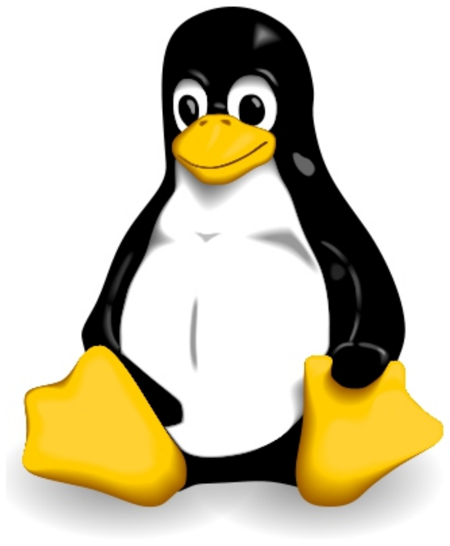
GNU/Linux Lab Manual

for

CSE220 Programming for Computer Engineering

Kevin R. Burger



Arizona State University :: School of Computing,
Informatics, and Decision Systems Engineering

Rev 1.8(b)
4 Feb 2013

Author Information
Kevin R. Burger
Computer Science and Engineering
School of Computing, Informatics, and Decision Systems Engineering
Fulton Schools of Engineering, Arizona State University
Tempe, AZ 85287-8809
Email: burgerk [@-sign] asu.edu, Web: http://kevin.floorsoup.com/

Contents

1 Introduction to Unix and GNU/Linux

1.1 Short History of Unix

A good history of the Unix operating system is available at Wikipedia[1]. UNIX was an operating system[2] originally written at AT&T Bell Labs in the late 1960's by Ken Thompson[3], Dennis Ritchie[4], and others. Thompson had been working on an earlier operating system project named Multics (Multiplexed Information and Computing Service) in conjunction with MIT and General Electric. While working on the project, Thompson had written a game called Space Travel for the GE mainframe being used for Multics development. After AT&T Bell Labs withdrew from the Multics project, Thompson began porting his Space Travel game to a DEC PDP-7 minicomputer at Bell Labs, and in the process began work on a new operating system he named Unics (for Uniplexed Information and Computing Service); the spelling was later changed to Unix and when it was owned by AT&T was written in all capitals as UNIX.

Over time, Thompson, Ritchie, and others at Bell Labs continued developing UNIX. In 1973 the entire operating system was rewritten in Ritchie's new C programming language, and during this time AT&T began giving the operating system to universities, research institutions, and the US government under licenses, which included all source code. In 1982, when AT&T realized that they had an OS on their hands that was worth something, they changed their licensing model and began selling UNIX as a commercial product without source code.

Meanwhile, at the University of California, Berkeley, Bill Joy and others had begun to develop BSD[5] Unix as an alternative to the now-commercialized AT&T Unix. BSD Unix went on to become immensely popular, especially in the academic world, and Bill Joy eventually left Berkeley to go to work for a small startup named Sun Microsystems where he created the SunOS version of Unix which eventually became Sun Solaris.

Throughout the next two decades, many companies and organizations produced their own versions of Unix, usually under license from AT&T or from the Berkeley branch. This led to a proliferation of differing operating systems that were called Unix. For example, at one time, one could acquire "Unix" from various entities such as: AT&T (System V), UC-Berkeley (BSD Unix), IBM (AIX), SGI (IRIX), Microsoft (Xenix), SCO, HP (HP-UX), and others. During the late 1980's, it was thought that Unix would become the dominant operating system among PC users, but the divisions among the participants and the

1 http://en.wikipedia.org/wiki/Unix

2 An operating system is software which manages the resources in a computer system.

3 http://www.cs.bell-labs.com/who/ken

4 http://cm.bell-labs.com/cm/cs/who/dmr. Dennis Ritchie is the creator of the C programming language. It was based on an earlier programming language named B that was created by Ken Thompson.

5 Berkeley Software Distribution.

wrangling for control of Unix created an opportunity for Microsoft Windows NT to fill the gap in the market, and for Microsoft Windows to, instead, dominate the industry[6]. Eventually, while getting out of the computer business, AT&T washed their hands of Unix and sold all rights to Novell, which eventually transferred control to the Open Group. The Open Group is not a company, but rather is an industry consortium[7] which sets standards for Unix. The Open Group published the Single Unix Specification (SUS) in 2002 which is a family of standards that defines what Unix is, is not, and what OS's qualify for the name. The SUS is now maintained by the Austin Group[8] for the Open Group.

1.2 The GNU Project and the Linux Kernel

In 1983, after AT&T "closed" Unix by commercializing it and restricting access to the source code, Richard Stallman founded the GNU project[9], whose goal was to provide a free, open-source Unix-like operating system which would replace Unix. The project was named GNU for GNU's Not Unix. However, the most import part of the operating system, the GNU kernel[10], was not completed very quickly, and is in fact, not entirely completed yet[11] (it is under active development).

In 1991, Linus Torvalds, a student at the University of Helsinki, Finland, started work on his own Unix-like kernel. It was based on a published, open-source operating system called Minix which was created at the University of Vrije, Netherlands, by Andrew Tanenbaum as an OS for teaching operating system principles to students; MINIX was written for the IBM PC. Linus began making modifications to MINIX, and his work eventually led to the creation of the Linux kernel. For whatever reason, and with much luck, Linux became extremely popular and the Linux kernel[12] is now commonly combined with many of the tools and much of the GNU source code from the GNU project to form an operating system that is called "Linux."

However, Richard Stallman, the founder of the GNU project, is adamant[13] that the moniker "Linux" is inappropriate as most of what is called the "Linux" operating system is, in fact, GNU code (the "Linux" kernel actually only comprises around 1.5–3% of the total Linux source code). Stallman expects the OS to be called GNU/Linux, but nonetheless, the moniker "Linux" is well-entrenched and is not going away any time soon.

---

6 Supposedly. Sigh.
7 Members include IBM, HP, and other major Unix players.
8 http://www.opengroup.org/austin
9 Pronounced "guh new". http://www.gnu.org/
10 Named *Hurd*. https://www.gnu.org/software/hurd/hurd.html
11 http://en.wikipedia.org/wiki/Gnu_hurd
12 http://en.wikipedia.org/wiki/Linux_kernel; http://www.kernel.org
13 http://www.gnu.org/gnu/ linux-and-gnu.html

2 Obtaining Access to a GNU/Linux System

2.1 Introduction

There are a plethora of ways you can obtain access to GNU/Linux (hereafter referred to as simply Linux), Unix, or a Unix-like environment. So many, in fact, that this document makes no attempt to describe them all. Rather, I will discuss the ways which are most convenient for us in CSE220. The methods include: (1) Running Cygwin on Windows; (2) Connecting to general.asu.edu using SSH; (3) Running Linux in a Windows-based[14] virtual machine using VMware Player; (4) Using the computers in the BYENG 214 computer lab; or (5) Buying a Mac.

2.2 First—Choosing a Distro

A **distro** is a Linux **distribution** consisting of files which you download or acquire on alternate media (e.g., CD, DVD, USB flash drive). The distro includes all the files you need to install and configure a working Linux system. There are hundreds of distros and the best place to research them is Distrowatch[15] which has information on numerous distros and links to websites where the files can be downloaded or acquired.

Distrowatch maintains a page hit ranking of which Linux distro pages are receiving the most page hits; this list will give you a rough idea of the popularity of each distro. On 1 Jan 2013, the top ten distros on the page hit ranking list were[16]: (1) Mint; (2) Mageia; (3) Ubuntu; (4) Fedora; (5) openSUSE; (6) Debian; (7) Arch; (8) PCLinuxOS; (9) Zorin; and (10) CentOS. The big winner this year seems to be Mageia moving up to number 2 from out of nowhere.

Some of these distros have been around for a long time, e.g., Debian (1996), openSUSE (1994, originally just SUSE, the oldest extant commercial distribution), Mageia is based on Mandriva (1998, originally named Mandrake[17]), Slackware (1993, the oldest extant noncommercial distro), FreeBSD (1993, based on UC Berkeley's version of UNIX called BSD), Red Hat (1994), and KNOPPIX (2000, one of the first Live CD's). The others are relatively new, e.g., Ubuntu (based on Debian) was first released in 2004 and has quickly become one of the top three most popular distributions; Linux Mint (based on Ubuntu) started in 2006 and has also become popular very quickly, primarily because of its reputation for being easy to install and main—especially for people with little Linux knowledge.

[14] If you own a Mac, I'm sorry. No, seriously, Apple makes quality products, but they're took expensive for my tastes.

[15] http://www.distrowatch.com

[16] Exactly one year ago, it was: (1) Mint; (2) Ubuntu; (3) Fedora; (4) openSUSE; (5) Debian; (6) Arch; (7) CentOS; (8) PCLinuxOS; (9) Puppy; and (10) Mageia.

[17] Until 2004 when they were sued and lost a court case to *King Features Syndicate*. Apparently, King Features own the rights to a cartoon character named Mandrake the Magician, which explains perfectly well how an educated person could possibly confuse an actual computer operating system called Mandrake Linux with a fictional comic strip character named Mandrake the Magician.

Personally, I like Debian and have been using it for several years. If you're new to Linux, I would recommend either Ubuntu or Mint, but Zorin—designed for newbies—also seems to be becoming quite popular. If you really like to get your hands dirty, then Slackware and Gentoo are designed especially for you. The true masochist will want to check out [18].

2.3 Second—Linux Desktops

A **desktop**[19] essentially refers to a graphical user interface (GUI) and applications which permit the user of the OS to interact with the system (the OS and applications) graphically using familiar keyboard and mouse commands. Unlike in the Windows world, where you essentially end up with the Windows desktop when you install Windows, in the Linux world there are different desktops that one can install and use. Historically, the two most popular were GNOME[20] (pronounced either "guh nome" or simply "nome") and KDE[21].

GNOME is part of the GNU project and is based on the GTK+ graphical user interface toolkit/library; it was first released in 1996, and since it is now 17 years old, it is a quite mature environment. GNOME 2 was widely supported by all the major Linux distros, but with the release of GNOME 3, fan support has dropped, as the familiar desktop metaphor has been replaced by something called the Gnome Shell[22]—which many former GNOME users are rebelling against, for various reasons[23]. Nonetheless, many popular distros ship with GNOME 3 including Fedora, Mint 12, openSUSE 12.1, Arch, OpenBSD, and Debian.

KDE is similar in functionality to GNOME and was first released a couple of years later in 1998. It is based on a different GUI toolkit called Qt (pronounced "cute") which was owned by mobile phone maker Nokia until they sold it in 2011 to Digia, Qt[24]. I have al-

[18] http://www.linuxfromscratch.org
[19] http://en.wikipedia.org/wiki/Desktop_Linux
[20] http://www.gnome.org
[21] http://www.kde.org
[22] https://en.wikipedia.org/wiki/GNOME_Shell
[23] In a nutshell, I think it boils down to the fact that Gnome Shell is a drastic change from the former Gnome desktop and many users, me included, don't like change unless there is an obvious payoff. For example, I no longer use MS-DOS 5.0, the command shell, and text-based programs, and my resistance to change for change's sake is also why I will never upgrade to continuing-to-get-dumber-with-every-release Windows 8.
[24] If you haven't heard, Nokia is on the verge of obsolescence and bankruptcy due to the fact that *nobody* buys Nokia phones anymore, so I suppose the sell-off might be related to that. Of course, it's a race between Research in Motion (aka the "Blackberry" people) and Nokia to see who goes under first. I'm betting on Nokia losing first considering that they just sold their world headquarters for 170 million eu and will now be leasing it back from the buyer. A word of career advice: if you're ever working for a company and they start selling IP (intellectual property) and their world headquarters to raise funds for a comeback, I would begin spending most of my days at work browsing Monster.com.

ways preferred KDE to GNOME—although KDE is a RAM-hog[25]—but on my To Do list is a task to check out some of the newer desktops with which I am unfamiliar.

The most popular newer desktops are Cinnamon, LXDE[26] (Lightweight X11 Desktop Environment), MATE, Unity, Razor-qt, and Xfce[27]. Cinnamon is a fork of GNOME 2 by the Linux Mint community because they did not see GNOME 3 fitting into their vision of Linux Mint. Both LXDE and Xfce (based on the GTK+ toolkit, which also used by GNOME) are designed to be "light" desktop environments—to consume less memory and fewer CPU cycles—for older computers and netbooks with less powerful processors and smaller memory than newer, more powerful computers. Razor-qt is a similar effort, but is based on the Qt toolkit (used by KDE). MATE is another fork of GNOME 2 which is supported by many major distributions (Mint, Arch, Fedora, Debian, openSUSE, and Ubuntu). Unity is an Ubuntu-developed Gnome Shell replacement. It's not widely supported outside Ubuntu, but if you're an Ubuntu user, you might want to check it out.

2.4 Running Cygwin on Windows

Cygwin[28] is a collection of tools which provide a Linux-like environment for Windows users. More technically, it consists of a DLL (dynamic link layer; named cygwin1.dll) which acts as a Linux API (Application Programming Interface) emulation layer. When a program makes a Linux operating system call by calling one of the functions in a system library, this API layer traps the call and transforms it into a functionally-equivalent Windows system call. From the program's perspective, it appears to be running on Linux, even though it is actually executing on a Windows machine.

In addition to the Cygwin DLL, there are a large number of native Linux applications which have been ported to Cygwin (native Linux applications will not run directly on Cygwin, but rather, must be modified in order to run with Cygwin at the API layer). To use Cygwin, you would want to install—at a minimum—the following packages,

Category	**Packages to Install**
Base	Install everything
Devel	binutils, ddd, gcc-core, gcc, g++, gdb, make, mercurial
Doc	cygwin-doc, man
Shells	bash
Utils	bzip2, cygutils

If you want to throw a GUI on top of Cygwin, specifically X, see [29]. I use Cygwin quite a bit. I normally work in Windows, but I often want to write some C or C++ code and for

[25] http://www.phoronix.com/scan.php?page=article&item=linux_desktop_vitals&num=1
[26] http://lxde.org
[27] http://www.xfce.org
[28] http://www.cygwin.com
[29] http://x.cygwin.com/

that I prefer to work in Linux. With Cygwin, I can just run it and do my C/C++ development as if I were booted into Linux even though I am still in Windows. It is really convenient and works flawlessly most of the time. You can follow these steps to install Cygwin on your Windows computer,

1. Using your favorite browser[30], navigate to the Cygwin home page.
2. Look for the link labeled **setup.exe** and click on it. You will download an installation file named (drum roll please...) setup.exe.
3. Once setup is downloaded, run it.
4. In the first dialog that appears, simply click the Next button.
5. In the next dialog, click Next.
6. It will work best if you let Cygwin install itself where it wants to, which is in C:\cygwin. You can install it somewhere else, but whatever you do, **do not** install Cygwin in a folder containing spaces in the name, e.g., in C:\Program Files. Unix and Linux do not like spaces in file/directory names, so if you do install it in the aforementioned C:\Program Files, Cygwin will, most likely, barf all over the place. So don't. Click the Next button.
7. In the next dialog, click Next.
8. In the next dialog, click Next.
9. In the next dialog, you have to choose a mirror site from which to download the packages you are going to install. The idea is that you're supposed to choose a mirror which is physically close to you because in theory, the files will download faster. I always use ftp://ftp.is.-co.za. I think that is in South Africa somewhere. So, choose a mirror close to you and then click Next.
10. In the next dialog, a configuration file named setup.bz2 will be downloaded. This is a bzip2-compressed file[31] which contains a list of the various packages that are available. Just wait for the download to finish.
11. In the next dialog, you will select the packages to download and install. First, click on the + button beside the Base category to expand that category. When you download the packages, you have the option of downloading just the binary (i.e., the actual program), the source code, or both. You really only need the binaries, so in the list of packages, for any packages which are marked Skip, click on the word Skip until the box in the Bin? column is checked. That is, we wish to install every binary in the Base package.
12. Now, click on the + button beside the Devel category. If any packages are already selected, just leave them alone. Make sure to select the following packages for installation: **binutils**, **ddd** (a graphical front-end for the GDB debugger), **gcc-core** (the GNU C compiler), **gcc-g++** (the GNU C++ compiler), **gdb** (the GNU debugger), **make** (an extremely useful and common utility for automating software builds), and **mercurial** (a revision control system).
13. Now, click on the + button beside the Doc category. If any packages are already selected, just leave them alone. Make sure to select the following packages for installation: **cyg-**

[30] You could use a browser that is not your favorite, e.g., Internet Exploder, but that would be strange, now wouldn't it?

[31] **bzip2** is a compression algorithm similar to **zip**. It is used a lot in the Linux world because it often compresses files more efficiently than zip.

win-doc and **man** (man is short for manual and is the historical way that Unix commands were documented for the user).

14. Now, click on the + button beside the Libs category. If any packages are already selected, just leave them alone. Make sure to select the following packages for installation: **libncurses10** (a library for writing text-based programs with menus, dialogs, buttons, etc.)
15. Now, click on the + button beside the Shells category. If any packages are already selected, just leave them alone. Make sure to select the following packages for installation: **bash** (this is the default shell you will run when using Cygwin. You can run a different shell, but I recommend bash for this course).
16. Now, click on the + button beside the System category. If any packages are already selected, just leave them alone. Make sure to select the following packages for installation: **man**.
17. Now, click on the + button beside the Utils category. If any packages are already selected, just leave them alone. Make sure to select the following packages for installation: **bzip2**, **cygutils** (a collection of utility programs for administering your Cygwin installation), **diffutils**, **dos2unix** (useful for converting text files from MS-DOS format to Unix format and vice versa), and **file**.
18. Now that you have selected all of the required packages, click the Next button. Cygwin may notify you that based on the packages you have selected, certain other packages will also have to be installed. If so, sigh, mutter, and just say yes, and the download process will begin. This might take a while. You might see what's on TV or call your mother; I'm sure she would love to hear from you.
19. After the download completes, in the next dialog, select both check boxes to create a Cygwin icon on the desktop and in the Windows Start Menu. Then click Finish.
20. Go to your desktop, and you should find an icon labeled Cygwin. If not, you can create the shortcut to the file C:\cygwin\cygwin.bat.
21. Before running Cygwin, you may wish to configure the window appearance. Right-click on the Cygwin icon on the desktop and select Properties from the popup window. Click on the Shortcut tab. Beside Run: select Maximized.
22. Click on the Font tab. I like the Lucida Console font. Click on it, and then select 12 for the Size.
23. Click on the Layout tab. Enter 132 and 1024 for the Screen Buffer Size (width and height). Enter 132 and 50 for the Window Size. Enter 0 and 0 for the Window Position. Unselect the Let system position window check box.
24. Click on the Colors tab. You can change the background and foreground colors here if you wish.
25. Finally, click the OK button to close the Properties dialog.
26. Now run Cygwin by double-clicking on the icon on your desktop.
27. The Cygwin bash shell terminal window will open. From now on, it's like you're logged into a *nix system. To close the window, type logout at the shell prompt, or simply hit the Ctrl+D key combination.

2.5 Connecting to General (general.asu.edu) Using SSH

The ASU University Technology Office (UTO) maintains two Linux clusters running, well, uh, Linux (RedHat distros to be precise). As an ASU student, you can obtain a free account on these clusters. Follow these steps,

1. Using your browser, navigate to the ASURITE Self Management system[32] and log in using your ASURITE credentials.
2. Once logged in, you will see a list of services that you can subscribe to. Select the radio buttons beside General/ Statistics UNIX clusters (general.asu.edu and stats.asu.edu). Then click the appropriate button to subscribe.
3. It will take the system anywhere from one minute to perhaps an hour or two to establish your accounts. When I first did this, it took around one hour. You can now log out of the ASURITE Self Management system.

