A Much-Too-Terse Introduction to Unix in the ODU Computer Science Dept.

STEVEN J. ZEIL

MARCH 30, 2000

Instructions on navigating this document.

A printable version of this document is available.

1. Working on the CS Dept Network

The Computer Science Department maintains a large network of Sun workstations for teaching and research. With only a few exceptions, if you are taking a CS class, you will receive an account allowing you to work on these machines.

These workstations run the Unix operating system. This document is designed to provide a quick introduction to the commands and concepts you will need to work under Unix.

If you have worked before with networks of Windows or Mac PCs, you need to understand that Unix offers a fundamentally different approach to running programs. On PC networks, when you run a program, you run it on the machine you are sitting at and you see the output on the machine you are sitting at. Unix programs are designed to run on any machine where they have been installed, and you still see the output on the machine you are sitting at. That's simple enough to accomplish when the program is doing simple text I/O, but the Unix philosophy also applies to programs that do graphics, windows, etc. That's one of the big reasons why Unix skills are so important, especially to distance-learning students. Both instructors and students can provide, design and build programs on the same machines, secure in the belief that if it runs properly for them, it will run for others as well.

There are two ways to interact with Unix and Unix programs: through a *text-only* interface, or via a *windows* interface — the Unix windowing system is called *X*. Which mode of interaction you use depends upon how you are accessing to the network, the software on the machine you are sitting at, and how fast your connection to the network is. But even if you are running X, one of the first things you are likely to open is an "xterm", a text-only command window. Unix users tend to launch programs from the text-only interface. If the program itself supports windows, mice, etc., then they can point and click to

their heart's content.

For many of the programs I discuss here, there are alternatives that will accomplish the same task. Experienced users may well argue with my choices. In particular, the X windowing system provides simpler interfaces for electronic mail, debuggers, etc., but these are normally available only when seated at the console. I tend to prefer techniques that can be used in both X and text-only modes, because these offer more flexibility.

2. The Basics

2.1. Logging In

2.1.1. Making a Connection

If you are a student registered for a CS course, and have never had a Unix account on the CS Dept system, you can get your account by going to the CS Dept home page and clicking on the "Account Management" link (under "Online Services").

If you have had an account in the recent past, it should be regenerated for you in any semester when you are registered for a CS course. Otherwise, you will need to contact your instructor or from the CS systems staff to get your account.

The first thing you must do is to decide on what machine you are going to work. If you are seated at one of the Dept.'s Sun workstations, the answer is easy — you will use the machine you're seated at. Otherwise, you need to pick a workstation. The easiest way to do this is to use the name "`lab.cs.odu.edu`", a "fake" machine name that actually requests that you be assigned a random selection among the more lightly loaded machines.[1] If you're not into pot-luck, however, you can select a specific machine from this list.Note that you will probably need to add the string "`.cs.odu.edu`" to any of these names.

Next, you need to get a log-in prompt from an ODU CS machine. How you do so depends upon your access mode:

---

[1] If you know that you are going to be doing somve very CPU-intensive work, you can also opt for the fake name "`fast.cs.odu.edu`", which makes a random selection among the faster CPU's in the Dept.

CS Dept. Sun Workstations If you are seated at one of the Dept.'s Sun workstations, just hit the `return` key to bring up the login screen. The screen will offer you the choice of logging in to the "Common Desktop Environment" (CDE) or to "OpenWindows". These are different "flavors" of X. You can use either one, but the CDE offers built-in tutorials for first-time users.

Other Unix/Linux Machines If you are logged into a non-CS workstation running Unix or Linux, and your internet connection is reasonably fast[2], go ahead and start X on that machine, if you have not already done so. Get into an `xterm` window, and issue the commands

```
xhost +machine-name
xon machine-name
```

where *machine-name* is the name of one of the CS Dept. machines (see this list).

Because you are connecting from outside the CS Dept., you must give the full machine name, ending in `.cs.odu.edu`. In the above lines, the `xhost` command gives the selected machine permission to send information to your screen, and the `xon` command makes the actual connection.[3]

---

[2] A 56k modem is OK, but you wouldn't want anything much slower.

[3] Note that you cannot use the fake names "`lab.cs.odu.edu`" and "`fast.cs.odu.edu`" because these choose a machine at random, and would make a different choice each time. Therefore you would wind up giving one machine permission to draw on your screen, and then trying to connect to a different machine.

The `xhost` command isn't necessary if your X server is set to give permission by default or to prompt you for specific permission when a connection is made. Try just doing the `xon` command and see if that works on your system. If so, you can use `lab.cs.odu.edu` and `fast.cs.odu.edu`.

Win32 machines with X servers If you are seated at a Win95/98/NT (collectively referred to as *Win32*) machine with a reasonably fast internet connection[4], you might want to see if it has X server software installed. The most commonly used X-servers appear in the Start button menu as “Exceed” or “X-Win32”.[5]

The trickiest part is getting started. There are two techniques:

1. The recommended technique is to create a “session” describing how you want to connect. For Exceed, this is done by running the “Session wizard”. For X-Win32, you start the `X-Util32` program, and select “New session”. In both cases, you want to create an `rexec` session running the command

   ```
   /usr/openwin/bin/xterm -ls -display $DISPLAY
   ```

   on one of the CS machines (see ‘Logging In’ on page 4). Run this session (e.g., run X-win32, click on the X-win32 icon on the taskbar, and select your new session) to connect to the CS network.

2. The alternative is to first connect via telnet, set your DISPLAY variable to point to the machine you are sitting at and then connect to the remote machine using `xon`:

   ```
   setenv DISPLAY local-machine-name:0
   xon remote-machine-name
   ```

---

[4] Again, a 56k modem is OK, but you wouldn’t want anything much slower.

[5] If you want to try using X on your own PC, StarNet has a fully functional demo of their X-Win32 server available for free downloading.

Other machines If you are seated at a machine that doesn't have X server software, or that has a slow internet connection, then you need to run a `telnet` program. I can't tell you where that will be or how to run it, as that information depends upon the machine you're sitting at. (I can tell you, however, that the Win32 operating systems come with a program named (surprise!) "`telnet`".) Use your telnet program to connect to `lab.cs.odu.edu` or a specific machine you have chosen.

Before going any further with telnet, find out the kind of terminal being emulated by your telnet program. Some programs may give you a choice of several terminals. You may want to try different ones to see which works best for you. Common choices include "vt100", "vt102", "vt52", "vpoint", and "adm3a". You'll need that information in just a minute.

2.1.2. Logging In

Now that you have a login prompt, enter your login name. At the "password:" prompt, enter your password.

2.1.3. You're logged in - What will you see?

Exactly what will appear on your screen after you successfully enter your password depends on your access mode and, for console users, the particular machine you chose and how your account is set up.

Telnet: You will have a "command-line" interface - you can type commands and see the output, if any, listed immediately after the command.

Console, X: Console users may get nothing more than a command-line interface as well. If so, you will want to quickly move up to using the X windows interface. Console users may also find themselves placed automatically into one of two "flavors" of the X windowing system:

Common Desktop Environment You can recognize this because, when you logged in, the phrase "Common Desktop Environment" was prominently displayed. You can also recognize it by the fancy "toolbar" at the bottom of the screen containing a number of icons and, in the middle, a set of buttons titled "One", "Two", "Three", and "Four".

In this environment, look for a window titled "Terminal", or click on the toolbar's picture of a screen and keyboard to get a terminal window. Click on the bar across the top of the window to select it, and you are ready to begin entering commands.

For help on all the features of this environment, click on the toolbar icon with the question mark.

other X window managers On those machines that do not run the common desktop environment, you may find yourself confronted with a screen with one or more "windows". The appearance and behavior of these windows is described further here.in Section . For now, though, try moving the mouse to an unoccupied portion of the screen and holding down the right mouse button. You will probably get a menu that includes an option to open a "terminal", "xterm", or "shell". Select this to get a windo3w in which you can enter commands. Click on the bar across the top of the window to select it before typing anything.

2.1.4. Setting Your Terminal Type

If you are at the console, you may skip this Section. If you are connected via via the Internet `telnet` program, you must now tell the Unix workstation what kind of terminal you are using. The command to do so is

```
setenv term xxxx
```

where *xxxx* is the kind of terminal (e.g, `setenv term vt100`).[6]

Some people also recommend that you follow this command with

```
tset -Q
```

which resets the terminal. In my own experience, this is usually unnecessary,and I have found that many communications programs don't deal well with this, but try it if your terminal seems to be misbehaving.

Most "dumb" terminals provide for 24 lines of text. Many communications programs, however, allow more. If yours is one of these, you should tell Unix how many lines you are using by giving the command

```
stty rows nn
```

where *nn* is the number of rows/lines.

[6]The proper terminal name for Internet users depends upon the machine you are actually sitting at. Consult the documentation for your telnet program, the local staff, or your Internet Service Provider for suggestions.

2.1.5. Changing Your Password

Whether we like it or not, we need to worry about the security of our computing environment. There are people who would take advantage of this computer system if they had any, or more complete, access to it. This could range from the use of computer resources they have no right to, to the willful destruction and/or appropriation of the information we all have online. In order to maintain the level of security in our computing environment that we need, there are some things we all have to take responsibility for. Even though you may not feel like you personally have much to lose if someone had access to your account or files, you have to realize that as soon as someone gains ANY access to our system, it's 100 times easier for them to gain access to ALL of it. So when you are lax with your own account, you are endangering the work and research of everyone else working here.

Your password is the fundamental element of security not only for your personal account, but for the whole UNIX system that we share. Without an account and password a person has NO access to our system. If someone discovers (or you tell someone) your password, not only will they have access to your personal files, but they will have a much better chance to launch attacks against the security of the entire system.

Your account password is the key to accessing and modifying all of your files. If another user discovers your password, he or she can delete all your files, modify important data, read your private correspondence, and send mail out in your name. You can lose much time and effort recovering from such an attack. If you practice the following suggestions, you can minimize the risk.

1. **NEVER** give another user your password. There is no reason to do this. You can change permissions and have groups set up if you need to share access with other individuals. Your account should

be yours alone.

2. Never write down your password. Another person can read it from your blotter, calendar, etc. as easily as you can.

3. Never use passwords that can be easily guessed. Personal information about you (birth date, etc.) may be known to the attacker or may be recorded in on-line databases that the attacker has already obtained.

   Passwords should not be single words (in any language) because on-line dictionaries are widely available for use in spelling checkers. A common approach to cracking passwords is to compile a set of such words and to run a program that tries each one on each account on the machine. Consider inserting punctuation and other "odd" characters into your password to foil such attacks.

