `<` character followed by a file name like this:

```
$mysql < lots_of_sql_queries.sql
```

Output goes the other way, so naturally you use `>` to send output to a file. By default one `>` overwrites a file, for output to append you must add a second `>>`. Before going further, a word on *file descriptors*. In unix when a file is opened by the system the open file handle isn’t attached directly to the file on disk, but to a file descriptor, which is a number that the process can use internally. File descriptors are also used by interpreted scripts, which when executed start an interpreter. Instead of passing the interpreter the filename (which could have been changed to do nasty things if someone is up to no good) the system passes the interpreter the file descriptor of an open handle to that file, as these are process specific and much harder to muck with. Normally you don’t have to worry about file descriptors, but when redirecting input and output its nice to be aware of 3 of them. These are fds 0, 1, and 2, which correspond to standard in, standard out, and standard error respectively. The left sides of `<`, `>`, `>>` all take an implicit fd argument, 0 for the input operator and 1 for the output operators. To redirect standard error add a 2 in front of your operators. You
can also redirect from one file descriptor to another using i>&j to send output on fd i to fd j.

File descriptors will be revisited when we discuss scripts, as you might have a need for more than the 3 default file descriptors there.

You might be wondering if you can read from or write to other programs. That's the topic of the next section, just remember with simple redirection you can't read from or send to programs.

**You mentioned plumbing? I don't see any pipes** Actually, there are pipes in the shell, they are one of the most useful shell features. A pipe or pipeline is a way to connect standard out of one program to standard in of another. This is represented with the | pipe character. To demonstrate its power say you have a command that generates a lot of output, you don't really want to save it but you need to look at it, and it doesn't fit on one screen. Using what you know above you do this:

```
$massive-output > output.txt
$less output.txt
$rm output.txt
```

Seems like a lot of work, since less by default reads from standard in. With pipes this mess gets shortened to:

```
$massive-output | less
```

You can also chain pipelines using inbetween commands as filters, you can even use many of the commands that normally take files in pipelines using the special - filename, consult the man pages for what you're using to get the specifics, a lot of commands take input from standard input by default. If you want to start the pipeline out with some text you enter you can put it in a file or you can use cat - to send standard in to standard out. When you are done typing press "D to send the End of File (EOF) character (Note not all control characters send signals).

Chaining pipes was mentioned, this allows you to insert filters into the stream we'll go over a few useful ones now, though for more in depth coverage check the text editing tutorial.

**tee**

`tee` creates a split in a pipe, saving the output to a file (or multiple files) as well as passing it on standard out so the pipeline can continue. Useful for saving state or putting a T in your pipeline and starting the other branch with cat `your_` tee savefile `| rest_of_pipeline`.

```
$ command_pipeline | tee save_file | more_pipeline
```
grep

grep's odd name comes from an antiquated unix editor and a common sequence of commands used to search. These commands are globally, regular expression, print, used to go through an entire file and print lines matching a pattern. For details on grep's pattern format check the text processing tutorial or the man page. grep can be used on files but its mentioned here as a filter. When used in a pipe it will filter what's coming in on the pipeline, outputting only what matches the pattern, for example:

```
$ps | grep mysql
```

will print information about every process with mysql somewhere in its information (usually things run by the mysql user or mysql client processes).

sed

sed stands for stream editor, and provides powerful facilities for editing data as it comes in through a pipeline. sed can also automate various other editing tasks. More information on sed can be found in the editing tutorial.

sort

sort does exactly what you think it does, takes input, outputs a sorted version. The various options in the man page control different ways of sorting, but its not worth mentioning them here, as I'm sure you'll look it up when you actually need it instead of reading it now and forgetting it.

uniq

uniq takes input and outputs only the unique lines. A useful option is -c which gives you a count of repetitions.

wc

wc is a word counting program. It takes input and outputs the number of lines, words, and characters in the input. It will also take a file or many files and give you by file output as well as totals. You might write a version someday...

Variables and the Environment (no you don't have to go green)

At this point you may have guessed that the shell is in fact an interactive programming language designed specifically for gluing other programs together and administering systems. If you did, you're right. If not, well it is, and like most programming languages it has support for variables. Variables in the shell come in two forms: locally scoped variables and environment variables. Locally scoped variables are visible only to one command (commands may start multiple programs, see other parts of this chapter for how) while environment variables
are visible to everything that runs as a child of your shell (unless otherwise specified, but we'll get to that).