2.5.1 Installing PuTTY on Windows

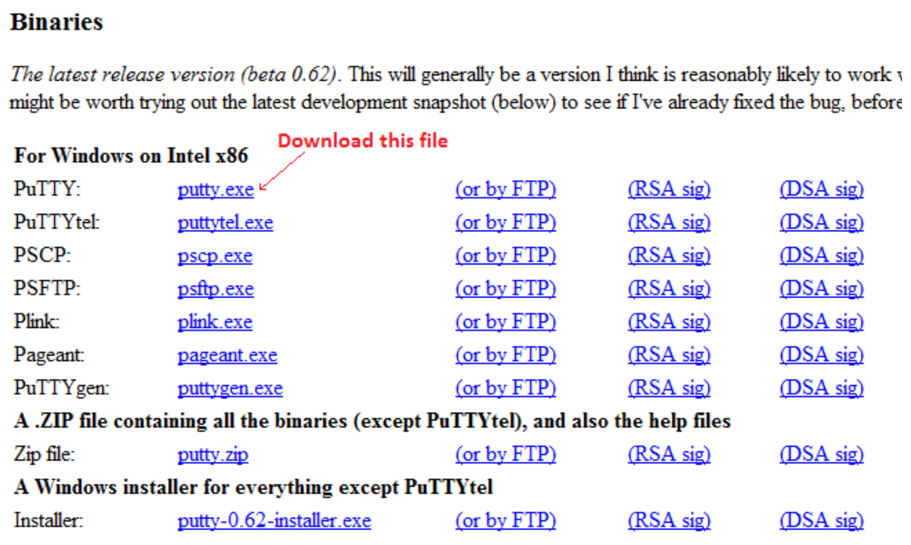
In order to connect to General and log in, you must run a **SSH client** program. In the early days of the internet, users could connect to a Unix server using a network protocol known as Telnet. The user would run a program—called a Telnet client—on his or her computer, and the Telnet client program would communicate with the server using the Telnet protocol. However, Telnet was designed at a time when network security was not as much of a concern as it is today. All information transmitted using Telnet is unencrypted and thus, a malicious user could intercept the data and read everything. For this reason, in 1995, a newer, more secure protocol known as the Secure Shell (SSH) protocol was created. With SSH, all bidirectionally-transmitted data, including passwords, are encrypted.

There are various Windows-based SSH client programs. One that is free, open source, and very popular is PuTTY[33]. PuTTY can be downloaded by navigating to the PuTTY website and clicking on the Download link at the top of the page. Scroll down on the next page that appears and click the link shown in the figure on the next page.

The file that is downloaded (putty.exe) is the actual PuTTY program, i.e., it is not an installation program. Just copy that file to a desired directory, and then you can create a shortcut on your desktop to the program. For example, I copied putty.exe from my download directory to a directory named C:\Programs\Putty. Then I created a shortcut on my desktop to the file C:\Programs\Putty\putty.exe.

[32] http://www.asu.edu/selfsub

[33] http://www.chiark.greenend.org.uk/~sgtatham/putty



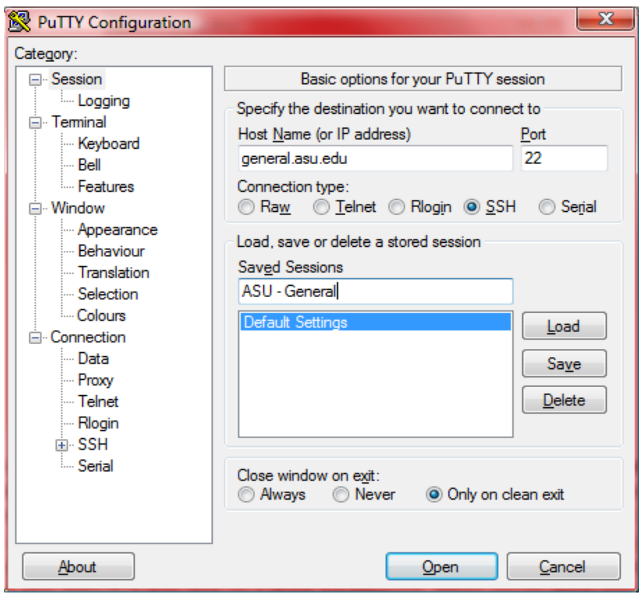
Binaries

The latest release version (beta 0.62). This will generally be a version I think is reasonably likely to work \
might be worth trying out the latest development snapshot (below) to see if I've already fixed the bug, before

For Windows on Intel x86				
PuTTY:	putty.exe	(or by FTP)	(RSA sig)	(DSA sig)
PuTTYtel:	puttytel.exe	(or by FTP)	(RSA sig)	(DSA sig)
PSCP:	pscp.exe	(or by FTP)	(RSA sig)	(DSA sig)
PSFTP:	psftp.exe	(or by FTP)	(RSA sig)	(DSA sig)
Plink:	plink.exe	(or by FTP)	(RSA sig)	(DSA sig)
Pageant:	pageant.exe	(or by FTP)	(RSA sig)	(DSA sig)
PuTTYgen:	puttygen.exe	(or by FTP)	(RSA sig)	(DSA sig)
A .ZIP file containing all the binaries (except PuTTYtel), and also the help files				
Zip file:	putty.zip	(or by FTP)	(RSA sig)	(DSA sig)
A Windows installer for everything except PuTTYtel				
Installer:	putty-0.62-installer.exe	(or by FTP)	(RSA sig)	(DSA sig)

2.5.1.1 Configuring PuTTY

After copying PuTTY to the destination directory and creating the desktop shortcut, double-click on the desktop icon to run the program. The PuTTY Configuration Dialog will be displayed.



Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): general.asu.edu Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Load, save or delete a stored session

Saved Sessions: ASU - General

Default Settings

Close window on exit: Always Never [✓ Only on clean exit]

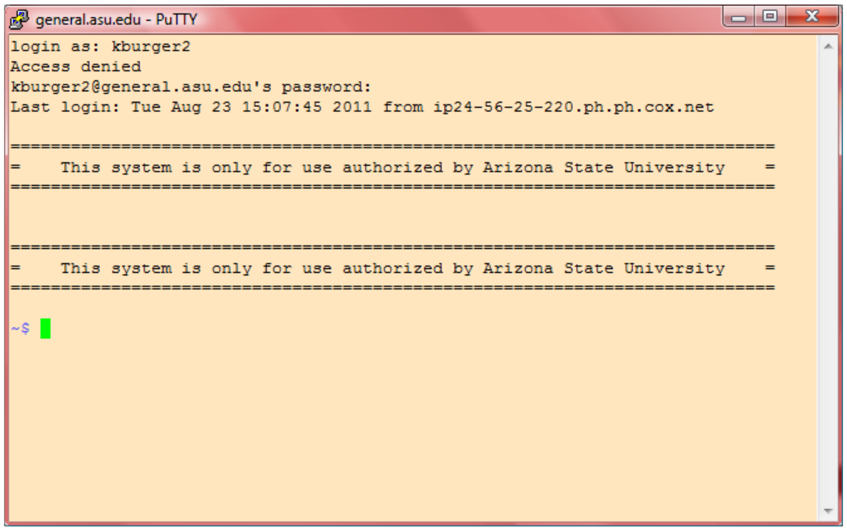
To configure PuTTY to connect to General, first enter general.asu.edu in the text field below Host Name (if you wish to use the Research cluster, enter either research2.asu.edu or medusa2.asu.edu here; note: if general.asu.edu does not work then try general1.asu.edu). Make sure Port is set to 22. Click the radio button for SSH under Connection Type.

Most of the configuration settings available through the left-hand pane can be left in the default state, but you will want to change the remote character set by expanding the Window category and clicking on Translation; then change Remote Character Set to UTF-8. Other settings that you may wish to change involve the window and color settings. Expand the Window category and click on Appearance. The default font is Courier New, Regular, 10 point; you may wish to change that by clicking the Change button. In the left-hand pane, if you click on Colours, you can set the background and foreground colors of the window and the window text. The default is gray text on a black background.

When you're done messing with colors, click Session in the left-hand pane to go back to the main configuration page. It is important to save your session, so you don't have to reconfigure PuTTY every time you wish to use it. In the text field under Saved Sessions, enter a name for the session, e.g., I used ASU–General, and then click the Save button; this will save your configuration settings.

2.5.1.2 Logging into and out of General

After configuring PuTTY, we are ready to log in to General. Running PuTTY, in the list of saved sessions, find ASU–General and double-click on it. The first time you do this, you will see a message saying something about "the server's host key is not cached in the registry, blah, blah, blah." This only happens once, and it has something to do with establishing the secure connection between General and your PC; just click the Yes button. Then, the terminal window will open and you will be presented with the login prompt. Your login user id and password is your ASURITE user id and password, e.g., mine is kburger2 and cookies. Enter your credentials, and if you are successfully logged in, you will see something like this,



```
login as: kburger2
Access denied
kburger2@general.asu.edu's password:
Last login: Tue Aug 23 15:07:45 2011 from ip24-56-25-220.ph.ph.cox.net

============================================================================
=    This system is only for use authorized by Arizona State University    =
============================================================================


============================================================================
=    This system is only for use authorized by Arizona State University    =
============================================================================

~$ 
```

Note: When I entered my user id at the *login as* prompt and pressed the enter key the system displayed Access denied, but it still allowed me to enter my password and log in.

Strange. By the way, did you know that this system is only for use authorized by Arizona State University? Just thought I would mention that, for the third time, in case you did not know that this system is only for use authorized by Arizona State University[34].

The $ symbol is called the shell prompt (or simply prompt) and it is an indication that the system (actually, the shell program bash) is sitting, waiting for you to type some command to perform some operation. We will be talking about commands as we progress, but for now, if you wish to logout you can either type **logout** and press the enter key, or hit the **Ctrl+D** key combination[35].