   A person with local knowledge can also try your spouse's name, pets' names, etc. Your account is vulnerable to this type of cracking unless you choose your password carefully.

4. Change your password the very first time you log in, and every few months thereafter. Security problems are often traceable to stale passwords and accounts. These are accounts that have become inactive for one reason or another or the password has not changed for a long time. In our particular environment we have had break-ins via such stale accounts. A password that remains the same for a long time provides an intruder the opportunity to run much more advanced and longer running programs to break such passwords.

5. Vary the system by which you choose a password. For example, don't repeatedly use combinations like BLUEgreen and REDyellow. If an intruder discovers your pattern, he or she can guess future

passwords.

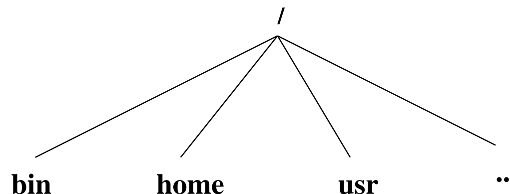
The command to change your password is

```
yppasswd
```

This command will first prompt you for your old password (just to check that you really are you!) and then will ask you to type your new password (twice, so that an inadvertent typing mistake won't leave you with a password that even you don't know!).

2.2. The Unix File System

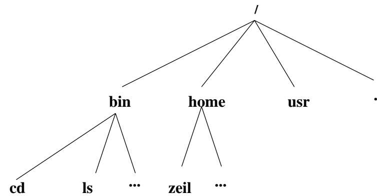
Files in Unix are organized by listing them in *directories*. Directories are themselves files, and so may appear within other directories. The result is a tree-like hierarchy. At the root of this tree is a directory known simply as "/".[7] This directory lists various others:



[7] It may be more precise to say that this directory's name is the empty string "".

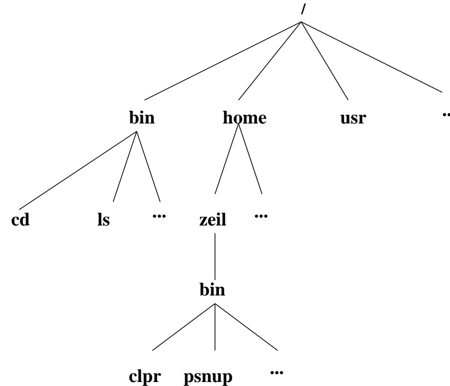
The `bin` directory contains many of the programs for performing common Unix commands. The `usr` directory contains many of the data files that are required by those and other commands. Of particular interest, however, is the `home` directory, which contains all of the files associated with individual users like you and me. Each individual user gets a directory within `home` bearing their own login name. My login name is `zeil`.

We can expand our view of the Unix files then as:



`cd` and `ls` are two common Unix commands, as will be explained later.

Within my own home directory, I have a directory also named "bin", containing my own personal programs. Two of these are called "clpr" and "psnup". So these files are arranged as:



The full name of any file is given by listing the entire path from the root of the directory tree down to the file itself, with "/" characters separating each directory from what follows. For example, the full names of the four programs in the above diagram are

```
/bin/cd
/bin/ls
/home/zeil/bin/clpr
/home/zeil/bin/psnup
```

There are some common abbreviations that can be used to shorten file names.

- You can refer to the home directory of someone with login name *name* as `~name`.

- You can refer to your own home directory simply as `~`.

  So you could refer to the file containing my `clpr` program as either `/home/zeil/bin/clpr` or `~zeil/bin/clpr`.

  When I myself am logged in, I can refer to this program by either of those two names, or simply as `~/bin/clpr`.

- At all times when entering Unix commands, you have a "working" directory. If the file you want is within that directory (or within other directories contained in the working directory), the name of the working directory may be omitted from the start of the filename. When you first log in, your home directory is your working directory. For example, when I have just logged in, I could refer to my program simply as `bin/clpr`, dropping the leading `/home/zeil/` because that would be my working directory at that time.

- The working directory itself can be referred to as simply ".".

- The "parent" of the working directory (i.e., the directory containing the working directory) can be referred to as "..".

Unix filenames can be almost any length and may contain almost any characters. As a practical matter, however, you should avoid using punctuation characters other than the hyphen, the underscore, and the period. Also, avoid blanks, and non-printable characters within file names. All of these have special meanings when you are typing commands and so would be very hard to enter within a filename.

Some things to keep in mind about Unix file names that may be different from other file systems you have used:

- Unix file names are often very long so that they describe their contents.[8] The rather perverse exception to this rule is that program/command names are, by tradition, very short, often confusingly so.
- Upper and lower case letters are distinct in Unix filenames. "`MyFile`" and "`myfile`" are different names.
- Periods (".") are not treated by Unix as a special character. "`This.Is.a.legal.name`" is perfectly acceptable as a Unix filename. Many programs, however, expect names of their data files to end in a period followed by a short "standard" extension indicating the type of data in that file. Thus data files with names like "`arglebargle.txt`" for text files or "`nonsense.p`" for Pascal source code are common.

  By convention, files containing executable programs generally do not receive such an extension.
- Keep in mind that directories are separated by "/" in file names, not by "\" as is common in some other operating systems.

2.3. Shell Games: Typing Unix Commands

To run a Unix command (or any program, for that matter), you normally must type the name of the command/program file followed by any arguments. There is actually a program running that accepts

[8] As we will see, one almost never needs to type an entire filename in a Unix command, so long file names are no harder to work with than short ones.

your keystrokes and launches the appropriate program. The program that reads and interprets your keystrokes is called the *shell*. There are many shells available, all of which offer different features. The default shell for ODU CS is called tcsh, and we'll concentrate on that.

The command/program name is usually not given as a full file name. Instead, certain directories, such as `/bin`, are automatically searched for a program of the appropriate name. Thus, one can invoke the `ls` command as either

```
/bin/ls
```

or simply as

```
ls
```

As you type, some characters have a special meaning. For example, if you have entered the first few letters of a file name and hit the "Tab" key, the shell will examine what you have typed so far and attempt to complete the filename by filling in the remaining characters. If the shell is unable to complete the filename, a bell or beep sound will be given. Even in this case, the shell will fill in as many characters as it can.

Most special characters are entered by holding down the "Control" key while typing a letter. By convention, we designate this by placing the symbol "ˆ" in front of the name of the second key. For example, if you have typed zero or more letters of a filename and want to see a list of what filenames begin with what you have typed, you could type ˆD, i.e., hold down the "Control" key and type "d".

Some other useful special keys are:

- ˆC is used to abort a program/command that is running too long or working incorrectly. Beware: aborting a program that is updating a file may leave garbage in that file.

- ˆD is used when a command/program is reading many lines of input from the keyboard and you want to signal the end of the input.
- ˆH, the “Backspace” key, and the “Delete” key all delete the most recently typed character.
- ˆB moves the cursor **B**ackwards over what you have just typed, without deleting those characters. This is useful in correcting typing mistakes. The “left” arrow on your keyboard may also do the same thing.
- ˆF moves the cursor **F**orwards over what you have just typed, without deleting those characters. The “right” arrow on your keyboard may also do the same thing.
- ˆP retrieves the **P**revious command that you had typed in. Repeated ˆP’s may be used to look back through a number of commands that you have issued recently.

  The “up” arrow on your keyboard may also do the same thing.
- ˆN is the opposite of ˆP. After one or more ˆP’s, a ˆN allows you to move back to the **N**ext more recent command.

  The “down” arrow on your keyboard may also do the same thing.
- In many programs, ˆZ pauses the program and returns you temporarily to the shell. To return to the paused program, give the command: `fg`

2.4. Some Basic Unix Commands

If you have not yet done so, log in now so that you can work though the following commands.

Upon logging in, your *working directory* should be your home directory. The command `pwd` will **p**rint the **w**orking **d**irectory. Give the command

```
pwd
```

You should see something like

~yourname

Now, let's make a place to play in. `mkdir` will make a new directory. Enter the command

```
mkdir playing
```

to create a directory named "playing".

The command `ls` lists the contents of the working directory. More generally, `ls` *`directoryname`* will list the contents of any directory.[9] Give the command

```
ls
```

and you should see `playing` listed. In fact, it may be the only thing listed.

The command `cd` is used to **ch**ange the working **d**irectory. Give the command sequence

[9] Well, not really *any* directory. People can protect their own directories from the prying eyes of others, in which case `ls` will fail.

```
pwd
cd playing
pwd
cd ..
pwd
cd ./playing
pwd
```

to see this in action.

The cp command copies one or more files. You can give this command as cp file1 file2 to make a copy of file file1, the copy being named file2. Alternatively, you can copy one or more files into a directory by giving the command as

cp file$_1$ file$_2$...file$_n$ directory

Now try the following:

```
ls ~public/Misc                  [Notice that there are a number
                                  of files ending with .txt]
cp ~public/Misc/*.txt ~/playing
ls ~/playing
```

The "*" is a *wildcard* character. It tells the shell to substitute any combination of zero or more characters that results in an existing filename. In cases where there are multiple possibilities, such as this one,

the shell forms a list of all the matches. So the `cp` command actually saw a list of all files in the `~public/Misc` directory whose names ended in ".txt".

To get a better feel for wildcards, try the following:

```
ls /usr/include
ls /usr/include/*.*           [List only file names containing a "."
ls /usr/include/f*.*
ls /usr/include/*f*.*
```

Here are some other common Unix commands. (A longer, more complete list can be found in `~public/Unix/commands.txt` and in Appendix A.) Try experimenting with these in your `~playing` directory.

cancel *request* Remove a file from the printer queue, so that it won't get printed. The *request* identifier is found using the `lpstat` command.

cat *file*$_1$...*file*$_n$ Lists the contents of each of the listed files on your screen.

exit Shut down the current shell. If this shell is the one you got at log-in, this command logs you out.

mv *file*$_1$ *file*$_2$ Renames *file*$_1$ as *file*$_2$. *file*$_2$ may be in a different directory.

mv *file*$_1$ *directory* Moves *file*$_1$ to the given directory.

lp *file* Send file to printer for printing. Most sites have multiple printers, each having its own name. One of these will be the "default" printer used by the `lp` command. For the others, you must give the printer name as part of the printer command:

```
lp -d printer file
```

For example, at the Norfolk ODU campus, `lp` by itself prints to a fast printer in the public workstation lab for text only. `lp -dcookie` prints to "cookie", a printer in the same room that offers the extra capability of printing Postscript graphics.