To set local variables just use = in a statement, like:

\$var="stuff";

and to get values append a $ to the name and the shell will substitute it:

\$echo $var

(ec ho just prints out what you pass to it).

Environment variables are global to your shell and allow you to customize various shell options and inform programs you start of your preferences for things like your pager and your preferred text editor. To get a full rundown of the available vars check out the manpage for environ(7) (if you haven't read at least a bit of man man yet the number is a manual section, and to specify it place it before the name of the page: man 7 environ). To set new ones or modify your current ones use export followed by your assignment statement, or just the variable name if you have it set already. If you're curious about what variables are already set just run export without arguments.

If you want environment variables to remain set in new sessions edit your .bash_profile or .profile file and add new lines with your export commands. For more on these files see the customizing your shell section.

Scripting the shell

Since the shell is really just an interactive programming language an obvious question is "does it have a non-interactive mode?". The answer is yes. This is called shell scripting and is a useful way to automate repetitive tasks and add desired functionality. Since the interactive shell and the scripted shell are the same thing, features discussed here can be used at the prompt, though they may not be useful.

Shell scripts consist of a series of shell commands on separate lines or separated by semicolons. To create one open an editor on a new file (call it whatever you like, by convention they have the .sh extension) and enter the sequence of commands you wish to be able to run over and over again. Save the file and make sure you have permission to execute it. Then at the prompt type ./$your_file_name to run it.

Scripts can also take arguments, these appear in variables $1 to $9, (if you need more than nine place the numbers in braces). There are also some useful special variables: $? is the exit status of the last command run, an indicator of error conditions; $$ is the process id of the script, and $0 is the name of the script, which provide relevant information for error messages as well as the change of behavior based on script name or process id (the first one is sometimes used for startup/shutdown scripts, the second one is insane).
Its a programming language? Where's while and friends?

Yes, the shell is an interactive programming language, and yes there are loops and conditional expressions. If you've ever used an imperative programming language (like Java or VB or Perl) these first few should be familiar. These are going to be quick commentaries, for more info you can read the shell man page.

```
if
  If works just like you'd expect, except there are also a myriad of useful
  file test options to query the properties of files and file descriptors.
  
  if [ -e "filename" ]; then
    echo "filename exists";
  fi

  if [ "$3" = "start" ]; then
    echo "starting";
  elif [ "$3" = "stop" ]; then
    echo "stopping";
  else
    echo "option 3 invalid";
  fi

while, until
  While loops while a condition is true, until loops until a condition is true.
  These are the same conditions used in if.

  while [ -e "lockfile" ]; do
    echo "waiting for lock to be released";
    sleep 5;
  done;

  X=0
  until [ $X = 20 ]; do
    echo "zomgwhat";
    X=$((X+1)) # arithmatic expression, read the man page. You probably
    # want to use a real programming language for heavy iteration
  done;

for x in
  While bash has a c style for loop, don't use it. You probably want a real
  programming language at that point. The useful for loop iterates through
  a list for you, setting x (or whatever other variable name you use) to the
  values one at a time. You can use a plain list, a file glob, or any command
  in backticks (which evaluates the command and substitutes the value).
```
# Backup git repositories in /backup/repos
for d in `ls -d /backup/repos`; do
cd $d;
git pull;
done;

Customizing your shell

Through this document we've been talking about "the shell". The fact is there are many, many, shells all with the standard features discussed here (and some without) as well as a variety of extensions and unique features.

The shell used as a reference when writing this guide and a common default is bash, Bourne Again Shell, also popular are ksh (Korn Shell), csh (C Shell), tcsh (an enhancement of csh), and zsh (the Z Shell, an addition of new features to a ksh and bash features). Feel free to experiment, you can start a new shell by invoking it as a command or using chsh and logging out and in again.

To set shell options and environment variables you can edit the various rc and profile files associated with your shell, check the man page for details on which files to edit.

In these files you can also start background jobs if you'd like, though it is often more useful to have automated tasks. Many systems have a cron program installed which can be used to run programs even when you aren't around, to add these edit a file called a crontab using the program with the same name to do so. Check the associated man page for syntax in the file and what you can and cannot run in it.

About the Authors (when they wrote this)

This document is maintained and copyright The University of Waterloo Computer Science Club 2008 and is released under the terms of the Artistic License version 2.0 available at http://www.perlfoundation.org/artistic_license_2_0.

Edgar Bering (ebering@csclub.uwaterloo.ca)

A student heading into his second year Edgar served as club secretary W08 and plans on running for executive most of the rest of the time he's around.