2.5.2 Bash Shell

Unix was created in the days before graphical terminals were common (or cheap), and historically, users would interact with the operating system by typing commands at the comment prompt. For example, type **ls** (i.e., ell ess not one ess) and press the Enter key. You should see some text displayed. The ls command is used to list the files and directories in the current directory. What you see displayed is a list of the files and directories in your **home directory** (your home directory is the directory where you log in to). As another example of a command, type **gcc --version** and press the Enter key.

```
~$ gcc --version
gcc (GCC) 4.7.0
Copyright (C) 2012 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

The gcc command invokes the GNU C compiler. The --version option tells gcc to display the version of the compiler software. As you can see, General has version 4.7.0 installed.

The major objective of the Linux part of CSE220 is to teach you how to perform various common bash shell commands, so we will be exploring those commands more in the lectures, and primarily in Chapter 5.

2.5.3 Accessing General from your Mac or Another *nix Machine

If you have a Mac, or are using another Linux machine, you can log in to General by opening a Terminal window and typing **ssh your-asurite-user-name@general.asu.edu**. The system will ask you to enter your ASURITE password and will log you in. To log out, hit Ctrl+D and you will be returned to your Mac Terminal window or other Linux machine.

[34] It probably wouldn't hurt for me to mention also, that this system is only for use authorized by Arizona State University, just in case you have some nefarious internet-hacking scheme in mind.

[35] Why Ctrl+D? In Unix, text files are terminated with an end-of-file (eof) character which is the character with ASCII value 4. When Ctrl+D is pressed in the terminal, the shell sees this as the end-of-file character because Ctrl+D is the character with ASCII value 4 (Ctrl+A is ASCII 1, Ctrl+B is ASCII 2, etc). Hence, by sending enf-of-file to the shell, you're telling it that there is not more input. Hence, it terminates.

2.6 Running Linux in a Virtual Machine with VMware Player

VMware is a company that develops hypervisor[36] software. A hypervisor is software that emulates a computer system as a "virtual machine" or VM. Any operating system can be installed on the VM, and at that level, the OS does not know that it is not running on bare hardware. The big advantage is that you can use the VM (or VM's because you can have multiple ones running different OS's independently or simultaneously).

VMware has a free, reduced-functionality virtual machine named VMware player (which only runs on Windows and Linux), which can be downloaded from [37]. To get started, navigate to the VMware player website and click the Download link. The most recent version is 5.01; click on the link labeled **VMware-player-5.0.1-894247.exe** which is a Windows installer[38] and install the program.

I've been itching to try the Cinnamon desktop created by the Linux Mint community, so let's install it. Navigate to the Linux Mint download webpage at [39] and click on either the 32-bit or 64-bit Cinnamon link that is not labeled "No codecs". If you want to use MATE, KDE, or Xfce instead, click on the appropriate link. While you wait for the download to complete, you probably have enough time to go bake your instructor some cookies[40]. Once the download completes, follow these instructions to install Linux Mint.

1. Launch VMware player and click the link labeled Create a New Virtual Machine.
2. In the New Virtual Machine Wizard dialog, select the Installer Disc Image File (.iso) button. Click the Browse button and navigate to the folder where you saved the Linux Mint image and open it. Then click the Next button.
3. In the next dialog, select the Linux radio button. In the drop down list box, select either Other Linux 2.6.x Kernel (for 32-bit Linux Mint) or Other Linux 2.6.x Kernel 64-bit. Then click Next.
4. You can name the VM anything, I'm going to name mine "Linux Mint 64-bit Cinnamon." Select a folder where the VM files will be stored, e.g., C:\Linux\linux-mint-64-cinnamon. You could also create the VM files on a USB flash drive so you can take your Linux distro with you. In that case, though, you would also want to install VMware Player on the flash drive as well. However, be aware that during the installation, you will be creating a virtual hard drive on which Linux will be installed. You need to make sure the size of this hard drive will not exceed the capacity of your flash drive. In any case, click Next.

[36] https://en.wikipedia.org/wiki/Hypervisor

[37] https://www.vmware.com/products/player/

[38] By the way, you VMware player also comes in a Linux version, which you can run on a Linux machine. This would permit you to try out different Linux distros by installing them in a VM rather than directly on your hard drive. It will also permit you to run Windows XP, Vista, 7, and 8 on your Linux box. A fun trick to try would be to install VMware player for Linux on a Linux machine, and run Windows 8 in a VM. Then, in your Windows 8 VM, install VMware player, and set it up to run Linux. I seriously doubt this would work, but let me know if it does.

[39] http://www.linuxmint.com/download.php

[40] I like chocolate chip and oatmeal raisin the best. Make em chewy too, I hate dry cookies.

5. In the next dialog, WMware player recommends a hard drive size of 8 GB for the Linux installation and your personal files. I probably wouldn't make this any smaller, but if you plan on using this Linux VM a lot, you may want to make it larger to store your personal files. I usually leave the Split Virtual Image into Multiple Files button selected, just in case I decide to move this VM somewhere else later. Click Next.
6. In the next dialog, click Customize Hardware. In the Customize Hardware dialog, you can select how much system memory to allocate to the VM. My laptop has 8 GB, so I'm going to allocate 2 GB; this can be changed later.
7. If you have a multicore machine, you can allocate more than one processor to the VM; I'm going to allocate 2 (this, and all of the other hardware options, can be changed later). If you have an Intel Core chip that supports virtualization via VT-x/EPT, then select that radio button (or select it if your AMD chip supports virtualization). **Note:** In order to take advantage of hardware virtualization, you will have to enable it in your BIOS.
8. Under Network Adapter your options are Bridged, NAT, Host-only, and Lan segment. NAT should work just fine, however, I've had some problems with my VM losing network connectivity when my laptop goes to sleep, so if that happens, selecting Bridged will fix the problem.
9. I always leave the rest of the hardware options as they are. When you're finished, click Close, and then Finish in the next dialog.
10. Your VM is now ready to install Linux. Click on the Play Virtual Machine link to boot it from the virtual CD-ROM which contains the Linux Mint .iso image.
11. Linux Mint will begin to boot from the CD-ROM. Mint (and Ubuntu) are hands down the easiest Linux distros to install, so this should be a no brainer for you. When the desktop appears, double-click on the Install Linux Mint icon.
12. I'm not going to walk you through the entire procedure, but I will point out the highlights. Oh, by the way, notice the verbiage at the bottom of the VMware player window which is blabbering about installing VMware tools. That's important, but leave it on hold for now. We'll finish that up after installing Mint.
13. In the Installation Type dialog, you can either erase the entire disk and install Mint using the default partitioning scheme, or you can partition the disk as you see fit. Don't worry about screwing up your hard drive, because the disk you're formatting and partitioning is a virtual hard drive (which is simply stored as a regular old file on your computer's hard drive). The easiest thing to do is to select the first radio button and click Continue.
14. Linux Mint will begin installing files on your virtual hard drive which is /dev/sda using the ext4 file system. If you don't know what that means, it doesn't matter, and if you do know, well, never mind.
15. When the install finishes, restart the system and log in.
16. Once you are logged in, click the Install Tools button to install the VMware Tools for Linux. After a bit, you will see a file icon for a file named VMwareTools-9.2.2-893683.tar.gz. We need to extract the contents of this file, so double-click on it and the file will be opened by Archive Manager.
17. You will see a file icon for a file named vmware-tools-distrib. Right-click on it and select Extract from the popup menu. In the next dialog, click on the Home icon under Places and

then click the Extract button. The contents of the file will be extracted to your home directory.

18. Close the Archive Manager windows. We're not done yet. In the lower-left corner of the Linux Mint desktop, you will see three icons on the task bar. The left one is for Firefox and the one to the right of it is for Terminal. Click on Terminal to open a terminal window.
19. When the Terminal window opens, type **cd vmware-tools-distrib** to make that directory your current directory.
20. To finish the installation, you must be logged in as the super user (also known as "root"). Type **su** and enter the root password. Now, type **./vmware-install.pl** to install the VMware Tools. Press the Enter key to accept the defaults at each prompt.
21. Once the installation finishes, type **/usr/bin/vmware-user**
22. You may see some error messages. I don't know why this happens. Hit **Ctrl+D** to logout of the root shell. Then hit Ctrl+D again to logout of Terminal.
23. Click on Menu in the lower-left corner of the taskbar. Click on the bottom icon to shut down the system. In the dialog that appears click the Restart button.
24. Log in again. You now have a fully-functional Linux Mint system. Good luck figuring it out.

2.6.1 Package Management with Synaptic

Programs in the Linux world are released as **packages** which are containers for the program files. Packages often depend on other packages being installed on the system, e.g., a program may use a special library which is released as another package, so the package file also specifies package dependencies. In Debian/Ubuntu/Mint, **Synaptic** is a GUI-based package manager which can be used to install, update, and remove packages from your system. To run Synaptic, click on Menu > Preferences > Synaptic Package Manager.

Mint comes with a large number of packages already installed, including Firefox, LibreOffice, Thunderbird mail client, GIMP image editor, and VLC media player. A lot of the fun in using Linux is exploring the large number of free and open-source software packages, and Synaptic makes it as easy as pie to do that.

Note that Mint comes with the GCC C compiler installed, but it does not install the GCC C++ compiler which you will need for CSE220. To install it, launch Synaptic and navigate to the Development category in the left-hand pane. Scroll down in the right-hand pane and mark the **g++** package for installation. You also will need to install the **Mercurial** package by navigating to the Development (universe) category and marking **mercurial** for installation.

If you wish to use a different text editor than GEdit, which is the default text editor installed by Mint, navigate to the Editors (universe) category. I cannot recommend any of these, other than Kate, because I am not familiar with them. Kate is the default KDE text editor, and installing it will also cause a lot of KDE-related packages to be installed. You may or may not want to do this, but I do like the Kate editor. For more discussion of text editors, refer to Chapter 11.

2.7 Use the Computers in the BYENG 214 Computer Lab

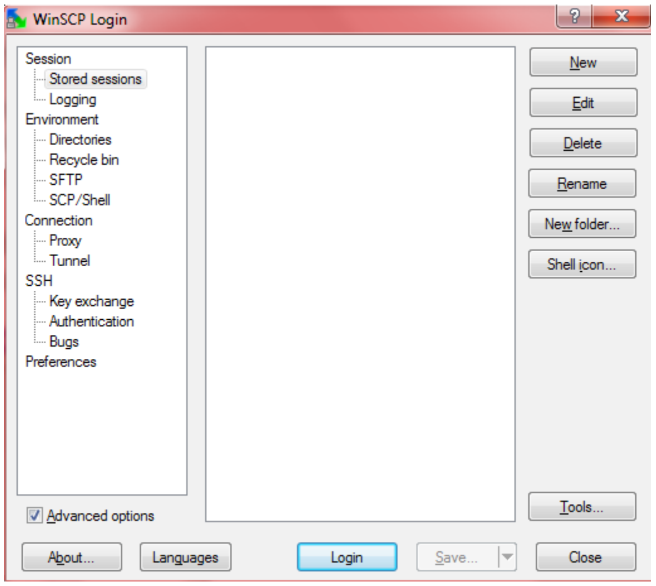
The computers in the CIDSE BYENG 214 computer lab are configured to dual boot Windows and the CentOS Linux operating system. CentOS is an open-source "community" version of Red Hat Enterprise Linux (i.e., Red Hat does not own it, work on it, or support it). You should be able to figure it out. By the way, your log in user identifier and password are your ASURITE credentials.

2.8 Buy a Mac

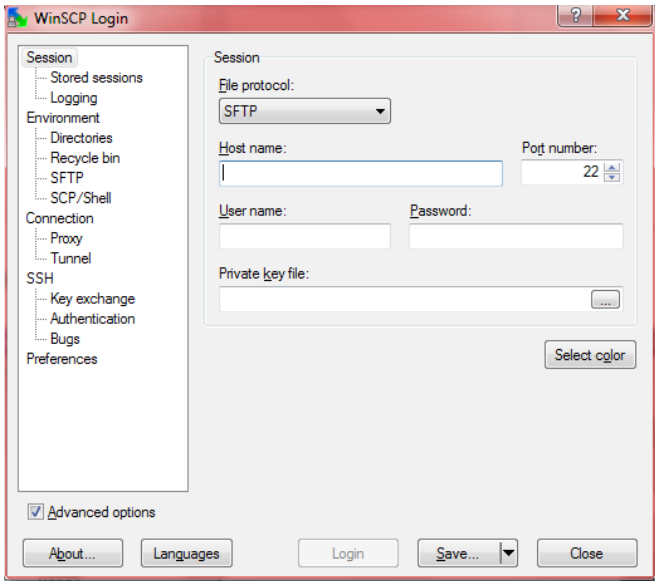
Mac's run a variant Unix kernel named Mach developed originally by Carnegie Mellon University which Apple acquired via their purchase of NeXT in 1997. The Apple-modified Mach kernel is packaged together with code derived from NeXTSTEP (the NeXT OS), BSD, and other free software projects to create the open-source Darwin OS. Darwin forms the basis of Mac OS X, the latest release being Version 10.7.2. Since Version 10.3 of Mac OS X, the default Unix shell has been Bash. Mac's also ship with the Tcsh shell which is also a very common Unix shell (which is based on an earlier shell named csh, pronounced see-shell). I suggest for CSE220 you stick with, or switch to, Bash. Your Mac product may already have the GNU C/C++ compilers installed. If not, they are available on the internet. There, I have just exhausted my entire knowledge base on the Apple Mac.

3 Transferring Files with WinSCP

If you are working at home and you want to transfer files from or to your account on one of the Linux systems, I recommend a program named WinSCP[41]. It is a free, open-source **Secure FTP** (SFTP) program with a very easy to use GUI. Download it and install it. When you launch it, you can set up a saved session for general.asu.edu. You will see dialog box that looks like the following,



Click the New button and the following dialog box will appear,



Session

File protocol: SFTP

Host name:

Port number: 22

User name:

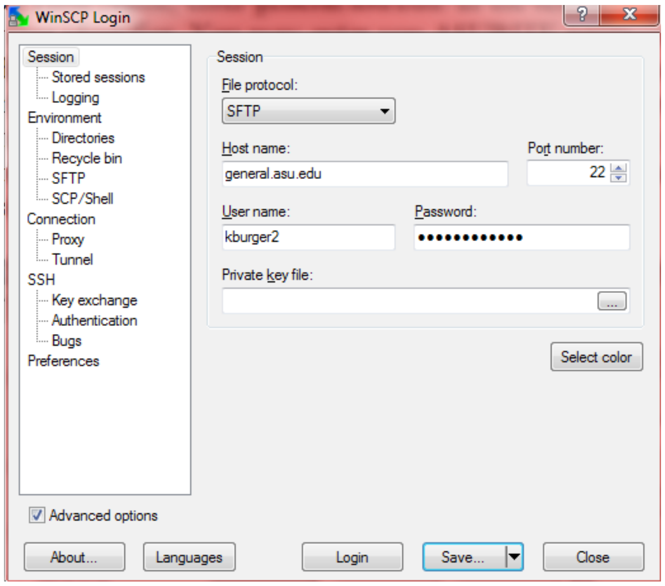
Password:

Private key file:

Advanced options [✓]

[41] http://winscp.net/eng/index.php

In the textfield for Host Name, enter general.asu.edu (or general1.asu.edu if you have to). In the textfield for User Name enter your ASURITE user identifier. You may enter your ASURITE password in the Password textfield if you do not want to be prompted to log in every time you connect. Make sure the Port Number is set to 22, and that SFTP is selected for the File protocol.



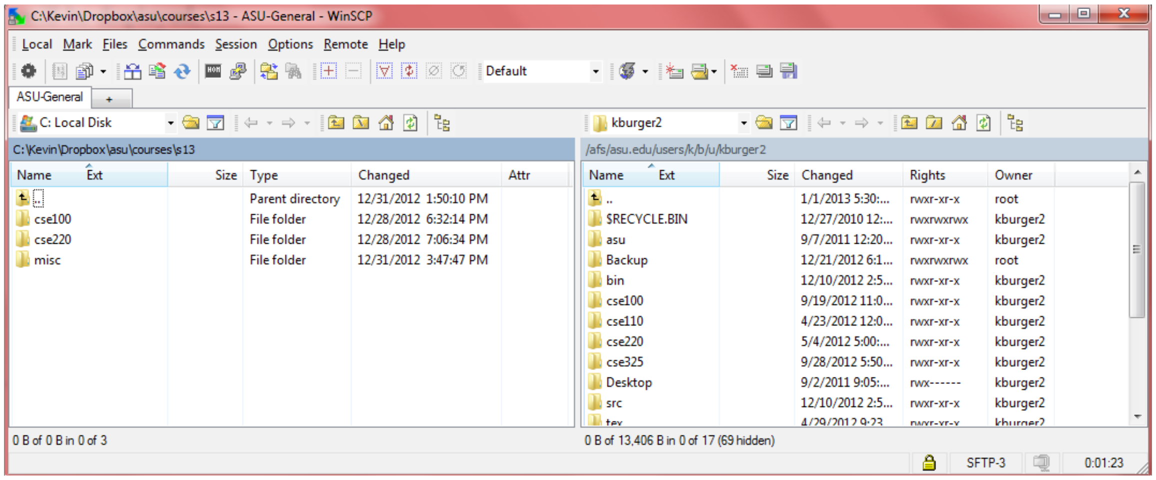
Session
File protocol: SFTP
Host name: general.asu.edu
Port number: 22
User name: kburger2
Password: ●●●●●●●●●●●●
Private key file:

To save this configuration so you won't have to reenter it every time you wish to transfer files, click the Save button. You are then asked to give the session a name, you might use ASU–General for example. If you don't want to have to enter your password every time you transfer files, select the Save Password check box.

After clicking OK, you will be returned to the original WinSCP Login dialog window and you will see ASU-General as a saved session. To connect, select ASU–General from the list of saved sessions, and click the Login button. You may get a dialog box that says something about the server's host key and the cache. This is part of the security and encryption setup, so just click Yes to accept it. If everything succeeds the window shown on the next page will be displayed.

In the left pane of the window are the files on your local machine; the right pane displays the files in your home directory on the remote machine. You can copy and move files between these two machines by drag-and-drop. You can also delete files, rename files, create directories, etc, but you should be able to figure that out.

A very handy feature of WinSCP is to keep the local and remote directories in sync. I use this quite a bit when I write code on General. I don't like to use vi, Nano, Joe, or any of the other crappy text editors on General, but I do like to compile and debug my code there. So, what I do is I navigate to the directories on my local computer and on General where my source code files are stored. I tell WinSCP to keep the remote directory up to date by selecting Commands | Keep Remote Directory up to Date... from the main menu (Ctrl+U).


C:\Kevin\Dropbox\asu\courses\s13 - ASU-General - WinSCP
Local Mark Files Commands Session Options Remote Help
Default
ASU-General
+
C: Local Disk
kburger2
SFTP-3
0:01:23

C:\Kevin\Dropbox\asu\courses\s13

Name	Ext	Size	Type	Changed	Attr
..			Parent directory	12/31/2012 1:50:10 PM	
cse100			File folder	12/28/2012 6:32:14 PM	
cse220			File folder	12/28/2012 7:06:34 PM	
misc			File folder	12/31/2012 3:47:47 PM	

0 B of 0 B in 0 of 3

/afs/asu.edu/users/k/b/u/kburger2

Name	Ext	Size	Changed	Rights	Owner
..			1/1/2013 5:30:...	rwxr-xr-x	root
$RECYCLE.BIN			12/27/2010 12:...	rwxrwxrwx	kburger2
asu			9/7/2011 12:20...	rwxr-xr-x	kburger2
Backup			12/21/2012 6:1...	rwxrwxrwx	root
bin			12/10/2012 2:5...	rwxr-xr-x	kburger2
cse100			9/19/2012 11:0...	rwxr-xr-x	kburger2
cse110			4/23/2012 12:0...	rwxr-xr-x	kburger2
cse220			5/4/2012 5:00:...	rwxr-xr-x	kburger2
cse325			9/28/2012 5:50...	rwxr-xr-x	kburger2
Desktop			9/2/2011 9:05:...	rwx------	kburger2
src			12/10/2012 2:5...	rwxr-xr-x	kburger2
tex			4/29/2012 9:23	rwxr-xr-x	kburger2

0 B of 13,406 B in 0 of 17 (69 hidden)

In the dialog window that appears, there are some settings you may wish to fool around with. The Delete Files check box makes WinSCP delete files and directories in the remote directory that you delete in the local directory. The Existing Files Only check box tells WinSCP that if a file or directory that exists in the local directory is not in the remote directory, then it should be ignored. The Update Subdirectories option tells WinSCP to keep remote subdirectories up to date as well. Check Use Same Options Next Time to force WinSCP to remember your settings and apply them the next time you invoke this command. Check Synchronize On Start to tell WinSCP to tell WinSCP to immediately synchronize the remote directory with the local directory.

To summarize: select Delete Files, Synchronize On Start, Update Subdirectories, and Use Same Options Next Time; deselect Existing Files Only. Next, I open a PuTTY connection to General, and when I need to compile and debug my code, I switch to the PuTTY window and do my business. When I want to make changes to my code, I switch to the editor window and do my business. WinSCP keeps everything up-to-date on General in the background.

By the way, for the lazy, WinSCP can be programmed to automatically upload and download files using scripts, see [42].

When you are finished transferring files, click Session | Disconnect on the menu bar. To exit WinSCP click Close on the session list window.

[42] http://winscp.net/eng/docs/scripting

4 Shells

In *nix the **shell**[43] is a text-based interface to the operating system. The **kernel**[44] is the lowest-level innermost part of the operating system, so the shell is so named because it "surrounds" the kernel.

Over the years, various shell programs have been written. The first was the Thompson shell (commonly referred to as simply sh[45]) written by Ken Thompson at Bell Labs. Other popular shells have that have been written over the years include the Berkeley Unix C shell (csh), the TC shell (tcsh; the T comes from the TENEX operating system), the Korn shell (ksh), the Z shell (zsh), the Bourne shell (sh), and the Bourne-again shell (Bash[46]).

Bash was the shell originally developed by GNU for the GNU operating system. It is commonly available in most GNU/Linux distributions and this manual is written assuming you are using Bash. Extensive online documentation for Bash is available[47]. Another reference is available at [48].

You can see which shells are installed on your system by typing **cat /etc/shells** (the **chsh -l** command simply displays this file). Most likely if you are using Linux, your default shell is Bash. An easy way to tell what shell you are running is to type **less /etc/passwd** and look for the line containing your username. The last entry on the line will be your default login shell.

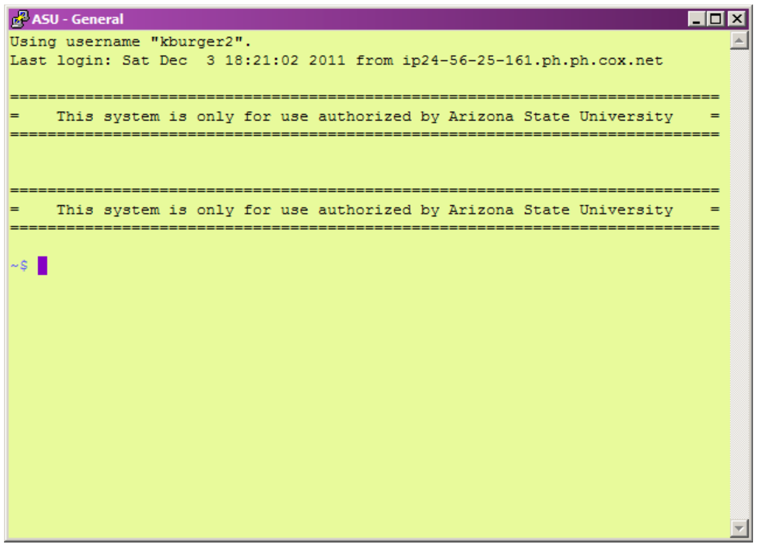
If you wish to change your default login shell you can use the **chsh -s *shell*** command, e.g., **chsh -s zsh**. This will modify the shell entry for your username in /etc/passwd so the next time you login you will be running the Z shell.

4.1 The Bash Shell

See Chapter 2 for instructions on how to log in to General. Once logged in, you will see a window that looks similar to the one on the top of the next page. This is called a **terminal window** or just **terminal**; you may also hear the term **console window** or just **console**.

The $ prompt is the Bash **shell prompt**. It is Bash's way of letting you know that it is waiting for you to type a command. The ~ symbol in Bash always refers to a user's **home directory**. Your home directory is where you are placed when you first log in and it is the area in the file system where you can store your files. The prompt string can be changed, see Section 7.1.

43 http://en.wikipedia.org/wiki/Shell_(computing)
44 http://en.wikipedia.org/wiki/Kernel_(computer_science)
45 http://en.wikipedia.org/wiki/Thompson_shell
46 http://www.gnu.org/software/bash
47 http://www.gnu.org/software/bash/manual/bashref.html
48 http://tiswww.case.edu/php/ chet/bash/bashref.html



```
Using username "kburger2".
Last login: Sat Dec  3 18:21:02 2011 from ip24-56-25-161.ph.ph.cox.net

==========================================================================
=    This system is only for use authorized by Arizona State University    =
==========================================================================


==========================================================================
=    This system is only for use authorized by Arizona State University    =
==========================================================================

~$
```

When you type commands at the prompt and press the Enter key, Bash will read the command string you typed in, parse it to break it into words and operators, perform some post-processing, and will then execute the command. Unless you tell it otherwise, Bash will wait for the command to finish executing before displaying the $ prompt again (see Chapter 10 on Processes for how to run a command in the background).

Some of the commands that you may use are built-in to Bash, i.e., the code that gets executed is inside the Bash program itself. However, most *nix commands are standalone programs that are found in the directories /bin, /usr/bin, and /sbin directories (see Section 5.1 for a discussion of these directories). For example, we can determine where the GNU C compiler is installed on General by typing,

```
$ whereis gcc
gcc: /usr/bin/gcc /usr/lib/gcc /usr/libexec/gcc /usr/share/man/man1/gcc.1.gz
```

whereis is a command which will search for the program file given to it. In this case we are searching for the program file named **gcc**. whereis tells us that a file named "gcc" can be found in four separate directories. In fact, the first one /usr/bin/gcc is the actual executable file which is the GNU C compiler. For fun, try to determine where the whereis program is installed,

```
$ whereis whereis
whereis: /usr/bin/whereis /usr/share/man/man1/whereis.1.gz
```

We can see that it is also installed in /usr/bin. In fact, most of the basic *nix commands (i.e. executable programs) are located in this directory.

Bash will also execute special programs called scripts written in the Bash shell scripting language. See Chapter 12 for a basic introduction to Bash shell scripting.

Overall, Bash is a very complex program, and we will only learn a small part of it in this course. However, what we learn in this course is supposed to provide you a solid foundation on which you can learn and obtain more knowledge of *nix and Bash.

5 The *nix File System and Common Commands

The *nix **file system**[49] is the part of the OS that manages the files and directories stored on secondary storage devices such as tape, optical and hard disks. Files and directories in the file system are stored in a **hierarchical directory tree**[50].

At the top level of the directory tree is the **root directory** indicated by a forward slash symbol /. Root is at the top of the file system hierarchy. Every file and directory on the system is stored under root and there is nothing above it in the directory tree.

5.1 Common Directories in *nix

On a typical *nix system you will find several common directories directly under root. Some of these common directories are discussed below. A very good discussion of the *nix file system is available at [51].

/bin
Binary images (executables, which in *nix terminology are called **binaries**) for various *nix commands used by users. These would include programs for commands such as ls, cp, mkdir and so on. In a typical distribution there will be a large number of commands files located in this directory, but only a small subset of them are technically required to be installed there.

/boot
Contains the files necessary to boot the *nix operating system. This includes the binary image of the kernel (i.e., the compiled kernel source code). In GNU/Linux The kernel file is commonly named vmlinux (uncompressed) or vmlinuz (compressed) followed by the version and release numbers. If you want to know which version of the kernel you are using, type **uname -r** at the shell prompt (discussed later).

/dev
The device directory contains files corresponding to different I/O devices connected to the system. These would include tape drives, cdrom drives, hard disks, terminals, printers, etc. A special device is /dev/null. Anything written to /dev/ null disappears into the ether. One of the features that made Unix popular in the early days was that all devices are treated as files, and could be accessed using file I/O commands.

/etc
This directory contains host-specific files and directories which are used primarily for system configuration purposes. Executable programs are not located here. If the X win-

[49] http://en.wikipedia.org/wiki/Berkeley_Fast_File_System
[50] http://amath.colorado.edu/computing/unix/tree.html
[51] http://www.pathname.com/fhs/pub/fhs-2.3.html

dow system is installed, the files for it will be installed in a subdirectory of /etc, e.g., /etc/X11.

/home
Users' personal directories are located here. Normally each user is given a home directory where he/she may store his/her files. This is the directory where, as a user, you would spend most of your time. A typical home directory for user fredf would be /home/fredf. Note that the special symbol ~ is used to refer to the home directory in the shell (discussed later).

/lib
Contains shared library images needed to boot the system and to run the commands in bin and sbin.

/mnt
Used to temporarily mount another file system under the current file system, e.g., to connect the file system for a floppy disk or a cdrom to the current file system. How these devices are mounted is not something ordinary users care about.

/opt
Third-party application programs can be installed here, each within its own subdirectory.

/root
This is root's home directory. The special login identifier **root** is the master login for the entire system. Root has privileges that other users do not have, and he or she is typically the system administrator who is responsible for administering the system, installing programs, adding users, mounting file systems, etc.

/sbin
This directory contains system binary files, i.e., executables that are normally used only by root in administering the system. Other system binaries may be found in /usr/sbin and /usr/local/sbin.

/tmp
Temporary files are often placed here.

/usr
The files and directories under usr are mounted read-only (except to root). Underneath usr are several important subdirectories.

> /usr/bin
> Other user-specific binaries. For example, the gcc program might be installed here.

/usr/include
Include header files for the C compiler are placed here.

/usr/lib
Contains object files and library files used in software development.

/usr/local
Another place where root can install user programs. For example, if you install the Firefox web browser, you may install it under /usr/local/firefox for example. Multiple networked host systems may access the files in /usr/local.

/usr/share
Shared data files, e.g., font files, man pages, other documentation, icon images files, etc.

/usr/src
Source code, generally for reference only (i.e., not for development purposes). On many GNU/Linux systems, the Linux kernel source code will be installed in /usr/src/linux (during installation, you usually have the choice to download and install the Linux source code).

/usr/X11
Files used by X Windows (X is part of the graphical system of your Linux installation).

/var
Variable data files (i.e., data files that contain data that are changing) are stored here. These would include temporary files and system log files.

5.2 Path Names

When referring to files in the file system, the shell maintains a current directory pointer which also "points" to the **current directory**. When you log in to the system, you will be placed in your home directory, e.g., /home/fredf if your log in identifier were fredf. That is now your current directory and is called an **absolute path name** because it refers to the complete path that must be followed from root to get to the directory fredf. Another example of an absolute path name might be /usr/local/bin/firefox/v1.4.0/bin/firefox.

A **relative path name** is a path to a file or directory which is specified relative to your current directory. Two special relative path names are . (pronounced dot) and .. (pronounced dot dot). Dot always refers to your current directory and dot dot refers to the directory above your current directory (i.e., the parent directory). For example, if my current directory is /usr/local/bin then . is /usr/local/bin and .. is /usr/local. These are quite useful as we will see.

5.3 Commands, Arguments, and Options

A **command** (some commands are built-in to the shell and others are separate binaries found in /bin, /sbin, /usr/bin, and other directories) is a word that is typed at the terminal window prompt in order to perform some action. For example, the **ls** command is used to list the contents of a directory (specifically, the current directory). If you would like to view the contents of a different directory, then you can supply an **argument** to the ls command which is the location of the desired directory, e.g., **ls /bin** would display the contents of the /bin directory where /bin is the argument to the ls command.

In addition to arguments, many (most) *nix commands accept command **options** as well. Options are specified either with a hyphen followed by a letter, e.g., -l, or a pair of hyphens followed by a word, e.g., --help (options starting with -- are GNU extensions to the older Unix command options).

For example, **ls -l /usr/bin** is a command (ls) with the option -l and the argument /usr/bin. We'll talk about what -l does when used with the ls command in a bit.

Options modify the way commands work. When no options are specified we would say that is the default behavior of the command. Options can be used to alter the default behavior. Note that like everything else in *nix, options are case-sensitive. There is a difference between ls -l and ls -L as there is between ls and LS (the former is a command; the latter is not).

A command can have multiple options, e.g., the ls command supports two options -l and -a that alter it in different ways. A legal ls command using these two options would be **ls -a -l**. But often times, options can be combined so this same command could be typed as **ls -al** and it would perform the same action.

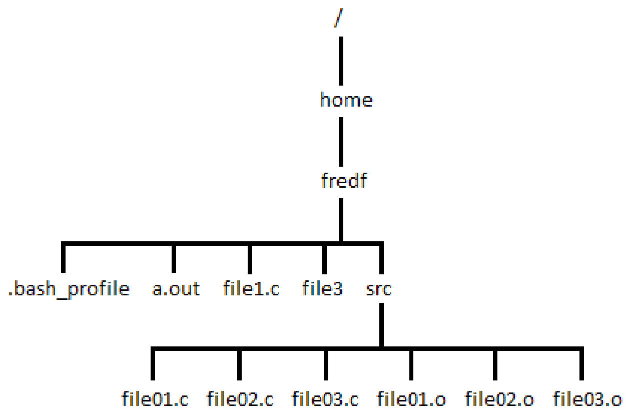
A useful option you should know is --help. Many commands will display a screen of text describing the commands, its action, and its options and arguments in response to the --help option.

Whitespace surrounding options is important. The command ls -a-l is erroneous because there is no space in between the -a and -l options. The correct command is ls<space>-a<space>-l.

In conclusion, know that the general format of a *nix command is: **command options arguments**—in that order.

5.4 Common File and Directory Commands

In this section, assume we are logged in as user fredf in our home directory /home/fredf and we have the files as displayed below,



5.5 How Do I Know Which Directory I Am In?

Use the command pwd (print working directory),