You will need to consult your local staff to see what printers are available at other sites.

lpstat Shows the list of files you have "queued up" awaiting their turn on printers. Entered by itself,

```
lpstat
```

it lists only your own print jobs, giving a unique identifier for each one.

To see the entire list of print jobs on some printer, enter

```
lpstat -o printer
```

ls -a By default, filenames beginning with "." are considered "hidden" and not shown by the `ls` command. The `-a` (for "**a**ll") option causes these files to be shown as well.

ls -l This is the "long" form of `ls`. It displays additional information about each file, such as the size and date on which the file was last modified.

Note that options can be combined. For example, you can say `ls -la` to get extra information including normally hidden files.

ls -F Adds a little bit of extra information to each file name. If the file is an executable program, its name is marked with an "*". If the file is a directory, its name is marked with "/".

more ***file***$_1$**. . .** ***file***$_n$ Lists files one screen at a time, pausing after each screen-full. Hit the space bar to advance to the next screen.

A related program is `less`, which also allows you to move backwards through the files by hitting "b".

rlogin ***machine*** Logs you in to another machine on the network. Use this if the machine you are on seems to be running slowly and the `who` command indicates that there are lots of others on the same machine.

rm ***file***$_1$**. . .** ***file***$_n$ Deletes the listed files. Be very careful using wildcards with this command. `rm *` will delete everything in the current working directory!

rm -i ***file***$_1$**. . .** ***file***$_n$ Deletes the listed files, but firsts asks permission to delete each one.

rm -r ***file***$_1$**. . .** ***file***$_n$ Deletes the listed files. If any of these files is a directory, it deletes that directory and everything in it as well.

rmdir ***directory*** Deletes a directory.

who Lists everyone logged into the same machine that you are using.

2.5. File Protection

2.5.1. Protections

Not every file on the system should be readable by everyone. Likewise, some files that everyone needs (such as the executables for commands like `cp`, `mv`, etc.) should not be subject to accidental deletion or alteration by ordinary users. This is where file *protection* comes into play.

Unix allows three forms of access to any file: read, write, and execute. For an ordinary file, if you have read (r) permission, you can use that file as input to any command/program. If you have write (w) permission, you can make changes to that file. If you have execute (x) permission, you can ask the shell to run that file as a program.

The owner of a file can decide to give any, all, or none of these permissions to each of three classes of people:

- To the owner of the file him/herself
- To members of a designated "group" established by the systems staff. Groups are generally set up for people who will be working together on a project and need to share files among the group members.
- To anyone else in the world.

These three classes are abbreviated "u", "g", and "o", respectively. The "u" is for "**u**ser", "g" for "**g**roup", and "o" is for "**o**thers". Until you actually join a project that needs its own group, you will mainly be concerned with "u" and "o" classes.

The `ls -l` command will show the permissions granted to each class. For example, if you said

```
ls -l ~/playing
```

you might see the response

```
-rwxrwx---  1 zeil       311296 Jul 21 09:17 a.out
-rw-rw----  1 zeil           82 Jul 21 09:12 hello.c
-rw-rw----  1 zeil           92 Jul 21 09:13 hello.cpp
-rw-rw----  1 zeil           85 Jul 20 15:27 hello.wc
```

Look at the pattern of hyphens and letters at the left. The first character will be a “d” if the file is a directory, “-” if it is not. Obviously, none of these are directories. The next 3 positions indicate the owner’s (u) permissions. By default, you get read and write permission for your own files, so each file has an “r” and a “w”. `a.out` is an executable program, so the compiler makes sure that you get execute (x) permission on it. The other files can’t be executed, so they get no “x”. This way the shell will not even try to let you use `hello.c` or any of the other source code files as a program.

The next three character positions indicate the group permissions. In this case, the group permissions are the same as the owner’s permissions. Knowing the group permissions isn’t very helpful, however, unless you know to which group it refers. The command `ls -g` produces output similar to `ls -l` but lists the files’ groups instead of their owners. For example, if you said

```
ls -l ~/playing
```

you might see the response

```
-rwxrwx---   1 student    311296 Jul  21  09:17 a.out
-rw-rw----   1 student        82 Jul  21  09:12 hello.c
-rw-rw----   1 student        92 Jul  21  09:13 hello.cpp
-rw-rw----   1 student        85 Jul  20  15:27 hello.wc
```

Some typical groups are “wheel”, “faculty”, “gradstud”, and “student”. “Wheel” has no members, but groups like “student” and “gradstud” have very broad membership, as their names imply. Although, as we shall see, the files in this example do not give any privileges to the world (others), they can be read and written by all students because of the group permissions.

The final three character positions indicate the permissions given to the world (others). Note that in this case, people other than the owner or members of the same group cannot read, write, or execute any of these files.

Directories also can get the same `rwx` permissions, though the meaning is slightly different. If you have read permission on a directory, you can see the list of files in the directory via `ls` or other commands. If you have execute permission on a directory, then you can use that directory inside a file name to get at the files it contains. So, if you have execute permission but not read permission on a directory, you can use those files in the directory whose names you already know, but you cannot look to see what other files are in there. If you have write permission on a directory, you can change the contents of that directory (i.e., you can add or delete files).

2.5.2. chmod

The `chmod` command changes the permissions on files. The general pattern is

```
chmod class±permissions files
```

Use "+" to add a permission, "-" to remove it. For example, chmod o+x a.out gives everyone permission to execute a.out. chmod g-rwx hello.* denies members of your group permission to do anything at all with the "hello" program source code files.

You can also add a -r option to chmod to make it "recursive" (i.e., when applied to any directories, it also applies to all files in the directory (and if any of those are directories, to the files inside them, and if...). For example, if I discovered that I really did not want the group to have permission to write or execute my files in ~/playing, I could say:

```
chmod -r g-rx ~/playing
```

2.5.3. Beware the umask!

Suppose you never use the chmod command. What would be the protection levels on any files you created?

The answer depends upon the value of umask. Look in your ~/.cshrc file for a command by that name, and note the number that follows it. If you don't have one, just give the command

```
umask
```

and note the number that it prints.

The umask number is a 3 digit (base 8) number. The first digit describes the default permissions for the owner (you), the second digit describes the default permissions for the group,[10] and the final digit

[10] Of course, if the number appears written using only 1 or 2 digits, the missing digits are simply leading zeros.

describes the default permissions for others.

Each of these three numbers is, in turn, formed as a 3-digit binary number where the first digit is the read permission, the second is the write permission, and the thrid digit is the execute permission. In each binary digit, a 0 indicates that the permission is given, a 1 that the permission is denied.

So if my umask is 027, that means that

- I (the owner) have 000 — permission to read, write and execute my own files.
- The group to which a file belongs has 010, permission to read, no permission to write, and permission to execute that file.
- The rest of the world has 111, no permission to read, write or execute.

Of course, these permissions can be changed for individual files via the `chmod` command. The umask only sets the default permissions for cases where you don't say `chmod`.

If you want to change your default permissions, you do it via the `umask` command by giving it the appropriate 3-digit octal number for the new default permissions. Some common forms are:

`umask 022` Owner has all permissions. Everyone else can read and execute, but not write.

`umask 077` Owner has all permissions. Everyone else is prohibited from reading, writing, or executing.

Since the point of the `umask` command is to establish the default behavior for all your files, this command is normally placed within your `.cshrc` file.

2.5.4. Planning for Protection

At the very least, you will want to make sure that files that you are preparing to turn in for class assignments are protected from prying eyes. You need to do a little bit of planning to prepare for this. There are two plausible approaches:

- Use a stringent enough umask (e.g., `umask 077`) so that everything is protected by default.
  - The only disadvantage is that files that you *want* to share (e.g., the files that make up your personal Web page) must be explicitly made world-readable (`chmod go+r` *`files`*).
- Use a more relaxed umask (e.g., `umask 022`) so that your files are readable by default, but establish certain directories in which you carry out all your private work and protect those directories so that no one can access the files within them. For example, you might do

  ```
  cd ~
  mkdir Assignments
  chmod go-rwx Assignments
  ```

  Now you can put anything you want inside `~/Assignments`, including subdirectories for specific courses, specific projects, etc. Even if the files inside `~/Assignments` are themselves unprotected, other people will be unable to get into `~/Assignments` to get at those files.
  - The one disadvantage to this approach is that it calls for discipline on your part. If you forget, and place your private files in another directory outside of `~/Assignments`, then the relaxed umask means that those files will be readable by everyone!

2.6. Getting Help

As you explore Unix, you are bound to have questions. Some ways to get answers include:

- The entire Unix manual is on-line.

  `man` *`command`*

  displays the manual page for the given command.

  `man -k` *`keyword`*

  looks up the given keyword in an index and lists the commands that may be relevant.

- The CS Department systems staff has collected a variety of additional help documents. If you are using X windows, you can access these via the command

  ```
  netscape &
  ```

  and then selecting "Unix & Labs".

- A staff member is generally on duty in the public CS workstation room of the Norfolk campus whenever that room is open.

- If none of the above help, then send e-mail to "root". This is also how you report bugs, machine failures, etc.

3. Editing Text Files

An *editor* is a program that allows you to easily create and alter text files. There are a variety of editors on the system, of which the most popular are `vi` and `emacs`. Neither is exactly the easiest thing in the world to learn to use. I suggest learning `emacs`, because

- It offers a built-in tutorial to get you started.
- As you gain more facility with `emacs` and with Unix in general, you will find that `emacs` offers many advanced facilities for doing specific tasks. For example, I use `emacs` to compile my programs, and to aid in debugging them.
- `emacs` is widely available (for free) for all Unix systems and also for MSDOS.

To run `emacs`, make sure that you have correctly identified your terminal kind (see Section 2.1). Then give the command

```
emacs
```

Then follow the directions given to bring up the tutorial (i.e., type ^\ followed by "t".).

When you are done with the tutorial, here are few extra things you should know about `emacs`:

- `emacs` offers a customized modes for different kinds of files that you might be editing. Some of these are chosen automatically for you depending upon the file name you give. Others can be chosen automatically by giving the command `M-x` *`name`*`-mode` where *name* indicates the desired mode. Some of the most popular modes are: `text`, `pascal`, `c`, and `c++`. The programming

language modes generally offer automatic indenting at the end of each line, though you may have to end lines with the "Line feed" or "C-j" key rather than "Return" or "Enter" to get this.

- The command `M-/` is a special friend to all programmers who use long variable names but hate to type them. Type a few letters of any word, then hit `M-/`. `emacs` will search backwards through what you have previously typed looking for a word beginning with those letters. When it finds one, it fills in the remaining letters. If that wasn't the word you wanted, just hit `M-/` again and `emacs` will search for a different word beginning with the same characters.