```
$ pwd
/home/fredf
```

5.6 List Files, File Permissions, The Chmod Command, ACL's

To list the files in a directory type ls,

```
$ ls
a.out file1.c file3 src
```

The option -a will show all the files in the directory,

```
$ ls -a
.bash_profile a.out file1.c file3 src
```

The special file **.bash_profile** is a configuration file which is used to configure the Bash shell when a user logs in; we will discuss it later. Note that files starting with a dot are **hidden files** and are not displayed during a normal ls command. Hidden files are usually used to store configuration information for various programs. The option -l will display a long listing,

```
$ ls -l
-rwxr--r--  1 fredf fredf   20987 Oct 10 10:04 a.out
-rw-r--r--  1 fredf fredf    1098 Mar  7  2005 file1.c
-rw-r--r--  1 fredf fredf     654 Jun 12 18:44 file3
drwxr-xr-x  1 fredf fredf       0 Jan 01  2005 src
```

The first column displays the permissions for the files[52]. **Permission attributes** are changed with the chmod command.

```
$ chmod go= a.out
$ ls -l
-rwx------  1 fredf fredf   20987 Oct 10 10:04 a.out
-rw-r--r--  1 fredf fredf    1098 Mar  7  2005 file1.c
-rw-r--r--  1 fredf fredf     654 Jun 12 18:44 file3
drwxr-xr-x  1 fredf fredf       0 Jan 01  2005 src
```

The above command removes all of the group and other's permission bits.

```
$ chmod ugo=rwx a.out
$ ls -l
-rwxrwxrwx  1 fredf fredf   20987 Oct 10 10:04 a.out
-rw-r--r--  1 fredf fredf    1098 Mar  7  2005 file1.c
-rw-r--r--  1 fredf fredf     654 Jun 12 18:44 file3
drwxr-xr-x  1 fredf fredf       0 Jan 01  2005 src
```

The above command gives everyone in the group and all other's full permissions. See [53] for more explanation.

Many Linux file systems (ext2, ext3, ext4, ReiserFS, JFS, XFS, and perhaps some others) support **access control lists** (ACL) extend the functionality of permission attributes by providing finer access granularity. For example, with traditional permission attributes, there is no straightforward way to specify access to files and directories for an indivual user[54]. With ACL's this—and other types of access—can be easily accomplished. However, I won't discuss it here, but if you are interested, see [55].

5.7 Deleting Files

To delete a file use the **rm** (remove) command,

```
$ rm a.out
$ ls
file1.c file3 src
$ rm file3
$ ls -al
-rw-r--r--  1 fredf fredf      36 Sep 23  7:53 .bash_profile
-rw-r--r--  1 fredf fredf    1098 Mar  7  2005 file1.c
drwxr-xr-x  1 fredf fredf       0 Jan 01  2005 src
```

[52] http://zzee.com/solutions/unix-permissions.shtml

[53] http://catcode.com/teachmod

[54] This *can* be accomplished by creating a system wide group containing only one member: the individual user. Then you could grant access to the group, which would effectively grant access to the individual user. However, this is cumbersome, and would quickly lead to an unmanagable list of groups.

[55] http://users.suse.com/~agruen/acl/chapter/fs_acl-en.pdf

5.8 Creating, Changing, and Removing Directories

To create a new directory use the **mkdir** (make directory) command and to remove it use the **rmdir** (remove directory) command,

```
$ mkdir cse220
$ ls
a.out cse220 file1.c file3 src
$ rmdir cse220
$ ls
a.out file1.c file3 src
$ mkdir cse220
$ mkdir cse220/assgn02
$ ls
a.out cse220 file1.c file3 src
$ ls cse220
assgn02
```

To change your working (current) directory, use the **cd** (change dir) command,

```
$ cd cse220
$ pwd
/home/fredf/cse220
$ ls
assgn02
```

Suppose we have created directories cse220 and cse220/assgn02 and that cse220/assgn02 has the files file1.c, file2.c, file1.o, file2.o, and a.out in it.

```
$ cd assgn02
$ pwd
/home/fredf/cse220/assgn02
$ ls
a.out file1.c file1.o file2.c file2.o
$ cd .
$ pwd
/home/fredf/cse220/assgn02
$ cd ..
$ pwd
/home/fredf/cse220
$ cd ..
$ pwd
/home/fredf
```

Before deleting a directory, it must be empty,

```
$ rmdir cse220
rmdir: `cse220': Directory not empty
$ cd cse220
$ rmdir assgn02
rmdir: `assgn02': Directory not empty
$ cd assgn02
$ rm file1.c
$ rm file2.c
$ rm file1.o
$ rm file2.o
$ rm a.out
$ ls
$ cd ..
$ rmdir assgn02
$ cd ..
$ rmdir cse220
$ ls
a.out file1.c file3 src
```

The command **cd** with no arguments or options will always change you back to your home directory. The symbol ~ also always refers to your home directory as well.

```
$ cd /usr/local/bin
$ pwd
/usr/local/bin
$ cd
$ pwd
/home/fredf
$ cd /usr/local/bin
$ pwd
/usr/local/bin
$ cd ~
$ pwd
/home/fredf
$ cd cse220
$ cd assgn02
$ pwd
/home/fredf/cse220/assgn02
$ rm ~/file1.c
```

The last command would remove the file /home/fredf/file1.c.

5.9 File Name Pattern Matching

Sometimes you want to do something to a group of files, e.g., delete all of them, without having to perform the command on each file individually. For example, suppose we want to delete all of the *.c* files in a directory. This would be annoying,

```
$ cd ~/src
$ rm file01.c
$ rm file02.c
$ rm file03.c
```

It would be even more annoying and downright tedious if we needed to delete fifty *.c* files. What we need is a quick-and-easy way to say to Bash, "Delete any file ending with *.c*." Fortunately, there is, using Bash **file name pattern matching**.

A Bash file name pattern is a string of symbols which will match zero or more file names. The most basic pattern symbols (called **wildcards**) are ***** (star) and **?**. Star means 0 or more characters and ? means exactly one character. Commands involving file names patterns are expanded by the shell (this is called **globbing** after the name of a former program called **glob** which used to do this as a separate program outside the shell). Thus, **file*** will match any file name which starts with the characters 'f', 'i', 'l', 'e', and then is followed by zero or more occurrences of any character. For example, in fredf's home directory, the file name pattern **file*** would match *file1.c* and *file3*. The pattern **file1*** would match only *file1.c*.

A single * matches every file name in a directory, so the command **rm *** will attempt to delete every file in a directory. To delete all of the *.c* files, type **rm *.c**.

If I have a group of files named *file1.c*, *file2.c*, *file3.c*, *file1.o*, *file2.o*, *file3.o*, *file01.c*, *file02.c*, *file03.c*, *file01.o*, *file02.o*, *file03.o*, *file01.xyz*, *file02.xyz*, and *file03.xyz*, then the pattern **file??.c** would match *file01.c*, *file02.c*, and *file03.c* but not *file1.c*, *file2.c*, or *file3.c*; it would also not match the *.o* files or the *.xyz* file. The pattern **file01.?** would match *file01.c* and *file01.o* but not *file01.xyz* since it's extension is three characters. The pattern **file01.???** would match any file name with the name *file01* and a three character extension, specifically, *file01.xyz*.

A set of characters to match can be specified with brackets [ ], e.g., **[abc]** would match exactly one 'a', 'b', or 'c' character. A **range expression** can be used to specify a range of characters, e.g., **[a-z]** would match exactly one of 'a', 'b', 'c', ..., 'y', or 'z'. Multiple range expressions can be given, e.g., **[a-zA-Z0-9]** would match exactly one lowercase letter, or uppercase letter, or digit. To match a hypen, make it the first or last character in the set, e.g., **[-A-Z]** would match exactly one hyphen or an uppercase letter (**[A-Z-]** would do the same thing). If the first character following the opening bracket [is ! or ^ then any character *not* in the range is matched, e.g., **[^A-Za-z]** means match exactly one character that is not a letter. Within brackets, **character classes** can be used to specify common classes. The syntax is **[:*class*:]** where *class* is one of,

Character Class	**What it Matches**
alnum	Matches a lowercase letter, an uppercase letter, or a digit.
alpha	Matches a lowercase letter or an uppercase letter.
blank	Matches a space or tab character.
digit	Matches a digit.
lower	Matches a lowercase letter.

Character Class	**What it Matches**
space	Matches a whitespace character[56].
upper	Matches an uppercase letter.
word	Matches a letter or underscore.
xdigit	Matches hex digits 0-9 and A-F.

More examples,

```
$ rm file0[23].c      Deletes file02.c and file03.c but not file01.c.
$ rm file0[1-3].c     Match file01.c, file02.c, file03.c, but not a file named file04.c.
$ rm [^f]*            Match any file that does not begin with f.
$ rm [^af-ix-]*       Match any file that does not begin with a, f, g, h, i, x, or hyphen.
$ rm *[:digit]*       Match any file that has at least one digit in the file name.
$ rm *[0-9]*          Same as above.
$ rm [:word:]??       Match any three letter file name that begins with a letter or underscore.
```

Bash permits extended shell globbing if this option is enabled. The **shopt** command can be used to display, clear, and set various shell options[57]. To enable extended shell globbing,

```
$ shopt -s extglob              Extended globbing is now enabled
```

The -s option specifies to **shopt** that the specified option should be set, i.e., enabled. To disable an option, use -u,

```
$ shopt -u extglob              Extended globbing is now disabled
$ shopt -s extglob              Enable it again
```

With extended shell globbing enable, the following extended pattern matching operators are recognized (where *pattern-list* is a list of one or more patterns, as discussed above, separated by a bar |),

Extended Pattern	**What it Matches**
?(*pattern-list*)	Matches 0 or 1 occurrence of the patterns in *pattern-list*.
*(*pattern-list*)	Matches 0 or more occurrences of the patterns.
+(*pattern-list*)	Matches 1 or more occurrences of the patterns.
@(*pattern-list*)	Matches exactly one of the patterns.
!(*pattern-list*)	Matches anything except the patterns.

[56] The ASCII whitespace characters are: horizontal tab (ASCII 9), line feed (also called newline; 10), vertical tab (11), form feed (12), carriage return (13), and space (32).

[57] https://www.gnu.org/software/bash/manual/html_node/The-Shopt-Builtin.html

Examples,

`$ rm !(*.c|*.h)`	*Delete every file except files ending with* .c *or* .h.
`$ rm *(*)`	*Match zero or more occurrences of zero or more characters*[58].
`$ rm *+(a)*`	*Deletes any file containing one or more* a *characters.*
`$ rm !(foo)`	*Deletes every file in the current directory except for* foo.

5.10 More on Deleting Subdirectories

To delete all the files in a subdirectory and all of its subdirectories and then delete the subdirectory itself, use the -r (recursive) command option with the rm command,

```
$ rmdir cse220
rmdir: `cse220': Directory not empty
$ rm -r cse220                    this deletes cse220 and everything in it
$ ls cse220
ls: cannot access cse220: No such file or directory
```

rm -rf * is a very dangerous command,

```
$ cd
$ pwd
/home/fredf
$ rm -rf *
$ ls
$
```

This last rm command deletes everything in your home directory (all files, all directories, all subdirectories, and all files in all subdirectories). The option -f means force and will cause even files marked as read-only to be deleted[59].

5.11 Copying Files and Directories

To copy a file use the **cp** command,

```
$ cd
$ pwd
/home/fredf
$ cp file1.c file1.c.backup
$ ls
a.out file1.c file1.c.backup file3 src
$ mkdir cse220
$ cp file3 cse220
$ ls
```

[58] This is a stupid, but legal, pattern. For example, **rm *(*)** is the same as **rm ***.

[59] Note: The files in your home directory and all subdirectories on General are backed up every night to ~/Backup which you cannot delete. Therefore, if you *do* delete a file or files that you shouldn't have, you can retrieve yesterday's copy.

```
a.out cse220 file1.c file1.c.backup file3 src
$ ls cse220
file3
$ rm file3
$ ls
a.out cse220 file1.c file1.c.backup src
$ cd cse220
$ ls
file3
$ cp file3 ..
$ ls
file3
$ cd ..
$ ls
a.out cse220 file1.c file1.c.backup file3 src
```

To copy all the .c files from one directory to another,

```
$ cd
$ pwd
/home/fredf
$ mkdir src-backup
$ cd src
$ cp *.c ../src-backup
$ ls ../src-backup
file01.c file02.c file03.c file01.o file02.o file03.o
```

To copy all the files in a directory (and its subdirectories) to another directory, use the -r command line option with cp (-r stands for recursive),

```
$ cd
$ cp -r src src-backup
$ ls src-backup
file01.c file02.c file03.c file01.o file02.o file03.o
```

To copy a file from another directory to the current directory,

```
$ cp ../dir/file .
```

Note that the . is necessary in the command to specify the destination for the file that is being copied. Without it you will get an error message,

```
$ cp ../dir/file
cp: missing destination file operand after `/dir/file'
Try `cp --help' for more information.
```

5.12 Moving and Renaming Files

To move a file from one location to another use the **mv** (move) command. Note that move is different from copy in that copy "makes a copy" so the original file remains where it was at. Move actually "moves the file" from one place to another, so the original is removed.

```
$ cd
$ ls
a.out file1.c file3 src
$ mv file1.c src
$ ls
a.out file3 src
$ ls src
file01.c file02.c file03.c file01.o file02.o file03.o file1.c
$ mv src/file1.c ../../tmp
$ ls src
file01.c file02.c file03.c file01.o file02.o file03.o
```

To **rename** a file in *nix you move it to a new file, e.g., to rename file1.c to file01.c,

```
$ mv file1.c file01.c
```

5.13 Finding a File

The **find** command can be used to find a file that is stored somewhere in the file system. It is a very complicated command with lots of options, so I will only show you the most basic command. Suppose you know there is a file named mlocate.db somewhere in the file system but you don't know where. The command to type would be,

```
$ find / -name mlocate.db
```

This command tells the find program to start at the root directory (the / part) and search for a file a filed named (-name) mlocate.db. If you want to search for a file but you're not sure about the case of the filename, use the -iname option rather than -name to ignore the case during the search.

It should be noted that a *nix installation might have 10,000 or more files in the file system, so this command could run for quite a long time before it finds the file. If you have a hunch that the file you are looking for is in a subdirectory below root, say /usr, then you could narrow down your search by typing,

```
$ find /usr –name mlocate.db
```

Now, only the directory /usr and its subdirectories will be searched.

5.14 Displaying a File at the Terminal: Cat and Less

To displays the contents of a file at the terminal window, use the **cat** (concatenate) command,

```
$ cat file1.c
#include <stdio.h>

int main() {
    printf("Hello world.\n");
    return 0;
}
```

With cat, the contents of the file will scroll right off the top of the window if the file is too large. You can use the **less** command to display a file on the window and then use the Page Up (or b for backwards) and Page Down (or f for forwards) keys to scroll through the file.