- Before starting up, `emacs` tries to read a file `~/.emacs` Many people store special commands in there to customize `emacs` to their own liking. (Reading the `.emacs` file of an experience `emacs` user can be instructive although, unfortunately, sometimes a bit intimidating. (Feel free to take a look at mine: `~zeil/.emacs`

4. X Windows

If you are working at the **console**, you can take advantage of the *X* windowing system. By running `X`, you can have several windows on the screen open at once, each devoted to a different task. For example, you can be reading electronic mail in one window while a lengthy compilation is running in another. `X` also allows the display of graphics and of a variety of fonts.

4.1. X Window Managers

`X` is a windowing system that can present a number of different appearances to the user. The appearance and behaviors that you actually see is controlled by a *window manager*, a program that is generally run as part of the `X` start-up procedure.

The most common window manager in our Dept. is the *Common Desktop Environment*. Most of our workstations are set up to run this all the time, as soon as you log in. Help and tutorial information is available by clicking on the picture of a set of books with a question mark in front, usually located near the bottom right corner of the screen.

4.2. Running X

If, however, you log in at a console and find yourself staring at a simple command-line interface, try giving the command

```
~public/xdemo/demo
```

to run an on-line tutorial of X under the default window manager for our environment, or give the command

```
X
```

to run `X` “for real”.

4.3. Working in X

I will simply note a few important items, including some not described in the tutorials:

- One of the first things you will want to do is to get a working window where you can enter Unix commands.

  In the Common Desktop Environment, look for a window titled “Terminal”, or click on the toolbar’s picture of a screen and keyboard to get a terminal window.

  In other X window managers, try moving the mouse to an unoccupied portion of the screen and holding down the right mouse button. You will probably get a menu that includes an option to open a “terminal”, “xterm”, or “shell”. Select this to get a windo3w in which you can enter commands. Click on the bar across the top of the window to select it before typing anything.

  Click on the bar across the top of the window to select it, and you are ready to begin entering commands.

- Any time you enter commands in Unix, you can place an ampersand ("&") at the end of the command to run that command in the background. This "disconnects" the command from your keyboard (in that window). You get a prompt immediately and can enter your next command even if the one you just launched has not yet completed.

  Now this capability is not all that useful if you're not running X. After all, if the program you are running needs input from you, it has been disconnected and can't see your subsequent keystrokes. Also, if that program produces output, it will still appear, but will be intermingled with the outputs of any new commands you have entered in the meantime. So, if you're not in X, the & is useful only for commands and programs that need no additional inputs and produce no additional outputs.

  Under X, however, many useful programs open their own windows and direct their inputs and outputs through those new windows. For example, you would enter "`emacs &`" rather than "`emacs`", and "`netscape &`" rather than "`netscape`". Without the &, the window where you entered the command to launch a program would be useless to you until that program has finished. With the &, that program runs in its own window and the old window gets a new prompt and can still be used to issue more commands.

- Most programs that run under `X` support a very simple "cut-and-paste" facility. Simply drag the mouse across a block of text in any window while holding down the left mouse button. Then position the mouse into a window where you would like that text to be "typed". Click the middle mouse button, and the selected text will be sent to that window just as if you had typed it yourself.

- When `emacs` is run under `X`, this cut-and-paste feature is supported, but in a different fashion. Text that has been selected in another window by dragging the mouse can be retrieved in `emacs` by the

command C-Y (ˆY). Text that has been “killed” in emacs by C-K, C-W, or M-W can be inserted into other windows by clicking the middle mouse button.

5. Customizing Your Unix Environment

By now, you are probably tired of typing "setenv term..." and other initial commands each time you log in. Now that you can edit files, one of the first things to do is to customize your login procedure.

The shell uses two important files to customize behavior. When the shell is started up, it executes the commands in a file called ~/.cshrc Now, in a typical Unix session, you may actually be running many copies of the shell at the same time, perhaps without even being aware of it. The first time, however, that a shell is started because you have just logged in, it also executes the commands in a file called ~/.login

You may or may not already have versions of these files.[11] You can check by giving the command

```
ls -a ~
```

If you don't have either of these, you should make one. If you do, consider changing it as described here.

First, let's create a .login file. Enter emacs, and create a file with the following:

```
if ("$TERM" == "vt100" || "$TERM" == "network" ) then
  set term=vt102
#   tset -Q
endif
```

[11] Note that because each of these filenames starts with a ".", you won't see these files with a ls command unless you use the -a option.

Instead of “vt102”, you should enter whatever terminal type you use most often. If you find that the tset command has been necessary for you in the past, delete the # character. If you usually reset the number of lines with stty, add that command as well. Now your terminal kind will be set automatically for you whenever you dial in.

If you would like to automatically run X whenever you are working at the console, add the following lines:

```
if ("`tty`" == "/dev/console") then
   echo -n "Entering X windows  (Control-C to interrupt)"
   sleep 5
   X
endif
```

Be sure to enter everything exactly as shown above, including the quotes. Note that " is the double-quotation key, and that the ` characters around the word tty are the “backwards” apostrophe (You may need to hunt around on your keyboard to find this one, but it will *not* be on the same key as the ").

Now let’s add a few useful items to the .cshrc file. Edit your .cshrc file and insert the following:

```
setenv EDITOR emacs
limit coredumpsize 0
#
#   skip remaining setup if not an interactive shell
#
if ($?USER == 0 || $?prompt == 0) exit
```

```
set history=40
set ignoreeof
set prompt="`hostname`: "
alias cp               'cp -i'
alias mv               'mv -i'
alias rm               'rm -i'
alias ls               'ls -F'
alias ff              'find . -name \!* -print'
```

The setenv line indicates that emacs is your editor of choice. Some programs, including the e-mail programs introduced in 'Using Electronic Mail' on page 41, will use this information to load an editor when you have large amounts of text to enter/alter.

Of the remaining lines, the most interesting are the alias commands. These set up "abbreviations" for commands. In this case, we are mainly adding options to familiar commands. The first three aliases add a -i option to the cp, mv, and rm commands. This option causes each command to prompt for a confirmation whenever its action would result in a file being deleted. The fourth alias adds the -F option to all ls commands, making it easier to tell the difference between ordinary files, directories, and executable programs. The final alias sets up a "find-file" command, ff. This will search the current directory and all subdirectories within it for a file matching a given pattern. For example the command sequence

```
cd ~
ff '*.txt'
```

will list all of your files with the `.txt` extension.

After you have checked these two files and saved them, you will have to log out and then log back in again before they take effect.

6. Using Electronic Mail

Electronic mail, or “e-mail”, for short, is an important part of the ODU CS environment. Besides being a useful way to exchange personal messages, e-mail is used by the Department for official announcements. Many instructors distribute grades by e-mail. They may send hints and corrections for projects and assignments that way, or distribute special files needed by all students in the class. E-mail may be the best way to pose a short question to your instructor outside of class, since you don’t actually need to catch your instructor in person at a time when he/she’s not busy with someone else.

Of course, you may already have an e-mail account at work, with the University, or with your own Internet Service Provider (ISP). If you prefer to receive all your mail at another account, see section 6.3 for instructions.

6.1. E-Mail addresses

Just as with physical mail, you can’t send someone e-mail unless you know their name and address. For e-mail, the name and address are usually combined as

```
name@machine
```

where *name* is the login name of the recipient and *machine* is the full name of the computer that processes their mail. This combination is generally called the person’s “e-mail address”.

For example, my login name is “zeil”, and my mail is handled by the machine “cs.odu.edu”, so my e-mail address is `zeil@cs.odu.edu`.

Actually all e-mail for CS Dept. login accounts is handled by cs.odu.edu. When you are sending mail to a user with the same mail handling machine, you can omit the "@" and everything that follows it. So if you are logged in to a CS Dept machine and want to send me e-mail you could just send it to zeil. But if you are logged in to a Teletechnet PC or a machine elsewhere on the Internet, you would need to give the full form, zeil@cs.odu.edu.

6.2. E-Mail Programs

There are several programs that you can use to get e-mail, and people tend to become rather fanatical about their personal favorite. The most basic of these is the Unix mail command, which also has the advantage of being universally available on any Unix machine. But mail is showing its age. New standards (the Multipurpose Internet Mail Extensions or MIME for short) have evolved to allow people to package files, graphics, sounds, etc., as part of an "extended" e-mail message. The mail command predates these standards and so cannot handle MIME e-mail. Also, mail is not the easiest e-mail program to learn.

I recommend the pine program for e-mail on our system. It is menu-driven, includes a substantial built-in help system, and can process and send MIME mail. I do occasionally fall back on the basic mail command, so I describe both of these in the following sections.

Later, you may want to check out the X-Windows mail tool, the mush, or elm programs, or the vm command for reading e-mail from within the emacs editor.

```
PINE 3.90   MAIN MENU                          Folder: INBOX  22 Messages

        ?      HELP                 -  Get help using Pine

        C      COMPOSE MESSAGE      -  Compose and send/post a message

        I      FOLDER INDEX         -  View messages in current folder

        L      FOLDER LIST          -  Select a folder OR news group to view

        A      ADDRESS BOOK         -  Update address book

        S      SETUP                -  Configure or update Pine

        Q      QUIT                 -  Exit the Pine program
```

Figure 1: Pine Main Menu

6.2.1. The PINE E-mail program

To run `pine`, make sure that you have correctly identified your terminal kind. Then give the command

```
pine
```

You should see a menu resembling Figure 1. If you get a garbled screen instead, you probably have not set your terminal kind correctly.

The most important choices are “C” to compose a message and send it to someone, and “I” to view an index of messages sent to you.

```
To       :
Cc       :
Attchmnt:
Subject :
----- Message Text -----






^G Get Help   ^X Send       ^R Rich Hdr   ^Y PrvPg/Top ^K Cut Line   ^O Postpone
^C Cancel     ^D Del Char   ^J Attach     ^V NxtPg/End ^U UnDel Line^T To AddrBk
```

Figure 2: Sending mail with Pine

Sending Messages Type "C" to compose a message to send to someone. You should see a screen resembling Figure 2. One thing to note is the list of possible commands in the lower two lines of the screen. In almost any context, `pine` will list the commands available to you, including a command to get "help" information.