```
$ less file1.c
file1.c is displayed on window
```

Frequently-Used Less Commands

h	display help
q	quit less
g	go to first line of file
G	go to last line of file
=	display name of file and current line
*n*g	go to line number n
/	search forward for a pattern
?	search backward for a pattern
n	repeat search in same direction
N	repeat search in opposite direction
!cmd	run the shell and execute cmd
v	load current file in vi

5.15 Archiving Files

An archive is a collection of one or more files grouped together into a single file. The program **tar** (tape archive) is used in *nix to form an archive. This archive file is commonly called a **tar file**. Note that the tar file is not compressed. A compressed or uncompressed tar file is also commonly called a **tarball**. We will see in the next section how to compress a tar file. To form a tar archive of one or more files,

```
$ cd
$ ls
a.out file1.c file3 src
```

```
$ tar cvf archive.tar file1.c file3
file1.c
file3
$ ls
a.out archive.tar file1.c file3 src
```

Note that the "tarred" files are not removed. This tar command would create an archive file named archive.tar which contains the files file1.c and file3. The v option to tar specifies that the program be verbose and display the files it is archiving as they are being archived. The c option means create an archive file. The f option means that what follows the options (the word "archive.tar") is to be the name of the archive file. It is common to give this file a .tar extension. Following the name of the archive file is a list of one or more file names (or directories) to be archived. To tar all of the .c files in a directory,

```
$ ls src
file01.c file02.c file03.c file01.o file02.o file03.o
$ cd src
$ tar cvf archive-c.tar *.c
file01.c
file02.c
file03.c
$ ls
archive-c.tar file01.c file02.c file03.c file01.o file02.o file03.o
```

To list the contents of a tar archive to the screen, i.e., to see what is in the archive, use the t option (t stands for table of contents),

```
$ tar tf archive-c.tar
file01.c
file02.c
file03.c
```

To extract the files from a tar archive use the x (extract) option,

```
$ cd
$ mkdir gromulate
$ mv src/archive-c.tar gromulate
$ cd gromulate
$ tar xvf archive-c.tar
file01.c
file02.c
file03.c
$ ls
file01.c file02.c file03.c
```

The file names will be displayed as each of them is extracted (because of the v option). Tar is often use to archive an entire directory and its subdirectories,

```
$ cd
$ ls src
file01.c file02.c file03.c file01.o file02.o file03.o
$ tar cvf src.tar src
file01.c
file02.c
file03.c
file01.o
file02.o
file03.o
$ ls
a.out file1.c file3 src src.tar
$ tar tf src.tar
file01.c
file02.c
file03.c
file01.o
file02.o
file03.o
```

5.16 Compressing Files

The most common file compression programs in *nix are **gzip** and **bzip2**. To compress a file with gzip,

```
$ gzip file1.c
$ ls
a.out file1.c.gz file3 src
```

This compresses the file and leaves file1.c.gz in the current directory; the original file file1.c is replaced by the .gz file. To decompress the file,

```
$ gzip -d file1.c.gz
$ ls
a.out file1.c file3 src
```

This decompresses the file and leaves file1.c in the current directory; the .gz file is removed. The program bzip2 is an alternative compression program which uses a different algorithm than gzip; often compressing with bzip2 will result in a smaller compressed file than if gzip were used. It works similarly to gzip,

```
$ bzip2 file1.c
```

This compresses the file and leaves file1.c.bz2 in the current directory; the original file file1.c is replaced by the .bz2 file. To decompress the file,

```
$ bzip2 -d file1.c.bz2
```

This uncompresses the file and leaves file1.c in the current directory; the .bz2 file is removed. To compress every .c file in a directory,

```
$ cd
$ ls src
file01.c file02.c file03.c file01.o file02.o file03.o
$ cd src
$ gzip *.c
$ ls
file01.c.gz file02.c.gz file03.c.gz file01.o file02.o file03.o
```

To decompress every gzipped .c file in a directory,

```
$ gzip -d *.c.gz
$ ls
file01.c file02.c file03.c file01.o file02.o file03.o
```

To zip a tar file, just gzip or bzip2 it like a normal file,

```
$ ls src
file01.c file02.c file03.c file01.o file02.o file03.o
$ tar cf src.tar src
$ ls
a.out file1.c file3 src src.tar
$ bzip2 src.tar
$ ls
a.out file1.c file3 src src.tar.bz2
```

It is possible to create a tar file and compress it after it is created using just one command and either the argument j (to bzip2 the tar file) or z (to gzip the tar file),

```
$ ls src
file01.c file02.c file03.c file01.o file02.o file03.o
$ tar cjf src.tar src
$ ls
a.out file1.c file3 src src.tar.bz2
$ tar czf src.tar src
$ ls
a.out file1.c file3 src src.tar.bz2 src.tar.gz
```

5.17 Hexdump

A **hex dump** is a display of the contents of a file in hexadecimal format. When we want to "see inside" a binary file, we often will use a program that produces a hex dump. In GNU/Linux the program is, aptly, named **hexdump**. In its simplest form, you would type **hexdump *file*** to view the contents of *file* in hex. More commonly, it is useful to use "canonical" mode by typing a -C option, e.g., **hexdump -C *file***. Note that hexdump will display the entire contents of the file on the terminal; you will probably want to pipe the output through less, for example, **hexdump -C *file* | less**. There are many other options you can explore by reading the man page.

6 Getting Help

6.1 Man Pages

Historically the online Unix manual was divided into eight sections:

1. Commands
2. System Calls
3. Library Functions
4. Special Files
5. File Formats
6. Games
7. Miscellaneous Information
8. System Administration

The **man** (manual) command can be used in *nix to get help on many commands. For example, type **man cd** to display the "man page" for the cd command. Typing **man kill** would display information on the **kill** command from section 1 of the manual. However, if there is also information about "kill" in section 2, one could type **man 2 kill** to display that entry (on some systems you may have to enter **man -s 2 kill** instead).

When reading man pages you may see a command or phrase followed by a number in parentheses, such as chmod(2). This tells you that man information about chmod can be found in section 2.

Man output is displayed with **less** so you can use the Page Up and Page Down keys to scroll through the manual page (or use the b and f keys; b for backward; f for forward). Hit q to get out of **less**.

6.1.1 Reading a Man Page

Man pages can be difficult to read sometimes because of the complexity of the commands and the various options and arguments. Nonetheless, if you really want to become a proficient *nix user, you will have to learn how to read them. There is no standard format for a man page, but many of them are organized in a a similar format,

Heading	**Meaning**
Name	name and purpose of the command
Synopsis	syntax of the command; shows whether it accepts options or args
Description	full description of the command; may be quite lengthy (e.g., bash)
Environment	environment variables used by the command
Author	the person or persons who wrote the program
Files	list of important files to this command
Copyright	who holds the copyright to the program
See Also	where to look for related information

Heading	**Meaning**
Diagnostics	possible errors and warnings
Bugs	known mistakes or shortcomings

For example, let's consider the man page for the cp command, shown below,

```
CP(1)                                                    User Commands

NAME
       cp - copy files and directories

SYNOPSIS
       cp [OPTION]... [-T] SOURCE DEST
       cp [OPTION]... SOURCE... DIRECTORY
       cp [OPTION]... -t DIRECTORY SOURCE...

DESCRIPTION
       Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

       Mandatory arguments to long options are mandatory for short options
       too.

       -a, --archive
              same as -dpR

       --backup[=CONTROL]
              make a backup of each existing destination file

       -b     like --backup but does not accept an argument

       --copy-contents
              copy contents of special files when recursive

       -d     same as --no-dereference --preserve=links

       -f, --force
              if an existing destination file cannot be opened, remove it and
              try again

       -i, --interactive
              prompt before overwrite

       -H     follow command-line symbolic links in SOURCE

       -l, --link
              link files instead of copying

       -L, --dereference
              always follow symbolic links in SOURCE

       -P, --no-dereference
              never follow symbolic links in SOURCE

       -p     same as --preserve=mode,ownership,timestamps

       --preserve[=ATTR_LIST]
              preserve  the  specified  attributes  (default: mode,ownership,
              timestamps),  if possible additional attributes: context, links,
              all
```

```
--no-preserve=ATTR_LIST
       don't preserve the specified attributes

--parents
       use full source file name under DIRECTORY

-R, -r, --recursive
       copy directories recursively

--remove-destination
       remove each existing destination file before attempting to open
       it (contrast with --force)

--sparse=WHEN
       control creation of sparse files

--strip-trailing-slashes
       remove any trailing slashes from each SOURCE argument

-s, --symbolic-link
       make symbolic links instead of copying

-S, --suffix=SUFFIX
       override the usual backup suffix

-t, --target-directory=DIRECTORY
       copy all SOURCE arguments into DIRECTORY

-T, --no-target-directory
       treat DEST as a normal file

-u, --update
       copy only when the SOURCE file is newer than the destination
       file or when the destination file is missing

-v, --verbose
       explain what is being done

-x, --one-file-system
       stay on this file system

--help
       display this help and exit

--version
       output version information and exit

By default, sparse SOURCE files are detected by a crude heuristic and
the corresponding DEST file is made sparse as well. That is the be-
havior selected by --sparse=auto. Specify --sparse=always to create a
sparse DEST file whenever the SOURCE file contains a long enough se-
quence of zero bytes. Use --sparse=never to inhibit creation of sparse
files.

The backup suffix is `~', unless set with --suffix or SIMPLE_BACKUP_SUF
FIX. The version control method may be selected via the --backup op-
tion or through the VERSION_CONTROL environment variable. Here are the
values:
       none, off
              never make backups (even if --backup is given)

       numbered, t
              make numbered backups

       existing, nil
              numbered if numbered backups exist, simple otherwise

       simple, never
              always make simple backups

       As a special case, cp makes a backup of SOURCE when the force and
       backup options are given and SOURCE and DEST are the same name for an
       existing, regular file.

AUTHOR
       Written by Torbjorn Granlund, David MacKenzie, and Jim Meyering.

REPORTING BUGS
       Report bugs to <bug-coreutils@gnu.org>.

COPYRIGHT
       Copyright (C) 2008 Free Software Foundation, Inc.  License GPLv3+: GNU
       GPL version 3 or later <http://gnu.org/licenses/gpl.html>
       This is free software: you are free to change and redistribute it.
       There is NO WARRANTY, to the extent permitted by law.

SEE ALSO
       The  full  documentation  for  cp is maintained as a Texinfo manual.
       If the info and cp programs are properly installed at your site, the
       command

              info cp

       should give you access to the complete manual.
```

The following sections are present in this man page: Name, Synopsis, Description, Author, Reporting bugs, Copyright, See Also.

The Name section is very simple: it states that the name of the command is **cp** and that it performs the action, "copy file and directories."

The Synopsis section states that there are three different "forms" of the cp command. The brackets indicates that the item is optional and the ellipses indicate or more (as in one or more). Therefore each of these three forms of the cp command start with the letters "cp" followed by zero or more options.

The first form (cp [OPTION]... [-T] SOURCE DEST) states that the options can be followed by a -T option and then a source (SOURCE) and a destination (DEST). To know what -T does you have to read about it in the Description section. There it tells us that -T tells cp to treat DEST as a normal file and not as a destination directory. Thus we are copying SOURCE to a file and not to a directory. What is SOURCE? It's a single file. An example of this command would be **cp -T *afile* ../someotherfile**. Note that if ../someotherfile is a directory, you will get an error message.

The second form (cp [OPTION]... SOURCE... DIRECTORY) states that multiple source files can be copied to a single directory. Note that the directory must be the last argument on the command line. An example of this command would be **cp file1 file2 file3 somedir**.

The final form (cp [OPTION]... -t DIRECTORY SOURCE...) requires the -t option. Looking in the Description section we see that -t means, "copy all SOURCE arguments into DIRECTORY". The target directory is specified following the -t option, and the list of one or more source code files is on the end of the command line. An example of this command would be **cp –t ../../somedir file1 file2 file3**.

Next is the Description section. This is usually the lengthiest section in a man page. Among other things, the Description section will provide a description of each of the options. We're not going to go through all of them here, but let's just look at a couple of them.

The -i (or --interactive) option will prompt you before it overwrites the destination file, e.g.,

```
$ cp –i file1 file2
cp: overwrite `file2'?
```

At this point you hit 'y' or 'n' to either allow or disallow the overwrite.

The -u (--update) option will copy the source to the destination only when the source is newer or when the destination is missing.

The Author section is self-obvious. Note that if you find a genuine bug in the cp program, you are told whom to contact in the Reporting Bugs section. Finally, the See Also section tells you that there is more information about cp in info (described Section 5.3).

6.2 Apropos

Sometimes you know what you would like to do but you are not sure what command to use. For example, suppose you want to rename a file but you don't know the command. You could type **man -k rename** and man would display a list of manual pages with a one-line description that contains the word "rename". You could look through that list to see if there is an entry for a command that renames files. The **apropos foo** command is equivalent to **man -k foo**. Note that apropos is pronounced "ap-pruh-po" and not "ap-pruh-pose".

6.3 Info

Info is an online help system, separate from the *nix manual, that is used to document GNU utilities. To get help on the bzip2 command you would type **info bzip2**. Where info differs from man is that info documentation is organized into a hierarchical document of nodes (it's a tree structure) and the user uses keypresses to navigate around the document. Info is, like the Emacs editor, incredibly complicated to use at first. Important info commands are summarized below.

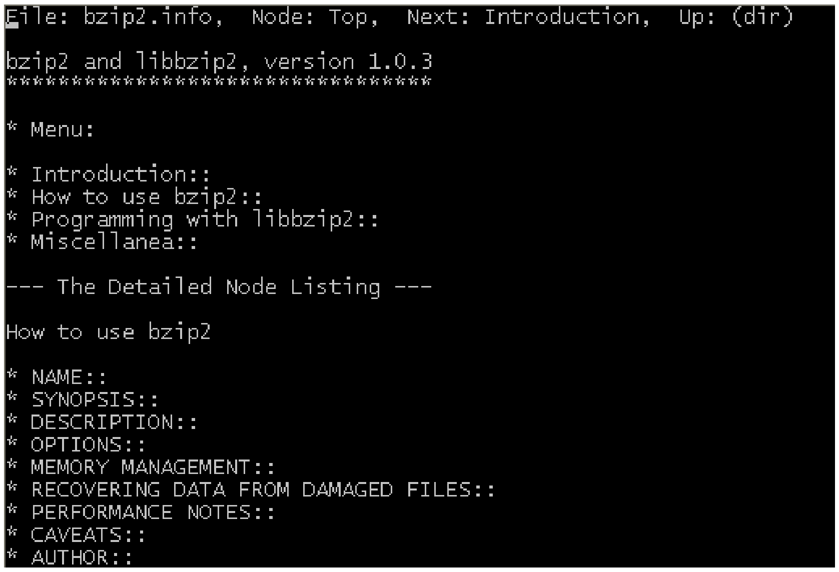
General Commands	
q	quit
z	start help tutorial
?	display command summary
^x 0	get out of the command summary window pane (that's a zero)

Reading a Node	
Pg Dn	display next screenful of text
Pg Up	display previous screenful of text
b	jump to first line of text in node
Up	move cursor up one line of text
Down	move cursor down one line of text
Left	move cursor one char to the left
Right	move cursor one char to the right

Moving Around Nodes	
n	jump to next node in document
p	jump to previous node in document
t	jump to the top node in the document
u	move up from this node to the parent node
Home	move to top line of node
End	move to bottom line of node

Jumping From Node File to Another	
Tab	move cursor to next hyperlink in current node
Return	follow hyperlink to a new node
l	jump to previously displayed node
d	jump to directory node (the root node of info)

Hyperlinks to other nodes are of the form * node:. For example, shown on the next page is part of the info node for the bzip2 command. The first hyperlink in this node is * Menu:. The cursor is currently in column one of row 1. Pressing tab will move the cursor to the Menu link. Pressing Return, then, would display the "Menu" node.



```
File: bzip2.info,  Node: Top,  Next: Introduction,  Up: (dir)

bzip2 and libbzip2, version 1.0.3
*********************************

* Menu:

* Introduction::
* How to use bzip2::
* Programming with libbzip2::
* Miscellanea::

--- The Detailed Node Listing ---

How to use bzip2

* NAME::
* SYNOPSIS::
* DESCRIPTION::
* OPTIONS::
* MEMORY MANAGEMENT::
* RECOVERING DATA FROM DAMAGED FILES::
* PERFORMANCE NOTES::
* CAVEATS::
* AUTHOR::
```

6.4 Final Notes

There is extensive online documentation about Unix, GNU, and GNU/Linux as well as Bash and other shells. An excellent starting point for learning more about GNU/Linux is the Linux Documentation Project[60].

The Bash reference manual is available from GNU[61]. A good reference on various *nix commands can be found at [62]. All of the man pages are online at various cites; just use Google to locate them. The GNU Compiler Collection[63] (GCC), is the standard suite of compilers used in the *nix world. The manuals for GCC are available at [64].

60 http://tldp.org
61 http://www.gnu.org/software/ bash/manual/bashref.html
62 http://www.computer hope.com/unix.htm
63 http://gcc.gnu.org
64 http://gcc.gnu. org/onlinedocs

7 Environment Variables, Search Path, Prompt

The shell maintains several **environment variables**[65] which are used to store information used by the shell. The PATH environment variable lists the directories in the order that the shell will search when trying to locate an executable file. You can display the environment variables with the **set** command (the **env** and **printenv** commands do the same thing),

```
$ set
a lot of stuff scrolls by
$ set | less
```

In the second set command, the output is being piped through the less program. We will discuss piping in more detail in Chapter 8. To change the value of an environment variable, use ***varname=value***,

```
$ MYNAME=Kevin
$ set | grep MYNAME
MYNAME=Kevin
```

To add the directory /sbin to your path,

```
$ PATH=/sbin:$PATH
$ set | grep PATH
PATH='/sbin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin'
```

Setting an environment variable at the Bash command prompt only changes the value of the environment variable for this session. To make the change permanent (so the variable will be defined when you log in again), edit your **.bash_profile** file in your home directory, and add the lines,

```
MYNAME=Kevin; export MYNAME            # or: export MYNAME=Kevin
PATH=/sbin:$PATH; export PATH          # or: PATH=/sbin:$PATH
```

The special file .bash_profile contains commands that are executed whenever a "new environment" is created, i.e., whenever you log in[66]. The next time you log in the environment variables will be updated. To delete an environment variable, use the **unset** command, e.g., unset MYNAME.

[65] http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_03_02.html

[66] The file **.bash_profile** contains commands that are executed when you log in to the *nix system. It differs from **.bashrc** in that the commands in .bash_profile are only executed once, during log in. On the other hand the commands in .bashrc are executed by the shell every time a new shell is started.

7.1.1 Putting . In Your PATH

When Bash is searching through the directories in PATH for a binary or shell script, it will not search your current directory (unless your current directory happens to be one of the PATH directories). It's not recommended[67] that you put . in your PATH, but it will keep you from having to remember to precede binaries and shell scripts that you create with "./". To add the current directory to PATH, you must edit .bash_profile by adding this line,

```
PATH=$PATH:.
```

Then log out, and log back in. This line appends "." to the end of the list of directories already set in the PATH environment variable. The shell will examine the directories in the order they are listed in the path when you type a command. It will execute the program from the directory in which it first finds the program executable file. Hence, because "." appears last, it will only run the program in the current directory if it is not present in any of the other directories in the PATH.

7.2 Changing your Prompt

The **shell prompt** is the string that is displayed by Bash when it is waiting for your input. By default your prompt may be some string ending with a $. You can change the prompt. An excellent online reference is at [68]. As an example, my prompt string is defined in my .bash_profile file as **PS1='\[\033[1;32m\]\w $\[\033 [0m\] '**. The environment variable PS1 is your main prompt string. This funky-looking command sets my prompt to display the current directory (the \w part) followed by a $ in bright green. Setting prompt colors is described at [69]. To permanently change your prompt in Bash, edit .bash_profile and enter a line such as,

```
PS1=prompt-string; export PS1          # or: export PS1=prompt-string
```

---

[67] Supposedly it could lead to a security breach.

[68] http://tldp.org/HOWTO/Bash-Prompt-HOWTO

[69] http://tldp.org/HOWTO/Bash-Prompt-HOWTO/x329.html

8 Searching Files

It is very common to want to search a file for a certain string. The **grep** (get regular expression) command[70] can be used to do this,

```
$ cd
$ grep MYNAME .bash_profile
MYNAME=Kevin
```

The string you are searching for is typed first and then you type the file name of the file in which you wish to search. To search for the string in every file of the current directory try **grep *string* *** (or **grep *string* .*** to include hidden files).

Grep can be used for more than simply searching a file or group of files for a fixed string. Grep allows searching files for patterns which are specified using a particularly obtuse syntax based on things called **regular expressions**. We are not going to discuss grep patterns in this course, but remember there is a lot of good online documentation that can be used to learn more. For example, see [71] or [72]. Grep is extremely useful at times and how to write regular expressions is an essential part of being a *nix user.

[70] http://en.wikipedia.org/wiki/Grep
[71] http://www.gnu.org/software/grep/doc/grep.html
[72] http://www. panix.com/~elflord/unix/grep.html

9 I/O and Redirection

Unix was created in the days of text-based computing before graphical user interfaces were common. One of the design philosophies was that users would interact with the system through small programs which performed some simple command or purpose. Dennis Ritchie and Ken Thompson designed these simple programs so they would accept input from the keyboard, a file, or as the output of another program. Thus, there was a need to be able to direct the output of a program to either a file or some other program and to also specify where a program was to get its input.

In *nix, the kernel opens and makes available three files for every program that is executed. These three files are called **standard input**, **standard output**, and **standard error**, usually abbreviated to **stdin**, **stdout**, and **stderr**.

By default, stdin is attached to the keyboard, and stdout and stderr to the console window. If a program prints something to stdout, then it will be displayed on the console window. If it reads something from stdin, the program will wait for input from the keyboard.

Stdin and stdout can be changed so that rather than reading input from the keyboard, the program will read input from a file. Traditionally in other operating systems of that era, this was done by changing the source code to read the data from a file rather than from the keyboard. One of the most beautiful things about Unix was that this could be accomplished without changing the source code. Rather, the user would redirect stdin from the keyboard to a data file when invoking the program and the source code would remain unchanged. The same could be done for stdout: rather than sending the output of a program to the console window it could be redirected to a file.

9.1 Redirection Example

An example may be helpful. The program **cat** normally reads a file and sends the output to the console window. When invoking cat we can redirect its output to a file rather than the console window,

```
$ cat file01.c > fooie
```

The > symbol is used to send cat's output to the file fooie rather than to the console window. Note that we have just made a copy of file01.c. The < symbol is used to redirect stdin. Consider a C program that reads in two integers and prints out the larger one.

```
/* larger.c */
#include <stdio.h>

int main() {
    int n1, n2;
    fscanf(stdin, "%d", &n1);
    fscanf(stdin, "%d", &n2);
    fprintf("The largest integer was the ");
    if (n1 > n2) fprintf(stdout, "first one: %d\n", n1);
    else fprintf(stdout, "second one: %d\n", n2);
    return 0;
}
```

Since this program reads its input from stdin and writes its output to stdout, we can redirect its input and output so it reads its input from a file and sends its output to a file. For example, suppose we create a text file named larger.in which has two lines, the first with a 2 and the second with a 4. The following command will execute our program and send its output to larger.out.

```
$ cat larger.in
2
4
$ larger < larger.in > larger.out
$ cat larger.out
The largest integer was the second one: 4
```

9.2 Redirection Example

For a different example, consider that cat normally sends its output to the console window. We can redirect cat's output to a file,

```
$ cat > larger.in
2                         user types 2 and presses enter
4                         user types 4 and presses enter
<Ctrl+D>                  user hits Ctrl and D keys simultaneously
$ cat larger.in
2
4
```

This is a commonly used quick-and-dirty way to create a small, simple text file without invoking a text editor.

9.3 Redirecting Stderr

Stderr can be redirected as well,

```
$ larger < larger.in > larger.out 2> larger.err
```

Note that < is really 0< and > is really 1>, so the above command is equivalent to,

```
$ larger 0< larger.in 1> larger.out 2> larger.err
```

Stderr is not used in C and C++ programs as much as stdout is. The idea with stderr is that error messages—which are special in that they are not part of the normal program output—can be sent to a different location than the normal output. For example, when running a program, you might send stdout to an output file and stderr to the console window,

```
$ blobx < data.in 1> data.out
```

Since the normal connection for stderr is to the console window, we only need to redirect stdout to the output data file. Any error messages printed to stderr will appear on the console window.

9.4 Sending Output to /dev/null

*nix interfaces to "devices" through files, e.g., the CD-ROM drive, the mouse, etc. A special device named /dev/null is created at startup. Anything written to /dev/null simply "goes away". So, we can use /dev/null and redirect stdout when, for example, we don't want to see the output of a program on the terminal window,

```
$ prog
output is displayed on the terminal
...
$ prog > /dev/null                          no output
```

Anything written to stdout within the program will be redirected to /dev/null and will not be displayed. Stderr can also be redirected to /dev/null.

9.5 Concatenating Files

Cat can read input from multiple input files and it will send its output to stdout,

```
$ cat file1
cookies
pizza
$ cat file2
dogs
cats
$ cat file3
red blue
$ cat file1 file2 file3 > file4
$ cat file4
cookies
pizza
dogs
cats
red blue
```

Here, the contents of the files file1, file2, and file3 have been concatenated together when creating the new file file4. Now you can see where the name cat came from (it is short for concatenate)

9.6 Sending Stdout and Stderr to the Same Place

Sometimes you want to send stdout and stderr to the same output file. This can be done using the command line,

```
$ blobx 1> data.out 2> data.out
```

The drawback is that you have to specify the same file name twice. To send stdout and stderr to the same file without having to specify the file name twice, use 1>&2 or 2>&1,

```
$ blobx 1> data.out 2>&1
```

The notation 2>&1 tells the shell to send stderr's output (that is, 2>'s output) to wherever stdout (that is 1>) is going. Note that if 2>&1 is used then 1> must appear before 2>&1. If 1>&2 is used (to send stdout to wherever stderr is going) then 2> must appear before 1>&2. If you do not do that, then something else happens. For example, $ blobx 2>&1 1> file.err would send stdout to file.err and stderr would go to wherever stdout normally goes, i.e., the console window.

9.7 Appending to a File

To append a file to another file use the >> symbol,

```
$ cat file1
cookies
pizza
$ cat file2
dogs
cats
$ cat file1 file2 >> file1
$ cat file1
cookies
pizza
cookies
pizza
dogs
cats
```

The notation >> is equivalent to 1>> and the notation 2>> can be used to concatenate to stderr.

9.8 Pipes

To connect the stdout of one program to the stdin of another program, in *nix we use a **pipe** (the output flows out one program through the pipe to the input of the other program). The pipe symbol is | and the way to send the output of one program to the input of another program is ***program1 | program2***. For example, to display a long listing of the files in a directory and sort them based on the time the file was last modified,

```
$ ls -l | sort +7
```

The program **sort** sorts its input (which is a text file) and sends the sorted output to stdout. The option +7 tells sort to sort the input file based on the data in field 7 which in the **ls -l** output is the column which contains the timestamp. To sort on size try **ls -l | sort + 4**. To display the contents of a directory and display the output using less,

```
$ ls -l | less
```

10 Processes

When you run a program in *nix, the OS loads it into memory and sets up some other information concerning your program. The loaded and running program is called a **process**. When the program terminates, so does the process.

Every process has a system-wide unique identifier called the **process id**, or **pid**; it is just an integer. The process is owned by the user who launched it. The **ps** command will show you a list of the running processes of which you are the owner.

```
$ ps
  PID TTY          TIME CMD
17776 pts/18  00:00:00 bash
19041 pts/18  00:00:00 ps
```

I am running two programs right now: bash and the ps program itself. The PID of each process is shown. The time column is the cumulative amount of CPU time the process has consumed—this is not the same as wall time, i.e., how many seconds it has been since you ran the program.

To see all of the processes running on the system, type **ps -A** or **ps -e**. The command **ps -f** will display full process information including the user id of the process owner. It can be combined with -A or -e. To sort this list by user id and view it in less, use **ps -ef | sort | less**.

When you run a program (i.e., a process) you can ask the OS to run it in the background. Do this by putting a & on the end of the command line,

```
$ some-program &
$ now-you-can-type-another-command
```

Now *some-program* will run in the background and you can get back to work typing commands. To bring a running program back into the foreground, type the **fg** command,

```
$ fg
```

Now *some-program* would be running and if it is displaying output, the output would be going to the console window. To put the program back into the background, hit Ctrl+Z to pause it, and then type **bg** to send it back to the background.

To see a list of your processes running in the background use the **jobs** command. If you have more than one background job they will be numbered with a job number 1, 2, 3, ... To bring job number 3 to the foreground, type **fg %3**. To send this job back to the background, hit Ctrl+Z, and type **bg**.

To terminate a runaway process, use the **kill** command. The syntax is **kill *pid-number***. You cannot kill processes that you do not own.

To kill a process running in the background, issue a jobs command to see your running processes and the determine the job number. Then use **kill %*job-number*** to kill the job; e.g., **kill %3** to terminate job number 3.

The **top** command will display a table of the processes which are currently using the most CPU time. It is a highly configurable program; try **man top**. (Be aware that top is a fairly resource-intensive program; it's not cool or nice to run top a lot).

The **nice** command was designed to allow a user to run a program with lower than normal priority, so it will not hog the CPU as much. The idea is that other people may be peeved if you're sucking up too much CPU time, so you can be nice about it and still run your program. The syntax is **nice -n *adjust command*** where *command* is the program you want to run (e.g., it could be a big make process). *Adjust* is an integer which determines the adjusted process priority: -20 is highest, to 19 lowest. No one ever uses nice.

11 Editing Files

11.1 Absolutely Painless Introduction to vi and vim

The **vi** text editor[73] is the standard editor available on most *nix systems. Some people who know it very well, love it. I don't know it very well; I don't love it. I've never taken the time to learn it very well because I would rather gouge my eyes out with infected hypodermic syringes found in the dumpster behind the plasma donation center until bloody pus oozes out of them than learn vi[74]; I simply do not have the time for it. Nonetheless, learning a bit of vi can be useful, if for nothing else, you may find yourself in a situation where it is the only editor that is available on the system you are on. At the least, knowing how to load a file, make some simple changes to it, and save it can be useful.

The version of vi found on most GNU/Linux systems is vim (vi improved)[75]. Purdue University has a good vi tutorial[76]. Also see [77].

11.1.1 Starting vi

At the Bash command prompt, type **vi** or **vi** ***some-filename*** (note: on General, this will actually load vim, but I will refer to it as vi in the following discussion). vi is a **modal editor** in which you are either in **editing mode** or **command mode**.

11.1.2 Getting out of vi

This is the most important thing you need to know about vi: how to get out of it. If you are in editing mode, hit ESC to enter command mode. Enter **:wq** to write your file and quit. To quit without writing the file, hit **:q**. If you really want to quit without saving the file you have made changes to hit **:q!**. To save the file under a different name, try **:wq** ***new-filename***.

11.1.3 Switching Between Editing Mode and Command Mode

If you are in editing mode, hit ESC to enter command mode. If you are in command mode hit **i** to enter insert (editing) mode. If you cannot tell which mode you are in, trying hitting ESC several times to make sure you are in command mode, then type **:set showmode**. This may tell vi to let you know when you are in insert mode (it depends on the version of vi you are using).

[73] http://en.wikipedia.org/wiki/Vi

[74] It would be preferable if I were also on fire and being eaten alive by rabid hyenas at the same time. That would *still* be more pleasurable than using vi.

[75] http://en.wikipedia.org/wiki/Vim_(text_editor). It's really not improved because it's still vi.

[76] http://engineering.purdue.edu/ECN/Support/KnowledgeBase/Docs/20020202121609

[77] http://thomer.com/vi/vi.html

11.1.4 Other Useful Settings

All of these are recommended in addition to **:set showmode**. In command mode, type **:set nocompatible** to enable advanced VIM features. Type **:set ruler** to tell vi to display your current cursor position. Type **:set number** to display line numbers.

11.1.5 Moving Around a Character at a Time

The original vi was written by Bill Joy (at UC Berkeley) using a computer system and keyboard that lacked arrow, page up, and page down keys. This is the reason for the stupid assignment of the keys for moving around: **h** (move left one character), **j** (move down one line), **k** (move up one line), and **l** (move right one character). On Joy's terminal, these four keys also had arrow keys on them and that is the historical reason they are still mapped that way today. However, if you have your terminal program, i.e., PuTTY, configured correctly, you should be able to use the arrow keys.

11.1.6 Moving Around a Word at a Time

Switch to command mode. Press **w** to move forward one word. Press **b** to move backward one word. Press ***n*w** to move forward *n* words, e.g., **3w** to move forward three words. Press ***n*w** to move backward *n* words.

11.1.7 Moving Around the File

Switch to command mode. To move to the end of the line, hit **$**. To move to the beginning, hit **0**. Hit **1G** to go the first line of text. Hit ***n*G** to go line number *n*. Hit **G** to go to the end of file. To display line numbers, in command mode type **:set number**. Note that **:13** is equivalent to 13G. To page down, hit Ctrl+F. To page up hit Ctrl+B.

11.1.8 Deleting Characters

Switch to command mode, move to the character you want to delete, and hit **x** to delete that character. To delete *n* characters hit ***n*x**, e.g., **17x** to delete 17 characters. To delete all the characters from the cursor position to the end of the line hit **D**.

11.1.9 Copying a Line of Text

To copy a line of text from one place in the file to another, switch to command mode. Move to the line you want to copy. Hit **yy** (yank something or other, I dunno). Move the cursor to the location where you want to make the copy. Hit **p** (lowercase p for paste) to put the copied line after the current line. Hit **P** (uppercase P) to put the copied line before the current line.

11.1.10 Moving a Line of Text

To move a line of text from one place in the file to another, switch to command mode. Move to the line you want to copy. Hit **dd**. Move the cursor to the location where you want to make the move. Hit **p** (lowercase p) to move the line after the current line. Hit **P** (uppercase P) to move the line before the current line.

11.1.11 Deleting Lines of Text

Switch to command mode. Move to the line you want to delete. Hit **dd**. Hit ***n*dd** to delete *n* lines of text, e.g., **3dd** will delete the line of text the cursor is on and the next two lines.

11.1.12 Cutting, Copying, Pasting Multiple Lines of Text

Use ***n*yy** to copy *n* lines of text to the copy buffer. Move to where you want the lines to be and hit **p** or **P** to copy them. Use ***n*dd** to delete *n* lines of text to the copy buffer, and then move to where you want to paste the lines and hit **p** or **P**.

11.1.13 Moving and Copying Text in Visual Mode

To cut, copy, and paste text that is not an entire line, enter visual mode by hitting **v** in command mode. Use the arrow keys to move around and select the text you want to copy or move. Once selected, hit **y** to copy the text to the copy buffer or **d** to delete the text to the buffer. Once copied (with y) or deleted with (with d) move to where you want the text to be. Hit **p** or **P** to paste the text.

11.1.14 Finding Text

Switch to command mode. Hit / (forward slash) and enter the string you are searching for, e.g., **/cookies**. The cursor will be placed on the first occurrence of cookies following the current location. Hitting **n** or **Enter** over-and-over will continue searching.

11.1.15 Find and Replace

Switch to command mode. To replace all occurrences of "homer" by "bart", try something like **:1,100 s/homer/bart/g**. The 1,100 part means search from line 1 to line 100 inclusive (I think 1,$ will search the entire file). The **s** stands for search. We are searching for "homer" and replacing all occurrences by "bart". The **g** stands for global which means it will replace all occurrences without prompting you for each one. If you want to be prompted for each one, use **c** instead (for confirmation).

11.1.16 Undo and Redo

Since you will use it the most because of vi's horrible user interface the most useful command in vi (other than **:q!**) is the undo command. Hitting **u** will undo your last change. Hitting **u** multiple times or useing ***n*u** will perform multiple undo's. Redo is Ctrl+R.

11.1.17 More Help

Let's say you're either insane or a masochist and want to learn more about vi. In command mode, hit **:h** to bring up help. The most useful thing you need to know here is how to get out of help, hit **:q** (like in less).

11.2 KDE - The Kate Editor

Kate (KDE Advanced Text Editor) is the default GUI text editor for KDE[78]. Its features include color syntax highlighting, search and replace, spell checker, copy-and-paste, adding comments, etc. To run Kate at the Bash command prompt type **kate** or **kate *file-name*** or look for it in your system menu. The Kate editor window will appear. From here on you can probably figure out what you need to know about Kate as it does not have a user interface which is as absolutely awful as vi.

11.3 GNOME - Gedit

The default GNOME text editor is **Gedit**[79]. To run gedit at the Bash command prompt type **gedit** or **gedit *filename*** or look for it in your system menu. The gedit editor window will appear. From here on you can probably figure out what you need to know about gedit as it, too, does not have a user interface which is as absolutely awful as vi.

11.4 Other Linux Editors

There are a plethora of text editors for GNU/Linux and Unix: vi, vim, Kate, Kedit, Gedit, Joe, Pico, Nano, Elvis, Emacs, X Emacs, Nedit, FTE, it goes on and on and on. Some are text-based (vi, vim, Nano, Emacs) and others GUI-based (Kate, Gedit, X Emacs).

If you installed Linux Mint as described in Chapter 2, other GUI-based text editors that you may wish to explore include[80]: Editra, jEdit, KWrite (a reduced-feature version of Kate), Medit, SciTE, Scribes, and Tea. A good place to start exploring text editors is [81], [82], and [83]

Editors are a matter of personal preference. Some people feel very strongly about the merits of their personal preference. Some vi people will never quit talking about how great vi is and how much Emacs sucks. There are Emacs people who will tell you that Emacs is the only editor in the Universe and God himself uses Emacs[84]. If you have the need or desire to learn to use another editor go ahead, but honestly, **ed**[85] is all you need.

---

[78] Kate is not installed unless you select a KDE desktop distro. However, in Mint—and probably other distributions as well—it can be installed later using your package management program.

[79] Gedit is not installed unless select a GNOME-based desktop distro. However, it can usually be installed later.

[80] In Synaptic, navigate to the Editors (universe) package.

[81] http://en.wikipedia.org/wiki/Comparison_of_text_editors

[82] http://www.thefreecountry.com/programming/editors.shtml

[83] https://en.wikipedia.org/wiki/Category:Linux_text_editors

[84] http://en.wikipedia.org/wiki/Editor_wars

[85] http://en.wikipedia.org/wiki/ Ed_(Unix)

12 Basic Bash Shell Scripting

One of the more powerful features of Unix systems and the shells is the ability to write shell programs, i.e., programs which are executed by the shell interpreter in order to perform useful commands or perform some task. These programs are called **shell scripts**, **scripts**, or **script files**. Shell scripts are a way of automating commonly performed commands or a complex set of commands.

A good basic introduction to Bash shell scripting is available at [86] and a more advanced guide at [87]. Complete books have been written about shell scripting, and it is not something we can cover in a day or two and five pages of notes. This will be a basic introduction, and you are referred to the online guides and tutorials for more information.

A script file is a text file, and the first line is usually supposed to be #!/bin/bash if the shell script is designed to be executed by the Bash shell. If you are using a different shell to write your script (e.g., the Korn shell), then set the first line to point to the Korn shell's binary image (usually it is in the /bin directory; e.g., /bin/ksh). This will ensure that the correct shell is loaded and used to execute the shell script regardless of what particular personal shell the user may be running.

Note that in shell scripts, # lines are comments, just as // are comments in C++. Any character from the # symbol to the end of the line is ignored by the shell interpreter.

12.1 Variables

Shell **variables** are used to store values, which are always strings. A variable can be defined at the same time it is assigned by the = operator, with **let** being optional,

```
x=33                                 # Same as let x=33
let y=12                             # Same as y=12
let z = 95                           # Error! No spaces are allowed.
first_name=Fred                      # Or: "Fred" since "" are optional
last_name="Flintstone"               # Or: "Flintstone"
friend="Barney Rubble"               # "" are required because of whitespace
full_name=${first_name}${last_name}  # full_name is assigned "FredFlinstone"
somevar=                             # Sets somevar to null
```

When assigning 95 to *z* it is important to *not* surround the = operator with spaces; that is a Bash syntax error. Note that "" marks are necessary for "Barney Rubble" because the string being assigned contains whitespace. When using the value of a variable, it must be preceded by a $, and it is recommend to enclose the variable name with braces as we did with ${first_name} and ${last_name}.

[86] http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
[87] http://tldp.org/LDP/abs/html

12.2 Capturing Command Output to a Variable

The **backquote operator** will execute the enclosed command and assign the output to the designated variable,

```
ls_out=`ls`
echo ${ls_out}
displays the contents of the current directory
```

An alternative and newer way to do this is to enclose the command in $(...), e.g.,

```
ls_out=$(ls -l /bin)
echo ${ls_out}
displays a long listing of the contents of /bin
```

12.3 Bash Operators

The standard **operators** are available as shown in the table, from highest precedence down to lowest precedence.

id++ *id*--	Postincrement, Postdecrement
++*id* --*id*	Preincrement, Predecrement
-*expr* +*expr*	Unary minus (negation), plus
!*expr* ~*expr*	Logical negation, bitwise negation
**	Exponentiation
* / %	Multiplication, division, remainder
+ -	Addition, Subtraction
<< >>	Left and right bitwise shifts
> < >= <=	Grtr than, Less than, Grtr than or equal, Less than or equal
= (or ==) !=	Equal, not equal
&	Bitwise and
^	Bitwise exclusive or
|	Bitwise or
&&	Logical and
||	Logical or
expr ? *expr* : *expr*	Conditional
*= /= %= += -= <<= >>= &= ^= |= =	Combined assignment, Assignment (=)

12.4 Basic Shell Script Examples

A script file can contain regular Bash commands, e.g., **rm**, **cp**, **mv**, **ls**, etc. A script file can also define variables and assign them values. As we will see shortly, the Bash shell programming language is quite sophisticated and allows **if** statements, **for**, **while**, and

until loops, and procedures, i.e., functions. Here is a sample shell script which moves to directory *tmp* and deletes all of the .o files (the script file is named *rmo.sh*),

```
Use VI or some other editor to edit and save rmo.sh
-----------------------------------------------------------------------------
#!/bin/bash
# rmo.sh
# usage: rmo.sh
mv tmp
rm *.o
-----------------------------------------------------------------------------
$ ./rmo.sh
./rmo.sh: Permission denied
```

The shell script failed to run because shell scripts are not executable files by default. To make the shell script file executable, you must change the permissions using the **chmod** command to add **x**,

```
$ ls -l rmo.sh
-rw-r--r--  1 kburger2 kburger 136 Feb  4 11:35 rmo.sh
$ chmod 755 rmo.sh                                        or chmod +x rmo.sh
$ ls -l rmo.sh
-rwxr-xr-x  1 kburger2 kburger 136 Feb  4 11:35 rmo.sh
```

It is also kind of a nuisance to have to type **./rmo.sh** to execute the script. An **alias** is another name for something. In Bash, you can define aliases for commands using the syntax **alias *name*=*command***,

```
$ alias home="cd ~"        "" required since rhs contains whitespace
$ pwd
/usr/bin
$ home
$ pwd
/home/fredf
```

Aliases defined on the command line only exist for the current Bash session. If you want the alias name to be available every time you log in, define the alias in *.bash_profile*. To delete an alias that was defined at the command prompt, use the **unalias** command, e.g., **unalias rmwild**. To delete all of your aliases, type **unalias -a**. With the **alias** command, we can create an easier-to-type alias for **./rmo.sh**,

```
$ alias rmo=./rmo.sh
$ rmo
rm: cannot remove `*.o': No such file or directory
```

The error message was displayed because there were no .o files in the current directory. If you do not want **rm** to complain about this, use the -f option, i.e., **rm -f *.o**. A slightly more sophisticated program might use variables.