Use your up/down arrow keys to select the "To:" line at the top of the screen. Here you can enter the e-mail address you want to send a message to. Just below that, on the "CC:" line, you can add the e-mail addresses of any other people to whom you would like copies of your message sent. Two lines down is the "Subject:" line. Although you are not required to put anything here, proper e-mail "etiquette" calls for all messages to carry a useful entry in the "Subject:" line.

Finally, move the cursor below the "Message Text" line, and you can begin typing your message.

When you are done and are ready to send your message, type ^X to send it.

A common variation on this procedure is when you need to send someone a copy of a file as part of your message. For example, if you are sending your instructor a question about some code you are writing, you might want to include the code in question as part of the message. Pine provides two ways to do this. The easiest is to go up to the "Attchmnt:" line near the top of the screen. Any file names you type on this line will be "attached" to the final message when it is sent (i.e., a copy is sent — your original files will be untouched). Alternatively, while you are typing your message you can ˆR to insert a file directly into the text of your message.

Some point to keep in mind when deciding which approach to use are:

Attachment:

- Recipient of message must be using `pine` or some other MIME-capable mail program.
- Can be used to send non-text files (e.g., programs, graphics, etc) as well as standard text.
- Can send multiple files without confusion. File names are preserved as part of the attachment.

ˆR

- Recipient of message can use any mail program.
- Can only be used to send text files.
- File names are lost. If you try to include more than one file in a message, the boundaries between the files are likely to be unclear to the reader.

```
  PINE 3.90   FOLDER INDEX                 Folder: INBOX  Message 5 of 5

+ A 1   Nov  6 JohnDoe@elsewhere. (34,483) Looking for volunteers
+   2   Nov  8 JohnDoe@elsewhere.  (2,472) Still need volunteers
    3   Nov 13 MaryJones@aol.com   (4,310) Conference on Programming
    4   Nov 20 Root                (1,432) Re: your account
    5   Nov 22 Professor Z           (747) Overdue Homework




? Help        M Main Menu  P PrevMsg     - PrevPage    D Delete      R Reply
O OTHER CMDS V [ViewMsg]  N NextMsg   Spc NextPage    U Undelete    F Forward
```

Figure 3: Reading mail with Pine

Receiving Messages From the main menu screen, type “I” to get a list of the messages in your system “mail-box”. It will look something like Figure 3. The plus signs in front of some messages indicate that you have already read them. The “A” in front of message 1 indicates that you have already sent an answer to that message.

Using your up/down arrow keys, you can select any of these messages and then type “V” to view the message.

While viewing messages, refer to the bottom two lines of your screen for the commands to page up/down through long messages, to compose and send replies to a message, to “forward” a copy of the message to someone else, or to return to the main menu (Figure 1).

If the message you are viewing is a MIME style message with attached files, another “V” command will allow you to view these files (if they are text, graphics, etc) or to save them in a directory of your

choice. Also, the “E” (Export) command will allow you to save the text of the current message in a file even if it is not a MIME message.

Folders After you have read some messages and try to exit from `pine`, `pine` will ask if you wish it to move your read messages out of your system mail-box (called the “INBOX”) into a “read-messages folder”. A *folder* is simply a container that can hold mail messages. `Pine` treats your system mail-box as simply a special form of folder. To see a list of your folders and to select one in which you want to view messages, use the “L” command from the `pine` main menu.

Moving read messages out of your system mail-box into another folder is often a good idea. It eases strain on the system resource area used for incoming e-mail. It means that when you enter `pine` and immediately hit the “I” key, you see your new mail right way instead of having it mixed in with old stuff. Finally, you can organize your saved mail by creating your own folders to save messages in. For example, you might have a folder for each class you are taking, to keep e-mail about different classes in separate containers. You can create new folders from inside the “L” command. To move a mail message that you are viewing into a folder, use the “S” save command.

6.2.2. The Unix mail command

Sending To send mail to someone with e-mail address *addr*, give the command

```
mail addr
```

For example, you could send me mail via the command

```
mail zeil@cs.odu.edu
```

Although, if you are logged in to a CS Dept machine and want to send me e-mail you could just say

```
mail zeil
```

After you have given the `mail` command, you will be prompted for a subject line to indicate what your message is about. After that, you begin typing your message on the next line. When you are done, type ˆD on a line by itself. You will then be prompted with "`Cc:`", which allows you to add the login names of other people to whom you would like to send a copy of your message. (Many people like to make a habit of cc'ing a copy to themselves.) If you do not want to send any extra copies, just hit the "Return" key. Your message will then be sent.

As you type your message, you can send special instructions to the mail program by entering any of the following at the start of a line:

˜e Enter the editor named by the EDITOR environment variable (see Section 5). This is a good way to correct mistakes made on previous lines.

˜r *filename* Insert the contents of a file into your mail message.

˜p Print the message as it appears so far.

˜m # If you are actually replying to a mail message that you received (see Receiving, below), this inserts the text of mail message number # into your reply.

Receiving When you first log in, you will be informed if you have received e-mail. At that time, or anytime thereafter, you can use the `frm` command to list the messages awaiting.

To actually read your mail, give the command `mail` with no arguments. You should see a numbered list of your messages. If not, the command "`h`" (for **h**eaders) will list them. You can then read a message by typing it's number.[12]

After reading the message, you can take any of several actions:

r Send a **r**eply to the author of the message you just read.

R Send a **r**eply to the author of the message you just read and to anyone in the Cc: list of that first message.

dp Delete this message and move on to the next (if any).

n Move on to the next message.

s *filename* Save a copy of this message in the specified file. If the file already exists, the message is added to the end.

[12]If you have no messages at the moment but would like to practice reading mail, try following the instructions under Section 6.2.2 to send yourself a couple of messages. Then just wait a few minutes until `frm` indicates that your messages have arrived.

6.3. Forwarding Addresses

If you prefer to read your e-mail on a different system, you can easily tell the Unix mailing system to forward all mail sent to your Unix e-mail address to a different address.

You will want to create, in your home directory (i.e., `cd ~`) a file named "`.forward`". The contents of this file should be a single line of text containing your preferred e-mail address.

You can create this file using your favorite text editor, or you can simply use the Unix `echo` command to write the desired text into the file. For example, if you wanted all your e-mail to be sent to `bogus@megacorp.com`, you would do the following

```
cd ~
echo "bogus@megacorp.com" > .forward
cat .forward
```

The final `cat` command should show the contents of the `.forward` file to be your desired address. (Note: Unix files that start with a "." are invisible to the normal `ls` command. To see them in a directory listing, you have to add the `-a` option: `ls -a`.)

Now, test it out! If you have a bad e-mail address in your `.forward` file, you could lose messages. So send yourself mail (to your ODU CS account). It should appear, in due course, at your preferred e-mail address. How long it actually takes depends on many factors. It may take only a few minutes. If after a few hours, you have not received the e-mail, delete your `.forward` file. Try again, if you wish. Or you might try on another day just in case the CS Dept. mail server, or the one at your preferred site, was temporarily out of commission. If you have repeated problems getting mail forwarded

quickly, you might want to rethink your desire to use this feature. For most people, this procedure works without much trouble.

You can actually have more than one forwarding address. The `.forward` file can contain a comma-separated list of forwarding addresses. For example, you might use

```
bogus@megacorp.com, bogus@home.net
```

Some people like to keep a backup copy of their mail on the CS Dept system, but to get their ”normal” mail somewhere else (e.g., because their mail server at work is crash-prone). This is possible, by making your e-mail address on the CS Dept. system one of the forwarding addresses, so that you forward a copy right back to yourself:

```
\yourLoginName, bogus@megacorp.com
```

The backslash is required: it helps prevent the mailer from consulting your `.forward` file a second time (which would lead to an infinite cycle of mail forwarding).

7. File Transfer

If you prepare files on one machine but want to use them on another, you need some means of transferring them. For example, if you edit files on your home PC or on a PC at one of the Teletechnet sites, you will eventually need to get those files onto the CS Department network. On the other hand, you may want to take files your instructor has provided off of that network for use on your home PC.

Exactly how you do this depends upon your usual access to the CS network. You may need to try several approaches until you find one that works well for you.

7.1. Text versus Binary Transfers

A further complicating factor is that you must decide whether the files you want to transfer should be treated as "text" of as "binary". Files that contain simple text, including program source code, should be transferred in "text" mode. Compiled programs, compressed files (e.g., *.zip or *.Z files), and word processor files with embedded formatting codes are generally transferred in "binary" mode.

The reason for this binary/text confusion is that Unix, MSDOS, IBM, and other systems disagree on how to represent basic text. For example, the end of a line in a Unix text file is represented by a single character (the ˆJ or "line-feed" character) while MSDOS uses a pair of characters at the end of each line (a ˆM or "return" character followed by a ˆJ). Other operating systems have their own peculiarities. Most file transfer programs will, when transferring text, try to convert the transferred file into the appropriate format for the destination machine. These conversions may involve changing, adding, or deleting characters. Of course, if the file being transferred were not text but a compiled

program, any such changes to individual bytes would be disastrous. Consequently, you need to be aware at all times whether the files you are working with are text or binary.

The easiest way to tell (though not foolproof) is to try listing the file on your screen using the Unix "cat" command or the MSDOS *type* command. If it looks OK, its probably text. If not, or if in doubt, transfer it in binary mode.

7.2. Transferring Files

7.2.1. At the console:

If you have physical access to the CS workstations, either on the Norfolk campus or at the Peninsula Graduate Center, you may be able to transfer your files via 3.5" floppies. The floppies must have been previously formatted on an IBM PC compatible machine, for either 720k or 1.44M capacity.

Look for a workstation with a floppy disk drive (not all have them). Insert the disk into the drive, and type `volcheck` to notify the workstation that a disk has been inserted.

You can now access the floppy disk as the Unix directory `/floppy/floppy0/`, just as you would any Unix directory. You can `ls` to see the contents, `cp` and `mv` files to and from the disk, etc.

When you are done, type `eject` and the workstation will eject your disk from its drive.

There are, however, just a few cautions to keep in mind:

- The Unix commands manipulate the files in binary mode. So your text files may wind up with the wrong form of line terminators unless you use `dos2unix` and `unix2dos`.

- The files on the disk will be known via their "short" MSDOS file names. If you have gotten used to the long filenames of Windows 95/NT, you will need to remember that all filenames will be truncated to "8.3 form". For example, the command

  ```
  cp -t longfilename.tar.gz /floppy/floppy
  ```

  will actually result in a file `longfile.gz` appearing on the disk.

7.2.2. Internet:

If you are on a machine that has an Internet connection, you can transfer files to other such machines using the `ftp` program. This would be the means of choice, for example, for exchanging files between the laboratories at the Peninsula Graduate Center and the main Norfolk campus.

All `ftp` file transfers to the ODU CS Dept. network go through a single machine: `ftp.cs.odu.edu`.

As with all internet client programs, the way you actually launch and run the client is determined by the particular software you run on your Internet Service Provider. You might, for example, just need to click on an "ftp" icon and then enter the machine name (`ftp.cs.odu.edu`).

If you are already logged in on a Unix system, give the command

```
ftp ftp.cs.odu.edu
```

Teletechnet students can use `ftp` from the Teletechnet PC's by selecting "Mainframe & TCPIP Access" from the main menu, then "FTP to another host". When prompted for a "hostname/IP address", respond with "`ftp.cs.odu.edu`".

You will then be prompted for your login name and your password. Enter those as usual.[13]

What follows thereafter depends upon your particular `ftp` program. The detailed instructions below are correct for the Unix ftp and for the ftp program used by the Teletechnet PC's. For others, the steps are much the same, but the exact commands may differ.

Your next command should be

```
hash
```

This simply increases the amount of feedback you get about the progress made during file transfers.

Before actually transferring files, you must decide whether to use binary or text file transfer. If you want binary transfers, give the command

```
binary
```

and if you want text transfers, give the command

```
ascii
```

[13]Some classes may provide materials in the "anonymous" area, especially early in the semester when not everyone has their login names and passwords yet. To enter this area, use the login name "anonymous" and for a password give your e-mail address. If you don't have a login name yet, and therefore have no e-mail address, just enter your last name followed by "@cs.odu.edu".

You can switch back and forth between these modes as necessary if you are transferring multiple files, some text and some binary.

Now you can use the commands `cd`, `pwd`, and `ls` to navigate the Unix directory structure as if you were in the shell. To get a file from the CS Unix machine to your local machine, the command is

```
get filename
```

To put a file form your local machine onto the CS Unix machine, the command is

```
put filename
```

Neither the `get` nor `put` commands can include wildcards in the filename, but by changing the commands to `mget` and `mput`, you are allowed to use wild cards.

To end your `ftp` session, the command is

```
quit
```

7.3. Problems and Inconsistencies

If you don't know whether to use binary or text transfer mode, try binary first.

If, however, you have transferred files to a Unix system and discover them to be full of ^M characters (you can see this by viewing the file in `emacs`), this is a sign that you should have used text mode. You can still recover, however, by using the command `dos2unix`:

dos2unix *file*$_1$ *file*$_2$

to produce a new file *file*$_2$ from *file*$_1$ by converting the line ends to the Unix format.

On the other hand, if you have transferred files from a Unix system and find that the received files appear to consist of a single, extremely long line, you can use the command `unix2dos`:

```
unix2dos file1 file2
```

to get a new file *file*$_2$ with ^M^J line terminators that can be transferred to your non-Unix machine instead of the original *file*$_1$.

Finally please note that, although easily transferred files may allow you to do most of the work of a programming assignment on your home PC, do not fall into the trap of believing that you can simply transfer the source code and submit it unchanged to your instructor for grading on the Unix system. Different compilers for the same language often allow a variety of non-standard language extensions (or because of bugs, fail to properly compile standard language constructs). Allow yourself ample time (at least a few days) to port your code from one compiler to another.

8. Using the Internet

If you have an account on the CS Dept. network, then you have access to the Internet through that account. Of course, if you connected to the CS Dept. via the Internet, you obviously have access to it already! But even if you are logged in at a console or via a terminal, you can use the most popular Internet tools.

Console: You can open a terminal session on another machine anywhere on the Internet via telnet (below). Note, however, that you can more easily log into other machines within the CS Dept network using `rlogin` (See Section 9).

You can transfer files to or from another machine anywhere on the Internet via ftp.

Finally, you can "surf" the World Wide Web using the `netscape` command (if you are running X windows). For example, try

```
netscape http://www.cs.odu.edu &
```

to see the CS Dept home page. Or look at

```
netscape http://www.cs.odu.edu/~zeil/zeil.html &
```

for information about me.

Terminal: You can open a terminal session on another machine anywhere on the Internet via telnet. Note, however, that you can more easily log into other machines within the CS Dept network using `rlogin` (See Section 9).

You can transfer files to or from another machine anywhere on the Internet via ftp.

9. Compilers

9.1. Compiling in the Shell

Now that you know how to create and edit files, you can generate new programs. The most commonly used languages in the CS Department at the moment are C++ and C. The most popular C++ and C compilers are `g++` and `gcc`.[14]

The simplest case for each compiler involves compiling a single-file program (or a program in which all files are combined by `#include` statements). For example, use emacs to prepare the following files:

hello.cc

```
#include <iostream.h>
int main ()
{
  cout << "Hello from C++!" << endl;
  return 0;
}
```

[14]Actually, `gcc` and `g++` are the same compiler being invoked with slightly different options.

hello.c

```
#include <stdio.h>
int main ()
{
  printf ("Hello from C!\n");
  return 0;
}
```

To compile and run these, the commands are:

```
g++ -g hello.cpp
a.out
gcc -g hello.c
a.out
```

The compiler generates an executable program called `a.out`. If you don't like that name, you can use the `mv` command to rename it. Alternatively, use a `-o` option to specify the name you would like for the compiled program:

```
g++ -g -o hello1 hello.cpp
hello1
gcc -g -o hello2 hello.c
hello2
```

When you have a program consisting of multiple files to be compiled separately, add a `-c` option

to each compilation. This will cause the compiler to generate a `.o` file instead of an executable. Then invoke the compiler on all the `.o` files together without the `-c` to produce an executable:

```
g++ -g -c file1.cpp
g++ -g -c file2.cpp
g++ -g -c file3.cpp
g++ -g -o programName file1.o file2.o file3.o
```

(Depending upon what else is in the same directory, the last command can often be abbreviated to "`g++ -o programName -g *.o`".) The same procedure works for the `gcc` compiler as well.

Actually, you don't have to type separate compilation commands for each file. You can do the whole thing in one step:

```
g++ -g -o programName file1.cpp file2.cpp file3.cpp
```

But the step-by-step procedure is a good habit to get into. As you begin debugging your code, you are likely to make changes to only one file at a time. If, for example, you find and fix a bug in `file2.cpp`, you need to only recompile that file and relink:

```
g++ -g -c file2.cpp
g++ -g -o programName file1.o file2.o file3.o
```

An even better way to manage multiple source files is to use the `make` command.

Another useful option in these compilers is `-D`. If you add an option `-D`*`name`*`=`*`value`*, then all occurrences of the identifier *name* in the program will be replaced by *value*. This can be useful as a way

of customizing programs without editing them. If you use this option without a value, `-D`*`name`*, then the compiler still notes that *name* has been "defined". This is useful in conjunction with compiler directive `#ifdef`, which causes certain code to be compiled only if a particular name is defined. For example, many programmers will insert debugging output into their code this way:

```
...
x = f(x, y, z);
#ifdef DEBUG
   cout << "the value of X is: " << x << endl;
#endif
y = g(z,x);
...
```

The output statement in this code will be ignored by the compiler unless the option `-DDEBUG` is included in the command line when the compiler is run.[15]

Sometimes your program may need functions from a previously-compiled library. For example, the `sqrt` and other mathmatical functions are kept in the "`m`" library (the filename is actually `libm.a`). To

[15]Zeil's 1st Rule of Debugging: Never remove debugging output. Just make it conditional. If you remove it, you're bound to want it again later.

Zeil's 2nd Rule of Debugging: Never leave your debugging code active when you submit your programs for grading. If the grader is using an automatic program to check the correctness of the output, unexpected output will make your program fail the tests. On the other hand, if the grader is reading the output to check its correctness, wading through extra output really ticks the grader off!

add functions from this library to your program, you would use the "-lm" option. (The "m" in "-lm" is the library name.) This is a linkage option, so it goes at the end of the command:

```
g++ -g -c file1.cpp
g++ -g -c file2.cpp
g++ -g -c file3.cpp
g++ -g -o programName file1.o file2.o file3.o -lm
```

The general form of gcc/g++ commands is

`g++` *compilation-options files linkage-options*

By default, gcc and g++ produce simple text applications — applications designed to run from within a shell (bash). They can, however, produce GUI applications with windows, menus, etc., by using -l to link in the windowing libraries.

Programming windowing code is a fairly involved process. I suggest getting a library that simplifies this process for beginners. The V library is a good choice, and has the additional advantage that code written for use with V can be compiled to produce either Microsoft Windows or Unix X windows programs.

Here is a summary of the most commonly used options for gcc/g++:

Compilation Flags	
-c	compile only, do not link
-o *filename*	Use *filename* as the name of the compiled program
-D*symbol=value*	Define *symbol* during compilation.
-g	Include debugging information in compiled code (required if you want to be able to run the debugger.
-O	Optimize the compiled code (produces smaller, faster programs but takes longer to compile)
-I *directory*	Add *directory* to the list of places searched when a "system" include (`#include <...>`) is encountered.
Linkage Flags	
-L *directory*	Add *directory* to the list of places searched for precompiled libraries.
-l*libname*	Link with the precompiled library `lib`*libname*`.a`

9.2. Compiling in emacs

When your programs contain mistakes, compiling them in the shell can result in large numbers of error messages scrolling by faster than you can read them. For this reason, I find it best to compile from within the `emacs` editor.

Get into `emacs` and call up one of the "hello" programs. Change it so that it contains one or more syntax errors, and save this file. Now give the `emacs` command: `M-x compile`. At the bottom of the screen, you will be asked for the compile command. If the suggested command is not what you want (it won't be, the first time you compile), then type in the proper command just as if you were typing it into the shell. `emacs` will invoke the compiler, showing it's output in a window.

In this case, there should be one or more error messages. The `emacs` command `` C-x ` `` will move you to the source code location of the first error. Each subsequent use of `` C-x ` `` will move you to the next error location in turn, until all the reported error messages have been dealt with.[16]

9.3. Debugging

In the compilation commands given above, the `-g` option causes the compiler to emit information useful for a run-time debugger. The debugger of choice with these compilers is called `gdb`. The easiest way to run `gdb` is, again, from inside `emacs`.

Try creating a longer program in the language of your choice, and compile it to produce an executable program `a.out`. From within `emacs`, look at one of the source code files for that program and then give the command `M-x gdb`.