```
-----------------------------------------------------------------------------
#!/bin/bash
# rmo.sh
# usage: rmo.sh
_dirname=tmp              # Or: "tmp"
_pattern=*.o              # Or: "*.o"
cd ${_dirname}
rm -f ${_pattern}
-----------------------------------------------------------------------------
```

Here *_dirname* and *_pattern* are variables. Note that when the variable is assigned (using =), we do not put a $ in front of the variable name, but we do use a $ when referring to the variable later on. If you don't, the shell will think that the variable name is the file-name of a program to be executed and it will probably fail with an error message. It is also a good idea to enclose the variable name in braces, although in most cases, it is not necessary,

```
-----------------------------------------------------------------------------
#!/bin/bash
# rmo.sh
# usage: rmo.sh
_dirname=tmp
_pattern=*.o
cd $_dirname         # Will work just fine
rm -f $_pattern      # Ditto
-----------------------------------------------------------------------------
```

This script captures output from the **date** command into variables *dow*, *month*, *day*, and *year*, and then displays the date in our format,

```
-----------------------------------------------------------------------------
#!/bin/bash
# today.sh
dow=$(date +%a)
month=$(date +%b)
day=$(date +%d)
year=$(date +%Y)
echo "Today is ${dow} ${month}-${day}-${year}"
-----------------------------------------------------------------------------
$ chmod +x today.sh
$ alias today=./today.sh
$ today
Today is Wed Oct-17-2007
```

12.5 Bash Operator Usage

Variables can be assigned integer values[88] using normal assignment, or using **let**,

```
x=33
let y=12
let z = 95      Error! Never surround operators with spaces
```

The last statement will cause the shell script to fail; there should be no spaces to the left and right of the = operator. Expressions are evaluated by enclosing them in $((...)) to form a **compund command**,

```
------------------------------------------------------------------------------
#!/bin/bash
# op.sh
# usage: op
x=33
y=12
z=$(( ${x} / ${y} ))                # Note: Integer division is performed
echo ${z}                           # Displays 2
z = $(( (${y}**2 + (2 - ${x})) % 5) # Displays 3
cd /bin && ls                       # Will perform ls since cd succeeds
cd /foobar && ls                    # Will not perform ls since cd fails
------------------------------------------------------------------------------
$ chmod +x op.sh
$ alias op=./op.sh
$ op
2
3
contents of /bin is displayed
./op.sh: line 10: cd: /foobar: No such file or directory
```

12.6 Command-Line Arguments

When a Bash shell script is launched, command line **arguments** may be passed,

```
$ ./mymake.sh foo bar install
```

Here *mymake.sh* is the name of the shell script file, and *foo*, *bar*, and *install* are arguments. Special shell variables, ${0}, ${1}, ${2}, …, are used to refer to the command line arguments. The variable ${0} refers to the name of the shell script (*./mymake.sh*), ${1} the first argument (*foo*), ${2} the second argument (*bar*) and so on. ${#} is the number of command line arguments following the script filename. Our *rmo.sh* script can be rewritten[89],

[88] To be more accurate, all Bash variables store strings, but a variable can be assigned the string representation of an integer.

```
------------------------------------------------------------------------------
#!/bin/bash
# rmwild.sh
# usage: rmwild.sh dir pattern
_dirname=${1}
_pattern=${2}
cd ${_dirname}
rm -f ${_pattern}
------------------------------------------------------------------------------
$ alias rmwild=./rmwild.sh
$ rmwild tmp *.o
```

Or more simply,

```
------------------------------------------------------------------------------
#!/bin/bash
# usage: rmwild.sh dir pattern
cd ${1}
rm ${2}
------------------------------------------------------------------------------
$ rmwild tmp *.o
```

12.7 Displaying Text from a Shell Script

An executing script can display text on the console window using the **echo** command. For example,

```
------------------------------------------------------------------------------
#!/bin/bash
# rmwild.sh
# usage: rmwild dir pattern
echo "Removing all ${2}| files from ${1}..."
cd ${1}
rm -f ${2}
echo "Done."
------------------------------------------------------------------------------
$ ls tmp
file1.c file1.o file2.c file2.o
$ rmwild tmp *.o
Removing all *.o files from tmp...
Done.
$ ls tmp
file1.c file2.c
```

12.8 A Simple Backup Script

This script will **tar** and **bzip2** compress the files in the specified directory which is located beneath the current directory,

89 Of course, this is a pretty silly script file, because it would be just as easy to type **rm tmp/*.o**, but don't focus on that; rather, focus on the syntax.

```
------------------------------------------------------------------------------
#!/bin/bash
# bu.sh
# usage: bu dir
tar cvf ${1}.tar ${1}
bzip2 ${1}.tar
mv ${1}.tar.bz2 mybackups
echo "${1} backed up."
------------------------------------------------------------------------------
$ alias bu="./bu.sh"
$ bu src
src backed up.
$ ls mybackups
src.tar.bz2
```

12.9 Reading From the Keyboard

To read a string from the keyboard and store it in a shell variable use the **read** command,

```
------------------------------------------------------------------------------
#!/bin/bash
# getname.sh
echo -n "Enter your name: "        # -n says do not print newline
read name
echo "Pleased to meet you ${name}."
------------------------------------------------------------------------------
$ chmod +x getname.sh
$ alias getname=./getname.sh
$ getname
Enter your name: Fred
Pleased to meet you Fred.
```

To display a prompt, use the -p option,

```
------------------------------------------------------------------------------
#!/bin/bash
# getname.sh
read -p "Enter your name: " name
echo "Pleased to meet you ${name}."
------------------------------------------------------------------------------
$ getname
Enter your name: Fred Confucious Flintstone
Pleased to meet you Fred Confucious Flintstone.
```

If you are reading a password, and you do not want the password characters to be displayed as the user is typing them, use the -s option,

```
------------------------------------------------------------------------------
#!/bin/bash
# getpass.sh
read -p "Password: " -s pwd
echo  # move cursor down to next line
echo "Shhhh. Don't tell anyone, but I know your password is ${pwd}."
-----------------------------------------------------------------------------
$ chmod +x getpass.sh
$ alias getpass=./getpass.sh
$ getpass
Password: user types, nothing is displayed
Shhhh. Don't tell anyone, but I know your password is foobar.
```

12.10 If Statements

The syntax of the **if** and **if-else** statements are,

```
if [ cond-expr ]; then              if [ cond-expr ]; then
    some-commands                       some-commands
fi                                  else
                                        some-other-commands
                                    fi
```

Note two important syntactic details. First, surround the brackets [ ] with spaces,

```
if [cond-expr]; then      Will not work. Must be [ cond-expr ];
```

Second, do not forget the semicolon preceding **then**. The result of the conditional expression is 0 or nonzero with 0 being false and nonzero being true. The character "[" actually refers to a built-in Bash command—it is a synonym for another command named **test**. For example, these two **if** statements are functionally equivalent,

```
if [ ${var}="foo" ]; then ...          if test ${var}="foo"; then ...
```

The six relational operators <, >, <=, >=, = (== may be used in place of =), and != may be used in conditional expressions, but it is very important to not surround the operator with spaces,

```
if [ ${var} = "foo" ]; then ...  # Wrong! Will always evaluate to true
```

This should be written as,

```
if [ ${var}="foo" ]; then ...    # Correct. Will be true if ${var} is "foo".
```

Alternatively, -lt, -gt, -le, -ge, -eq, and -ne may be used,

```
if [ ${var} -eq "foo" ]; then ...
```

The operators -n and -z may be used to test a string to see if it is nonempty or empty,

```
if [ -n ${var} ]; then ...      True if ${var} is nonempty
if [ -z ${var} ]; then ...      True if ${var} is empty
```

These tests would be equivalent to writing,

```
if [ ${var}!="" ]; then ...
if [ ${var}="" ]; then ...
```

The logical operators are && (AND), || (OR), and ! (NOT),

```
if [ ${var1}="foo" && ${var2}<4 ]; then ...
```

Alternatively, -a and -o may be used,

```
if [ ${var1}="foo" -a ${var2}<4 -o ${var3}!=2 ]; then ...
```

The relational and logical operators have the same precedence as in C and C++. Operators exist for checking the status of files,

`[ -d` *`file`* `]`	*True if* file *exists and is a directory*
`[ -e` *`file`* `]`	*True if* file *exists*
`[ -N` *`file`* `]`	*True if* file *exists and has been modified since it was last read*
`[ -r` *`file`* `]`	*True if* file *exists and is readable*
`[ -s` *`file`* `]`	*True if* file *exists and has size > 0*
`[ -w` *`file`* `]`	*True if* file *exists and is writable.*
`[ -x` *`file`* `]`	*True if* file *exists and is executable*
`[` *`file1`* `-nt` *`file2`*`]`	*True if* file1 *is newer than* file2 *or if* file1 *exists and* file2 *does not*
`[` *`file1`* `-ot` *`file2`*`]`	*True if* file1 *is older than* file2 *or if* file2 *exists and* file1 *does not*

Some examples,

```
------------------------------------------------------------------------------
#!/bin/bash
# if.sh
today=0
if [ ${today}=1 ]; the   # Important: do not surround = with spaces
    echo "today = 1"
else
    echo "today <> 1"
fi
------------------------------------------------------------------------------
$ chmod 755 if.sh
$ ./if.sh
today <> 1
```

Since using [ ] is equivalent to using **test** and since variables always store strings,

```
------------------------------------------------------------------------------
#!/bin/bash
# if.sh
today=1
if test ${today}="1"; then
    echo "today = 1"
else
    echo "today <> 1"
fi
------------------------------------------------------------------------------
$ ./if.sh
today = 1
```

In a shell script, the ! symbol is used as the NOT logical operator, e.g.,

```
if [ ! -d src ]; then
    echo '"'src'"' does not exist, creating directory.
    mkdir src
fi
```

To learn more about conditional expressions I suggest you try **man test** and **man bash**.

12.11 For Loops

Bash implements **for**, **while**, and **until**[90] loops. The basic syntax of the **for loop** is,

```
for var in list; do
    some-commands
done
```

Note the semicolon following *list*; it is important to not omit it. The commands are executed once for every item in the list. The current item is accessed through the variable $*var*,

```
------------------------------------------------------------------------------
#!/bin/bash
# for.sh
for i in 1 2 3 4 5 6 7 8 9 10; do
    echo -n "${i} "                      # -n means do not print newline
done
echo
------------------------------------------------------------------------------
$ chmod 755 for.sh
$ ./for.sh
1 2 3 4 5 6 7 8 9 10
```

[90] The **until** loop is equivalent to a C/C++ **do while** loop.

The option -n to echo means do not print a newline character after outputting the text. Another example,

```
for i in $(ls); do
    echo ${i}
done
```

This script display the name of each of the files in the current directory one per output line. Another example,

```
------------------------------------------------------------------------------
#!/bin/bash
# lsdir.sh
for i in *; do
    if [ -d "${i}" ]; then
        echo "${i} [dir]"
    else
        echo ${i}
    fi
done
------------------------------------------------------------------------------
$ mkdir foo1
$ mkdir foo2
$ chmod 755 lsdir.sh                      or chmod ugo=+x lsdir.sh
$ alias lsdir="./lsdir.sh"
$ ls
foo1 foo2 for.sh if.sh lsdir.sh
$ lsdir
foo1 [dir]
foo2 [dir]
for.sh
if.sh
lsdir.sh
```

The **seq** command (type **man seq** for help) can be used to generate an integer sequence. The syntax is,

seq *first increment last*

which will generate an integer sequence starting at *first*, incrementing by *increment*, and stopping at *last*. For example, here is a loop which displays the first 100 odd integers,

```
for i in $(seq 1 2 100); do      # Or write `seq 1 2 100`
    echo -n ${i} " "
done
```

The following for loop prints the first 100 odd integers in reverse order,

```
for i in $(seq 99 -2 1); do
    echo -n ${i} " "
done
```

For more on **for** loops consult [91].

12.12 While Loops

The basic syntax of the **while loop** is,

```
while [ conditional-expression ]; do
    statements
done
```

If you can figure out **for loops**, I'm pretty certain you can write **while loops**.

12.13 Expr and Substrings, Relational Operators

Sometimes in a script, you would like to evaluate an expression, e.g., to find the sum of two variables. We have seen that a compound command $((...)) will do the trick. The **expr** command can be also be used to do this,

```
------------------------------------------------------------------------------
#!/bin/bash
# sum.sh
sum=$(expr $1 + $2)       # sum = $(($1+$2)) would work as well
echo ${sum}
------------------------------------------------------------------------------
$ alias sum="./sum.sh"
$ sum 10 20
30
```

Typing **man expr** will display a help screen about the expr command. Arithmetic expressions can also be evaluated using the Bash $((...)) syntax,

```
i=1
j=$(( ${i}++ ))           j is assigned 1 and i is incremented to 2
j=$(( ${i}-- ))           j is assigned 2 and i is decremented to 1
j=$(( ++${i} ))           j is assigned 2 and i is incremented to 2
j=$(( --${i} ))           j is assigned 1 and i is decremented to 1
i=$(( ${i} + 5 ))         i is assigned 6
j=$(( ${i} ** 3 ))        j is assigned 6^3 = 216
i=$(( (${j} - 20) %5 ))   Subtract 20 from j and then return 196 mod 5
```

Refer to the **man bash** page and the *Arithmetic Evaluation* section for more discussion of Bash operators. The **expr** ***substr*** command can be used to extract a substring from a

[91] http://tldp.org/LDP/abs/html/loops1.html

string variable, e.g., this program prints the individual characters of a string stored in a variable *v*,

```
v="foobar"
len=$(expr length ${v})          # ${#v} will also eval to the length of v
for i in $(seq 1 1 ${len}); do
    echo -n $(expr substr ${v} ${i} 1)
done
echo
```

The syntax of **expr *substr*** is,

```
expr substr string pos len              ${string:pos:len} does the same thing
```

Which will extract *len* characters from the string *string* starting at position *pos*. Note that the first character of *string* is at position 1. The command **expr *index string charset*** returns the first index in *string* where the first character in *charset* is found, or 0 if no characters in *charset* occur in *string*. Note that the characters in *string* are in positions numbered starting at 1.

Strings may be compared using the <, >, <=, >=, ==, and != relational operators or the -lt, -gt, -le, -ge, -eq, and -ne operators. **expr *length*** can be used to find the length of a string. Examples,

```
------------------------------------------------------------------------------
#!/bin/bash
# testexpr.sh
echo $(expr index "Fred Flintstone" "F")
echo $(expr index "Fred Flintstone" "e")
echo $(expr length "Fred Flintstone")
v1=10
v2=20
lte=$(expr ${v1} \<= ${v2})   # precede < by \ because < redirs stdin
gt=$(expr ${v1} \> ${v2})     # precede > by \ because > redirs stdout
if [ ${lte} ]; then
    echo "Less than or equal is true"
fi
if [ ${gt} ]; then
    echo "Greater than is true"
fi
------------------------------------------------------------------------------
$ alias testexpr="./testexpr.sh"
$ testexpr
1
3
15
Less than or equal is true
```

12.14 Translate and Delete Command

tr is the translate or delete characters command. The basic syntax is,

```
tr SET1 SET2        translate
tr -d SET1          delete
```

where **tr** reads from stdin and writes to stdout. In the first form, the characters appearing in *SET1* of the the string read from stdin are translated to the characters appearing in *SET2*. The output is sent to stdout. If you wish to assign the output to a Bash variable, you can use the ***var*=$(command)** trick. In the second form, the -d option tells **tr** to delete all occurrences of the characters in *SET1* from the string read from stdin.

If a program reads input from stdin, but you wish to specify a string in a Bash variable as the input, use the command **<<<*var*** which sends the string stored in ${*var*} as stdin to the command. For example,

```
------------------------------------------------------------------------------
#!/bin/bash
# testtr.sh
string="Some String"
tr [S] [s] <<< ${string}            # sends "some string" to stdout
string=$(tr [S] [s] <<< ${string})  # sets string to "some string"
tr [Se] [eS] <<< ${string}          # sends "eomS etring" to stdout
tr -d [aeiouAEIOU\ ] <<< ${string}  # sends "SmStrng" to stdout
tr -d [\ ] <<< ${string}            # sends "SomeString" to stdout
full_name="Fred Flintstone"
tr [F] [x] <<< ${full_name}         # sends "xref xlintstone" to stdout
lower=$(tr [:upper:] [:lower:] <<< ${full_name})
echo ${lower}                       # sends "fred flintstone" to stdout
upper=$(tr [:lower:] [:upper:] <<< ${full_name})
echo ${upper}                       # sends "FRED FLINTSTONE" to stdout
------------------------------------------------------------------------------
$ alias testtr="./testtr.sh"
$ testtr
some string
eomS etring
SmStrng
SomeString
xred xlintstone
fred flintstone
FRED FLINTSTONE
```

Built-in sets include,

`[:alnum:]`	*All letters and digits* [A-Za-z0-9]
`[:alpha:]`	*All letters* [A-Za-z]
`[:blank:]`	*All horizontal whitespace, i.e., spaces and tabs*
`[:digit:]`	*All digits* [0-9]
`[:lower:]`	*All lowercase letters* [a-z]
`[:print:]`	*All printable characters, including space*
`[:space:]`	*All horizontal or vertical whitespace, i.e., spaces, tabs, newlines, etc.*
`[:upper:]`	*All uppercase letters* [A-Z]

Well, that is my basic introduction to Bash shell scripting. There is significantly more that could be discussed, but I trust if you can master the basic concepts, you will have no trouble learning the more advanced concepts some day.

13 Program Development, GCC, and GNU Make

*nix is the OS of choice for serious programmers because it was designed by, and for, programmers. Any OS that was designed so that my mother could easily use it (aka Windoze) is obviously of no use to someone who prefers to toggle his programs into the computer on the front panel[92] in binary.

Just as *nix is the OS of choice for serious software development, GCC is the compiler of choice for serious programmers.

A Real Computer—The Altair 8800 (it did not come with an OS; why would you need one when you've got all those LED's and toggle switches on the front panel?)

GCC is an acronym for the Gnu Compiler Collection (I used to think it meant GNU C Compiler; that's false). It is a free, open-source set of compilers for building software, and includes compilers for the C, C++, Objective-C, Ada, Go, Java, and FORTRAN languages.

GCC is part of the GNU **toolchain**[93] and is the name given to the set of programming tools produced by the GNU project for developing the GNU operating system. Programs included in the toolchain include:

GCC	Compilers for various languages.
GNU Make	A program which facilitates automated building of software.
GNU Binutils	Includes an assembler, linker, profiler, and an archiver.
GDB	The GNU debugger.

In this course we will focus on learning to use the GCC C/C++ compilers, GNU Make, and GDB. The compilers and GNU make will be discussed in this chapter. The GNU debugger GDB is discussed in the next chapter.

13.1 Using GCC

The online documentation[94] for GCC is worthwhile reading. In this section we will discuss how to use the GCC C compiler to compile a program written in C. The basic command syntax is,

```
$ gcc file.c
```

92 http://en.wikipedia.org/wiki/Front_panel
93 http://en.wikipedia.org/wiki/GNU_toolchain
94 http://gcc.gnu.org/onlinedocs

where file.c is the filename of the C program source code file. This command will compile the code and link it with the C libraries to create an executable file. This executable will typically be named a.exe in Cygwin and a.out in *nix. To run the program,

```
$ ./a.out            or just type a.out if you added . to PATH
```

13.1.1 Useful GCC Command Line Options

When you invoke the GCC C compiler to compile a C source code file, you can also specify command line options[95] which modify the default behavior of the compiler. In this section we will discuss some of the more useful command line options.

13.1.1.1 --version

Prints the version number of the compiler. Note the double hyphens.

```
$ gcc --version
gcc (GCC) 3.4.4 (cygming special, gdc 0.12, using dmd 0.125)
Copyright (C) 2004 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

The output above tells me that the version of the GCC C compiler I have installed is 3.4.4, and that it was compiled for the Cygwin/Mingw platform (I am using Cygwin right now).