At the prompt "`Run gdb like this:`", type the program name `a.out`. `emacs` will then launch `gdb`, and eventually you will get the prompt "`(gdb)`" in a window. You can now control `gdb` by typing commands into the `gdb` window. The most important commands are:

[16]Note carefully that the second character in the `` C-x ` `` command is the "backwards" apostrophe, not the regular one.

set args . . . If your program expects arguments on its command lane when it is invoked from the shell, list those arguments in this command before running the program.[17].

break *function* Sets a breakpoint at the entry to the named function (i.e., indicates that you want execution to pause upon entry to that function).

run Starts the program running.

c **C**ontinues execution after it has been paused at a breakpoint.

n Executes the **n**ext statement, then pauses execution. If that statement is a function/procedure call, the entire call is performed before pausing.

You can also do this by giving the `emacs` command `C-C C-N`.

s Like `n`, executes the next statement, but if that statement is a function procedure call, this commands **s**teps into the body of the function/procedure and pauses there.

You can also do this by giving the `emacs` command `C-C C-S`.

p *expression* Prints the value of *expression*, which may include any variables that are visible at the current execution location.

quit Ends your `gdb` session.

[17]These may include redirection of the input and output

In addition to the above, the `emacs` command `C-C <` moves your view of the code up the call stack, allowing you to see the caller of the current procedure/function. The command `C-C >` moves you back down. If you change to a window containing the source code and give the command `C-X space`, a breakpoint will be set at the line of code where the cursor is positioned.

10. More Shell Games

10.1. Redirection and Pipes

One of the interesting ideas that pervades Unix is that many, if not most, programs can be viewed as "filters" or "transforms" that take a stream of text as input and produce an altered stream of text as output. Many Unix commands are designed to perform relatively trivial tasks, perhaps not very useful by themselves, that can be chained together in interesting and useful ways.

The practical consequence of this is that Unix shells devote special attention to a *standard input* stream that forms the main input to most programs/commands, and to a *standard output* stream that forms the main output from most programs/commands.[18] The shell attempts to make it easy either to *redirect* one of these standard streams to a file or to *pipe* the standard output stream of one program into the standard input of another.

For example, the program `wc` (for **w**ord **c**ount) reads text from its input stream and produces as its output stream three number indicating the number of lines, words, and characters that it saw. You could invoke this directly:

```
wc
Hello.
How are you?
^D
```

[18]There is actually a second output stream supported by many programs, the *standard error* stream, used for writing error/debugging messages.

in which case, you would see as output:

2 4 20

For this to be very useful, however, we need to make it accept a file as input. This is done by using the < operator in the shell. Think of the < as an arrow indicating data flowing towards the command from a filename:

```
wc < hello.c
```

where hello.c is the file from Section 9.1, produces the output

6 13 80

On the output end, the shell operator > directs the standard output into a file (again, think of this as an arrow indicating data flowing into a filename from the command):

```
wc < hello.p > hello.wc
```

produces no output on the screen, but creates a file called hello.wc. That file will contain the output

6 13 80

of the wc command.

The output redirection operator has a couple of important variants. First, the shell generally does not allow you to redirect into an existing file. If you give the command

```
wc < hello.c > hello.wc
```

a second time, the shell will refuse to perform the command. You can force the shell to delete an existing file and create a new one for redirection by changing the `>` to `>!`.

Second, sometimes we would like to add output to the end of an existing file instead of replacing that file. This is done with the operator `>>`. So the code sequence

```
wc < hello.c >! hello.wc
wc < hello.c >> hello.wc
```

would result in a file `hello.wc` with contents

6 13 80
6 13 80

regardless of whether `hello.wc` had existed previously.

To pipe the output of one command into the input of another, use the shell operator `|`. A common example of a pipe is to take a command that may have a large amount of output and to pipe it through `more` to facilitate viewing. For example, try

```
ls /bin | more
```

As you gain facility with a greater variety of Unix text manipulation commands, you will find that redirection and pipes can be a powerful combination. For example, suppose that you have written program `myprog` that emits a great deal of output, among which might be some error messages starting with the phrase "`ERROR:`". If you wanted to read only the error messages, you could, of course, just view *all* the output, watching for the occasional error message:

```
myprog | more
```

But if the program produces a lot of output, this will quickly become tedious. However, the program `grep` searches its input stream for a given character string,[19] emitting at its output only the lines containing that string. By piping through `grep`, we can limit the output to the part we really want to see:

```
myprog | grep "ERROR:" | more
```

10.2. Scripts

You can put any sequence of Unix commands into a file and turn that file into a command. Such a file is called a *script*. For example, suppose that you are working on a program `myprog` and have several files of test data that you run through it each time you make a change. Create a file `dotest1` with the following lines:

```
myprog < test1.dat > test1.dat.out
myprog < test2.dat > test2.dat.out
myprog < test3.dat > test3.dat.out
myprog < test4.dat > test4.dat.out
myprog < test5.dat > test5.dat.out
```

Now, you can't execute `dotest1`, because you don't have execute permission. (Do `ls -l dotest1` to see this.) So use the `chmod` command to add execute permission:

[19] Actually, `grep` is far more powerful, enabling you to search for strings matching elaborate patterns.

```
chmod u+x dotest1
```

Now you can execute `dotest1` by simply typing

```
dotest1
```

Most shells provide special facilities for use in scripts. Since these differ from one shell to another, it's a good idea to tell Unix which shell to use when running the script. You do this by placing the command `#!/bin/csh` in the first line of the script.[20]

One such special feature is the use of the symbol `$k` to stand for the kth argument given to the script. For example, suppose that we wanted the ability to use a different set of test files each time we used the test script. One approach would be to create a script `dotest2`, as follows:

```
#!/bin/csh
myprog < $1 > $1.out
myprog < $2 > $2.out
myprog < $3 > $3.out
myprog < $4 > $4.out
myprog < $5 > $5.out
```

After the appropriate `chmod`, this could then be invoked as

```
dotest2 test1.dat test2.dat test3.dat test4.dat test5.dat
```

[20] In fact, you can list any program there, not just `/bin/csh`, and Unix will use that program to process the remainder of the lines in the script.

or with any other five test files. Of course, if we want to test with only four files, or with six files, we're out of luck. It would be nicer if we could have the script loop through as many files as we list on the command line each time we run it. Such a script begins to sound more like a program, and in fact most shells provide loops, if's, and other programming language-like statements. Here, for example, is the script `dotest3` that will process each argument in turn, however many there are:

```
#!/bin/csh
foreach file ($*)
  myprog < $file > $file.out
end
```

Here we use another special feature, the use of `$` to indicate that we want to retrieve a value from a variable, in this case the variable `file` which is assigned by the `foreach` loop. Also, we use `$*`, which denotes the entire list of arguments given to the script.

After the appropriate `chmod`, this script could then be invoked as

```
dotest3 test1.dat test2.dat test3.dat test4.dat test5.dat test6.dat
```

or perhaps as easily as

```
dotest3 test*.dat
```

Either way, the `foreach` statement will loop though all files named in the argument list, setting `file` to each file name in turn.

11. Project Management with Make

When you begin to develop projects that involve multiple files that need to be compiled or otherwise processed, keeping them all up-to-date can be a problem. Even more of a problem is passing them on to someone else (e.g., your instructor) and expecting them to know what to do to build your project from the source code.

The Unix program `make` is designed to simplify such project management. In a *makefile*, you record the steps necessary to build both the final file (e.g., your executable program) and each intermediate file (e.g., the `.o` files produced by compiling a single source code file).

We say that a file `file1` *depends upon* a second file `file2` if the `file2` is used as input to some command used to produce `file1`.

When the `make` program is run, it then checks to be sure that all of the needed files exist, and that each needed file has been updated more recently than all of the files it depends upon. The key bits of information in a makefile, therefore are

- For each file, a list of other files it depends upon, and
- The command used to produce the dependent file from the files it depends upon.

A makefile may also include various macros/abbreviations designed to simplify the task of dealing with many instances of the same commands or files.

Suppose that we are engaged in a project to produce 2 programs, `progA` and `progB`. `progA` is produced by compiling files `utilities.c`, `progA1.cpp`, and `progA2.cpp` and linking together the resulting `.o` files. Program `progB` is produced by compiling file `utilities.c` and `progB1.cpp`

and linking together the resulting `.o` files. All of the `.c` and `.cpp` files have `#include` statements for a file `utilities.h`. Also, both of the `.cpp` files have an `#include` statement for a file `progA1.h`.

Here is a makefile for this project. This file should reside in the project directory, and should be called "`Makefile`" or "`makefile`".

```
# Macro definitions for "standard" language compilations
#
#  First, define special compilation flags. These may change when
#  we're done testing and debugging.
FLAGS=-g -DDEBUG
#
#  The following is "boilerplate" to set up the stan-
dard compilation
#  commands:
.SUFFIXES:
.SUFFIXES: .cpp .c .cpp .h .o
.c.o: ; gcc $(FLAGS) -c $*.c
.cc.o: ; g++ $(FLAGS) -c $*.cc
.cpp.o: ; g++ $(FLAGS) -c $*.cpp
#
# Targets:
#
progA: utilities.o progA1.o progA2.o
        g++ $(FLAGS) utilities.o progA1.o progA2.o
        mv a.out progA

progB: utilities.o progB1.o
        g++ $(FLAGS) utilities.o progB1.o
        mv a.out prog
```

In the "SUFFIXES" area, standard commands are defined for producing a `.o` file from a `.c`, `.cc`, or `.cpp` file. Of course these standard commands simply invoke the C or C++ compilers.

The key information is in the area Labeled "Targets". Each target begins with a single line containing the name of the file to produce, a colon, and then a list of all files that serve as inputs to the commands that produce the file. Following that are any number of command lines that give the Unix commands to actually produce the file. Each command line starts with a "Tab" character (invisible in this listing). Command lines are not needed if the standard commands form the "Suffixes" area can be used to build the desired file.

Suppose that, with just this `Makefile` and the various source code files in your directory, you issued the command `make progB`. `make` reads the `Makefile` and notes that `progB` depends upon `utilities.o` and `progB1.o`. Since neither of these files exists, `make` sets out to create them. `utilities.o` depends upon `utilities.c` and `utilities.h`. Since these files exist and do not themselves depend upon anything else, `make` will issue the command to create `utilities.o` from them. This command is the "standard" command for making a `.o` file from a `.c` file:

```
gcc -g -DDEBUG -c utilities.c
```

Next `make` looks at `progB1.o`. It depends upon `progB1.cpp` which exists and does not depend upon anything else. So `make` uses the standard command for C++ files:

```
g++ -g -DDEBUG -c progB1.cpp
```

Now that both `.o` files have been created, `make` proceeds to build its main target, `progB`, using the command lines provided for that purpose:

```
g++ -g -DDEBUG utilities.o progB1.o
```

and the progB program has been created.

Now suppose that we immediately give the command "make progA" (or just "make", since by default make builds the first target when none is explicitly given). Then the following commands would be performed:

```
g++ -g -DDEBUG -c progA1.cpp
g++ -g -DDEBUG -c progA2.cpp
g++ -g -DDEBUG utilities.o progA1.o progA2.o
mv a.out progA
```

Note that utilities.c is not recompiled, because make would notice that utilities.o already exists and was created more recently than the last time when either utilities.c or utilities.h was changed.

Now, creating a makefile may seem like a lot of trouble the first time that you want to compile your program. The payoff comes while you are testing and debugging, and find yourself making changes to two or three files and then needing to recompile. Which files do you really need to recompile? It can be hard to remember some times, and the errors resulting from an incorrect guess may be hard to understand. make eliminates this problem (as well as just being easier to type than a whole series of recompilation commands). (This is why, when you give the M-x compile command in emacs, the default compilation command is "make" rather than a direct use of any particular compiler.)

If you want to test your makefile without actually performing the commands, add a -n option to your command (e.g., make -n progB) and make will simply list the commands it would issue without

actually doing any of them.

Most of the details of generating a makefile can be automated. Although the details are beyond the scope of this tutorial, you can obtain my "self-constructing" Makefile. To use it, copy it into your working directory where you keep the source code files for any single program. Your copy must be named "Makefile". Edit your copy of the file to supply the appropriate program name, list of source code files needed for that program, and to indicate whether the final step (linking) should be done with the `C` (`gcc`) or `C++` (`g++`) compiler.

Now you can compile your program by saying `make`.

As you continue to work with your code, just remember to keep the `OBJS` list in the Makefile up to date.

12. Where to Go From Here?

We've only scratched the surface in this document. There are many more useful commands and programs available on the CS Department Unix machines, and many of the commands that we have covered have additional options that have not been mentioned here. Remember that you can use the Unix `man` command to call up documentation on any command. The appendix lists a number of additional commands that you may want to check out as you become more familiar with Unix.

A. Unix Command Summary

[] denotes options
{} denotes required argument
ˆ denotes control key (depress while typing listed letter).
. . . indicates that command has many options. Use `man` to learn about this command.

awk ...	a pattern matching and text manipulation language.
bg	puts process in background after ^z
cal [month] {year}	displays calendar for that month
cal	displays calendar for current month
cat {filename}	displays filename
cat [options]	
-b	number the lines, as -n, but omit the line numbers from blank lines.
-n	precede each line output with its line number.
cd [directoryname]	changes to directoryname, no argument indicates home directory
cd ..	changes to directory one above current
cp {file1} {file2}	copy file1 naming it file2
mv {file1} {file2 or directoryname}	move files or rename them
date	displays date
diff {file1} {file2}	compares two files, reporting any differences
echo	repeats line; useful when using $*$ and ? in filenames
fg	puts first command in background into the foreground
grep {pattern} {filename}	find pattern in filename
head -n	Prints the first n lines of its input, ignoring the rest

kill [option] {process id #}	stop a process
-9	kill no matter what: can be DANGEROUS
logout	end session, must be in login shell
lpr {filename}	send file to printer for printing
lprm {request} {userid}	remove a file from the printer queue
lpq	check status of printer and jobs
ls [options]	list files
-l	long form
-a	all files, including .files
-g	groups
mail	see "man mail" and /home/public/help for more information
mush	shell for mail, see "man mush" for more information
man [option] {command}	display manual page for command
mesg {y or n}	enable/disable messages to terminal
mkdir {directoryname}	create a directory
more {filename}	list filename one screen at a time
nroff,troff . . .	text formatting programs
phones	gives instructions for using modems
hours	gives lab hours
ps	show processes you are running
pwd	print working directory
rm [option] {filename}	remove files
-i	interactive
-r	recursive (use with caution)

rmdir {directoryname}	remove directoryname
rwho	who is on your current network
X	X windows environment
openwin	openwindows environment
sed . . .	A non-interactive editor, useful for writing scripts that involve string replacements, line deletions, etc.
sort [options] {filename}	sort filename
-b	ignore spaces and tabs
-f	sort upper- and lower-case together
-r	reverse the sorting order
-o filename	save the output of sort in filename
-t letter	set field separator to letter
-u	remove duplicate lines
spell {filename}	check spelling of filename
tail -n	Prints the last n lines of its input,
tr	Replaces/deletes characters ignoring the rest
wc [options] {filename}	count words, lines, and characters
-c	characters only
-l	number of lines only
-w	number of words only
who	who is running remote logins on your machine

write {user}	write message to user, ^d to quit
yppasswd	change password, follow prompts

<table>
<tr><td>?</td><td>matches any single character in a filename</td></tr>
<tr><td>*</td><td>matches any number of characters in a filename (or no characters)</td></tr>
<tr><td>&</td><td>puts command in background when appended to a command line</td></tr>
<tr><td>|</td><td>pipe, connects output of one command with input of another</td></tr>
<tr><td>></td><td>redirects output of a command to a file, erasing current contents of a file</td></tr>
<tr><td>>></td><td>appends output of a command to an existing file</td></tr>
<tr><td><</td><td>uses the file as an input for a command</td></tr>
<tr><td>^c</td><td>aborts process (useful when ”hung-up”)</td></tr>
<tr><td>^d</td><td>stops a process or signals ”done” on console, indicates logout</td></tr>
</table>

B. Emacs Command Summary

EMACS command summary

C. Linking to this Document

Instructors interested in linking directly to specific sections of this document may do so by appending the appropriate anchor name to the URL of this documents (e.g.,

`http://www.cs.odu.edu/~zeil/unix/unix.pdf#loggingin`)

The defined anchors are listed in the tables below.

Anchor	Section	Page	Section Title
theBasics	2	4	The Basics
loggingin	2.1	4	Logging In
termtypes	2.1.4	9	Setting Your Terminal Type
xon	1	5	Other Unix/Linux Machines
telnet	5	7	Other machines
unixFiles	2.2	12	The Unix File System
shellgamesi	2.3	16	Shell Games: Typing Unix Commands
basicUnix	2.4	19	Some Basic Unix Commands
fileprot	2.5	24	File Protection
emacs	3	31	Editing Text Files
Xwin	4	33	X Windows
custom	5	37	Customizing Your Unix Environment

Anchor	Section	Page	Section Title
mail	6	41	Using Electronic Mail
mailsend	6.2.2	47	Sending
forwarding	6.3	50	Forwarding Addresses
filetransfer	7	52	File Transfer
xfermode	7.1	52	Text versus Binary Transfers
ftp	7.2.2	54	Internet:
dostounix	7.3	56	Problems and Inconsistencies
unixtodos	7.3	57	Problems and Inconsistencies
gccCompilation	9	60	Compilers
compshell	9.1	60	Compiling in the Shell
emacscompile	9.2	65	Compiling in emacs
debugging	9.3	66	Debugging
shellgamesii	10	69	More Shell Games
redirect	10.1	69	Redirection and Pipes
make	11	75	Project Management with Make
commands	A	82	Unix Command Summary
linking	C	88	Linking to this Document

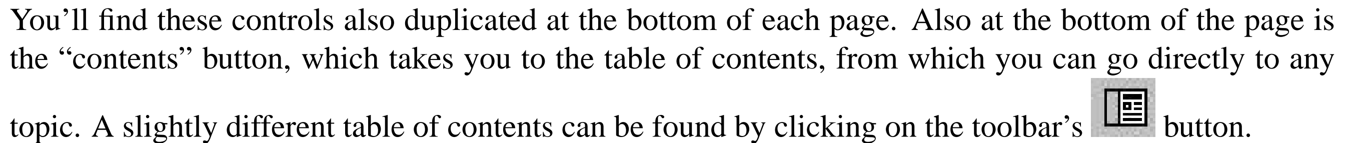
D. Navigating this Document

You should see a toolbar at the top of the document. The critical controls on this toolbar are:

► Moves you to the next page. You may also be able to do this with the “page down” key.

◄ Moves you to the previous page. You may also be able to do this with the “page up” key.

⬅ Moves you backwards through the most recently visited pages. (Use this to return to the title page, if you clicked on a link to get there).

You’ll find these controls also duplicated at the bottom of each page. Also at the bottom of the page is the “contents” button, which takes you to the table of contents, from which you can go directly to any topic. A slightly different table of contents can be found by clicking on the toolbar’s  button.

A final note is that if this document appears inside a web browser window, you may find it easier to read if you go to a “full screen” mode. In Internet Explorer, hit the F11 key to toggle to and from this mode. In Netscape, click on the small textured rectangles on the left of each toolbar.

E. Table of Contents

1 Working on the CS Dept Network **2**

2 The Basics **4**

2.1 Logging In 4

2.1.1 Making a Connection 4

2.1.2 Logging In 7

2.1.3 You're logged in - What will you see? 7

2.1.4 Setting Your Terminal Type 9

2.1.5 Changing Your Password 10

2.2 The Unix File System 12

2.3 Shell Games: Typing Unix Commands 16

2.4 Some Basic Unix Commands 19

2.5 File Protection 24

2.5.1 Protections 24

2.5.2 chmod 26

2.5.3 Beware the umask! 27

2.5.4 Planning for Protection 29

2.6 Getting Help 30

3 Editing Text Files **31**

4 X Windows **33**
4.1 X Window Managers 33
4.2 Running X 33
4.3 Working in X 34

5 Customizing Your Unix Environment **37**

6 Using Electronic Mail **41**
6.1 E-Mail addresses 41
6.2 E-Mail Programs 42
6.2.1 The PINE E-mail program 43
6.2.2 The Unix mail command 47
6.3 Forwarding Addresses 50

7 File Transfer **52**
7.1 Text versus Binary Transfers 52
7.2 Transferring Files 53
7.2.1 At the console: 53
7.2.2 Internet: 54
7.3 Problems and Inconsistencies 56

8 Using the Internet **58**

9 Compilers **60**
9.1 Compiling in the Shell 60
9.2 Compiling in emacs 65
9.3 Debugging 66

10 More Shell Games **69**
10.1 Redirection and Pipes 69
10.2 Scripts 72

11 Project Management with Make **75**

12 Where to Go From Here? **81**

A Unix Command Summary **82**

B Emacs Command Summary **87**

C Linking to this Document **88**

D Navigating this Document **90**