13.1.1.2 -ansi

Tells GCC to compile the C code as if it conforms to ANSI Standard C. This is equivalent to the command line option: -std=c89. Non ANSI Standard C code should be rejected as syntactically incorrect.

13.1.1.3 -Dname[=def]

This option defines a preprocessor macro named *name*. These macros are interpreted by the C preprocessor[96]. One common use of preprocessor macros is to include code that is conditionally compiled only if the macro is defined. For example,

```
/* file.c */
int main() {
    #ifdef DEBUG
        printf("Calling init()\n");
    #endif
    init();
    #ifdef DEBUG
        printf("Calling process()\n");
```

[95] http://en.wikipedia.org/wiki/Command_line_argument
[96] http://en.wikipedia.org/wiki/Preprocessor

```
    #endif
    process();
    #ifdef DEBUG
        printf("Calling terminate()\n");
    #endif
    terminate();
    return 0;
}

$ gcc -DDEBUG file.c
```

If we compile with DEBUG defined, then the printf() statements will be compiled as well, so when we run the program, the debugging printf() statements will be executed. If we compile with DEBUG not defined, then the printf() statements will not be compiled. This is a handy way to put debugging printf() statements in the program and decide at compile time if we want them to be output or not.

As another example, consider writing C code for a program which depends on the endianness[97] of the bytes in an integer. Generally in the Intel x86 world, multibyte integers are stored in little-endian format. Motorola's PowerPC architecture uses big-endian. In order to make our C code portable (i.e., so it will run on both an x86 system or a PowerPC system), we might do something like the following:

```
/* file.c */
...
#if ARCH == x86
    /* little-endian specific code here */
#elif ARCH == ppc
    /* big-endian specific code here */
#endif
...

$ gcc -DARCH=x86 file.c
```

Or, in building the PowerPC version of the program,

```
$ gcc -DARCH=ppc file.c
```

13.1.1.4 -g

This option tells the C compiler to include additional debugging information in the executable file so we can debug the program using GDB. If you do not compile a file with -g, you can still use GDB to debug the program but symbolic information (names and types of variables, line numbers, etc) will be missing.

[97] http://en.wikipedia.org/wiki/Endianness

13.1.1.5 -o file

Write the output of the gcc command to *file*. For example,

```
$ gcc file.c -o file
$ ./file
```

This command will compile and link the source code file file.c and will name the executable *file* rather than a.out.

13.1.1.6 -S (capital)

Stops the compiler after the assembly code generation phase, e.g.,

```
$ gcc -S file.c
```

This command will convert the code in file.c into functionally equivalent assembly language in the file named file.s. It will not assemble, link, and produce an executable. This is a handy way to help yourself learn assembly language: write some C code, compile it to assembly code, and study the assembly code to see how it is implemented.

13.1.1.7 -c

Compile only, do not link.

```
$ gcc -c file.c
```

This command will compile file.c and produce the output file file.o. However, it will stop at that point and not attempt to link to produce an executable.

Normally, when you make a project containing multiple C source code files, you will compile all of the source code files individually using -c and compile the main source code file last without using -c. We will discuss this in more detail when we talk about makefiles later in this chapter. For example, suppose we have a project with four C source code files: file1.c, file2.c, file3.c, and file_main.c:

```
$ gcc -c file1.c                                             produces file1.o
$ gcc -c file2.c                                             produces file2.o
$ gcc -c file3.c                                             produces file3.o
$ gcc -o file file_main.c file1.o file2.o file3.o
$ ./file                                             produces executable file
```

13.1.1.8 -w

Inhibits all warnings. **Warnings** are messages from the compiler that something about the code it is compiling seems fishy but it is not syntactically incorrect. The canonical example would be,

```
if (a = b) {
    ...
}
```

This is perfectly legal C, but often it is a mistake because the programmer really intended to write a == b. Some compilers will issue a warning message when they compile this code to let the programmer know that what the compiler is seeing may not be what the programmer expected.

If you compile with -w these sorts of warning messages will not be displayed. In general, warning messages are a good thing and your code should compile successfully without any errors or warnings. If you do get warnings from the compiler you should fix them. For warnings which are issued by the compiler for code which is okay, then **pragmas**[98] or **__attributes__**[99] should be used to inform the compiler that what you wrote is what you intended.

13.1.1.9 -Wall

Turns on all warning checks. I **strongly advise** you use this option and check all of the warning messages generated by using it. This may be helpful in tracking down difficult-to-find bugs which are the result of syntactically correct code.

13.1.1.10 Optimization Options

-O0 (i.e., dash capital-oh zero) turns off all optimization. This is the default. -O1, -O2, and -O3 successively increase the amount of optimization that the compiler performs on the final machine language code. -Os optimizes for size over speed.

13.1.1.11 -Idir

(That is: dash capital-eye followed by dir). Add dir to the list of directories searched for included header files. By default the GCC system directories and the current directory will be searched.

13.1.1.12 -M and -MM

Useful for generating dependencies for makefiles. We will discuss this more when we discuss makefiles.

13.1.1.13 -std=standard

Treats the code as conforming to C/C++ standard[100] *standard* where the supported standards are,

[98] http://gcc.gnu.org/onlinedocs/cpp/Pragmas.html
[99] http://unixwiz.net/techtips/gnu-c-attributes.html
[100] http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Standards.html#Standards

-std=c90	Is equivalent to -ansi; 1990 ISO Standard C definition.
-std=c99	1999 ISO Standard C definition (GCC has partial support).
-std=c11	2011 ISO Standard C definition (limited incomplete support).
-std=gnu90	1990 ISO Standard C plus GNU extensions. Default for C code.
-std=gnu99	1999 ISO Standard C plus GNU extensions
-std=gnu11	2011 ISO Standard C plus GNU extensions
-std=c++98	1998 ISO Standard C++
-std=gnu++98	1998 ISO Standard C++ plus GNU extensions. Default for C++.
-std=c++11	2011 ISO Standard C++ (experimental).
-std=gnu++1	2011 ISO Standard C++ plus GNU extensions.

Nonstandard code elements will be treated as syntactically incorrect.

13.1.1.14 -llibrary

(That is: dash lowercase-ell followed by library). Search the library named library when linking. We will discuss libraries and this command line option a bit later.

13.1.1.15 -Ldir

Search the directory named dir for library files in addition to the standard directories and the current directory.

13.1.1.16 --help

Prints help information about the compiler.

13.2 Using G++

The **g++** command invokes the GCC C++ compiler. Most of the options for gcc are the same as for g++, e.g.,

```
$ g++ -Ddbg=YES -I./include -L./lib -o foo -O3 -w foo.c bar.cpp baz.cpp
$ ./foo
```

Explanation: (g++) compile using the g++ compiler; (-D) define the macro *dbg* to YES; (-I) search the directory *include* located under the current directory for header files; (-L) search the directory *lib* located under the current directory for library files; (-o) send the output executable to the file named *foo*; (-O3) optimize the heck out of the code. (-w) disable all warnings (foo.c, ...) and compile the files foo.c, bar.cpp, and baz.cpp.

Note that C++ programs commonly have an extension of .cc, .cpp, or occasionally .c++. C++ header files are usually either .h or .hpp or do not have a filename extension.

13.3 *nix Libraries

A good reference on libraries is available at [101].

A **library** contains compiled (binary) code and data that can be incorporated into a program. Generally, libraries are created by adding one or more object files to the library file. You can create your own library files, but more commonly in *nix development, you will be linking your code with preexisting code contained in various library files (which are usually installed in /lib and /usr/lib).

The advantages of libraries are that they contain code that has already been developed and presumably debugged so the functions in the libraries can be used over and over again in different programs. This increases programmer productivity which is a good thing.

In *nix systems there are three main types of library files: **static libraries**, **shared libraries**, and **dynamic libraries**. In current Linux distros, library files and executable files are stored in a **file format** named ELF[102].

13.3.1 Static Libraries

Static libraries are the simplest. A static library (file) contains just a collection of one or more object code files (much as a zip file contains one or more files). Static libraries are identified with a .a suffix on the filename, e.g., /usr/lib/libm.a is the math library in the Standard C library.

Static libraries are created with the **ar** command (ar standards for archive), e.g.,

```
$ pwd
/home/fredf/proj01
$ cd lib
$ pwd
/home/fredf/proj01/lib
$ gcc -c stack.c                        produces stack.o
$ gcc -c list.c                         produces list.o
$ gcc -c math.c                         produces math.o
$ ar -rcs mylib stack.o list.o math.o   creates archive named mylib
$ cd ../src
$ pwd
/home/fredf/proj01/src
$ gcc -o myexe my-program.c some-other-file.c -L../lib -lmylib
$ ./myexe
```

[101] http://www.faqs.org/docs/Linux-HOWTO/ Program-Library-HOWTO.html
[102] http://en.wikipedia.org/wiki/Executable_and_Linking_Format

The option r to the ar command means insert the named file(s) into the archive and replace any files already in the archive with the same filename. The c option tells ar to create the archive without complaining about it if it does not exist. The s option writes an index of the object files contained in the archive. There are additional command line options but these are sufficient to create a simple archive. For more help, try **man ar**.

The major disadvantage of static libraries is that each executable file that is produced by linking with the library will end up with identical copies of the object code linked from the library. For example, in libm.a there is a function named sin() that can be used to obtain the sine of an angle given in radians. If Bob compiles a program executable named foo which uses sin() by linking with libm.a then Bob's executable file will contain the machine language code for the sin() function. If Alice does the same thing, then her program executable will also contain the same machine language code.

Now consider when Bob and Alice both run their programs at the same time. Each program will be loaded into memory with two identical copies of the machine language code for sin() taking up room in memory. This is not a very efficient allocation of resources.

13.3.2 Shared Libraries

A shared library is designed to overcome the major problem with static libraries. When Bob links his code against the shared library, rather than including the machine language code for the sin() function in Bob's executable, a reference to the machine language code will be inserted. This reference does not take up much space. Alice's program will end up with the same sort of reference.

Now suppose Bob runs his program first. When it is loaded into memory, the OS will check to see if the machine language code for the sin() function is loaded. If it is not, it will be loaded. Then Bob's program code will be modified to point to the address of the sin() function so that when Bob's program makes a call to sin() it will jump to that address and execute the code of the shared sin() function. Alice's program will work similarly.

Shared libraries have a more complex naming scheme that static libraries. We won't get into it in this course, but if you are interested, I would recommend reading the FAQ referenced at the beginning of this section.

The gcc command can be used to compile files and create a shared library. Also, as with static libraries, we can link our code with existing shared libraries. In fact, this happens automatically behind the scenes when we compile C programs calling common library functions such as printf(); gcc handles this for us.

13.3.3 Dynamic Libraries

A dynamic library is a library which is not linked with the program during the build process. Rather, a running program can call the OS to open a dynamic library file for reading, and can load in the machine language code for a desired function at run-time. Then the running program can jump to the address where this code was loaded which effectively calls the function. That's all I am going to say about it.

13.3.4 Linking With a Static Library

The library file libm.a is the math library in the C Standard library. It contains various mathematical functions such as sin(), cos(), etc. If we write a program which calls the sin() function then we must link our code with the math library, e.g.,

```
$ cat myprogram.c
#include <math.h>

int main() {
  double f = sin(.434);
  return 0;
}

$ gcc myprogram.c -o myprogram -lm
```

Note that gcc already looks in /lib for library files by default so we do not need to use -L/lib on the command line to tell gcc to look there. Note also that the syntax -lm means link with the library named libm.a. In general, the syntax -l*foo* means link with the static library named lib*foo*.a (gcc will look in /lib and /usr/lib). Finally, note that because the linking stage comes after the compilation stage in the build process, the library file must be listed after the C source code file and output file. It is an error to do this,

```
$ gcc -lm myprogram.c -o myprogram
```

Most of the standard *nix libraries are found in the /lib and /usr/lib directories, e.g., on General, /lib/libc.so.6 (which is a symbolic link to /lib/libc-2.5.so) is the shared version of the GNU implementation[103] of the C Standard library—which is linked automatically so we do not need to include -lc on the command line of the gcc command. The 2.5 part of the filename is the version number. By the way, this is a pretty outdated version from 2006, with the most recent release being 2.17 in December 2012.

Many Standard C library header files are in /usr/include, e.g., /usr/include/stdio.h. Use #include <> for system libraries. The compiler will know to look in /usr/ include and its subdirectories for the header file.

[103] https://en.wikipedia.org/wiki/Glibc

On the other hand, using #include "" will tell the compiler to look in the current directory and any directories specified with the -I command line option for the header file. In general, <> is used for system header files, and "" for your own header files.

13.4 Makefiles

Most software projects of a reasonable size consist of multiple source code files, for very large projects perhaps thousands. Consider these numbers,

Linux Kernel 3.6	15.9 MLOC (million lines of code)
Windows Server 2003	50 MLOC
Debian 5.0	324 MLOC

The project may also include libraries (prebuilt as well as built) and many other types of resource files. One of the problems with large software projects containing hundreds or thousands of files is that we don't want to recompile every source code file every time we do a build. Therefore, some automated way of recompiling only those source code files that need to be recompiled is highly desirable.

13.4.1 Building a Project the Hard Way

Now the hard way to build a multifile program would be,

```
$ gcc -c file1.c           wait
$ gcc -c file2.c           wait
$ gcc -c file3.c           wait
...
$ gcc -c file100.c         wait
$ gcc file1.o file2.o ... file100.o -I./include -o myprog -L./lib -lmylib
$ ./myprog
```

Oops, found a bug. Edit file4.c and correct the bug. Start typing again. Sigh.

13.4.2 Building a Project with a Script File

A slightly better approach would be to put all of these commands in a shell script:

```
#!/bin/bash
#File: b.sh
gcc -c file1.c
gcc -c file2.c
gcc -c file3.c
...
gcc -c file100.c
gcc file1.o file2.o ... file100.o -I./include -o myprog -L./lib -lmylib

$ alias b=./b.sh
$ chmod 755 b.sh
$ b
go get some coffee
```

You would know the build is complete when the $ prompt comes back. Note that you could send the build to the background,

```
$ b&
$ tetris     do something productive now
```

Problems with this approach? Suppose our program grows to contain 1000 files. Every time a new source code file is added to the project, someone has to edit the b.sh script to add the new file. Or, every time a source code file is removed from the project, someone has to edit the b.sh script to delete the file. Or, every time we rename a source code file... I hope you get the idea.

Furthermore, there is nothing in the script file that determines if a source code file really has to be recompiled. Note that the only time a source code file must be recompiled is when it's text is modified (or the text of a header file the .c file includes changes). Recompiling 100 files over and over every time we perform a build, when only one or two of those files has changed is simply too inefficient (unless you're on a contract and getting paid by the hour). Surely there has to be a better way.

13.4.3 Introduction to Make

Make is a common *nix utility that was designed to automate the build process. According to Wikipedia[104], the first version was created at Bell Labs in 1977. Make has evolved over the years and there are many variants of it (e.g., Apache ANT and Maven for Java programmers). GNU Make[105] is the GNU project's version of make (online documentation[106]).

Make is an extremely complicated program and would require us to spend much more time than we have to fully discuss it. Therefore, we will focus on a very simplified make process to give you a feel for how it is used. I highly recommend you read the online documentation for further understanding.

Make processes one or more **makefiles** which contain rules (or instructions) for how to build the program. The syntax of these makefiles is, in usual *nix fashion, weird if you don't understand it[107].

[104] http://en.wikipedia.org/wiki/Make_(software)
[105] http://www.gnu.org/software/make/
[106] http://www.gnu.org/software/make/manual/make.html
[107] It's weird even if you do understand it.

We will start with a simple makefile and build upon it to make (ha ha) a complete, final makefile. Suppose my program is comprised of two C source code files main.c and mutils.c. Here is an initial, reasonably simple makefile,

```
#!/bin/bash
# File: Makefile

myprog: main.o mutils.o
	gcc main.o mutils.o -o myprog

main.o: main.c
	gcc -c main.c

mutils.o: mutils.c
	gcc -c mutils.c
```

Note that the lines starting with gcc are indented with a **hard tab**, i.e., not spaces. The words myprog, main.o, and mutils.o are called **targets**. They specify what make is to build. For the target myprog, main.o and mutils.o are dependencies (or prerequisites[108]). This says in English that the target myprog depends on main.o and mutils.o. If either one of main.o or mutils.o changes, then myprog is "out of date" and must be rebuilt.

How does make know when a file has changed? It looks at the timestamp, i.e., the date and time the file was last modified. If main.o was last modified (written) at 10/01/2007 11:15:23pm and main.c was last modified at 10/01/2007 11:18:21 then make assumes main.c has changed since the last time it was rebuilt (because the timestamp on main.c is more recent).

The target myprog is the default target because it appears first in the file. If we do not specify a target for make, it will attempt to build the default target.

```
$ make Makefile
gcc -c main.c
gcc -c mutils.c
gcc main.o mutils.o -o myprog
$ make makefile
make: 'myprog' is up to date.
```

Note that typing **make** is equivalent to typing **make Makefile**, i.e., make will always try to read commands from Makefile; if that file cannot be found it will look for **makefile**.

[108] **Dependencies**: One source code file, A, may depend on another source code file, B. Changes to B may cause A to change. In GNU terminology, dependencies are called **prerequisites**.

13.4.4 Make Command Line Options

Make's default behavior can be modified with command line options. Some of the more useful ones are described here.

-f *file*	Read make commands from a file named *file* rather than from Makefile.
-d	Display debugging info. Makefiles can be extremely complex, so this option is useful when your makefile is not working correctly.
-n	Do everything but actually execute each of the make commands. Useful to see what make is going to do before you run it.
-s	Silent mode, no output.
--version	Print the version of make. Sometimes when you are building a program on *nix from source the instructions will tell you that you need a specific specific version of make.
-B	Make everything unconditionally without regard to timestamps.
-i	If a command generates an error keep going.
-k	If a target cannot be made keep going.
--help	Display help information.

13.4.5 Automatic Variables and Implicit Rules

Suppose we have a project with 1000 C source code files:

```
-----------------------------------------------------------------------------
#!/bin/bash
# File: Makefile

myprog: file1.o file2.o file3.o ... file1000.o
	gcc file1.o file2.o ... file1000.o -l<libs> -o myprog

file1.o: file1.c
	gcc -c file1.c

file2.o: file2.c
	gcc -c file2.c

...

file1000.o: file1000.c
	gcc -c file1000.c
-----------------------------------------------------------------------------
```

There has to be a better way to specify that every .c file should be recompiled to a .o file when the .c file is out of date. There is, using **implicit** or **suffix rules**. Try,

```
$ make -p > make-defaults.txt
$ less make-defaults.txt
CC = cc
OUTPUT_OPTION = -o $@
...
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) -c
...
%.o: %.c
	$(COMPILE.c) $(OUTPUT_OPTION) $<
```

The rule starting with %.o is a suffix rule. The characters %.o specify that this is a **pattern rule**. The characters $@ refers to the name of the target .o file. The characters $< refers to the name of the prerequisite .c file. Collectively, $@ and $< are called **automatic variables** in make terminology (there are other automatic variables as well, you should read the online manual about $? and $^ which are also very useful).

Therefore, since make already has a predefined (or built-in) rule for building a .o file from a .c file, we can modify our makefile,

```
----------------------------------------------------------------------------
#!/bin/bash
# File: Makefile

myprog: file1.o file2.o file3.o ... file1000.o
	gcc file1.o file2.o ... file1000.o -l<libs> -o myprog
----------------------------------------------------------------------------
```

or,

```
----------------------------------------------------------------------------
#!/bin/bash
# File: makefile

myprog: main.o mutils.o
	gcc main.o mutils.o -o myprog
----------------------------------------------------------------------------
```

Now make,

```
$ make
cc    -c -o main.o main.c
cc    -c -o mutils.o mutils.c
gcc main.o mutils.o -o myprog
```

13.4.6 Compiler Command Line Options

We see a macro variable $(CFLAGS) was defined and used in the implicit rule for building a .c file. We can use this variable to specify our own compiler command line options,

```
----------------------------------------------------------------------------
#!/bin/bash
# File: makefile

CFLAGS=-g -O2

myprog: main.o mutils.o
	gcc main.o mutils.o -o myprog
----------------------------------------------------------------------------
```

Then make,

```
$ !m                !m repeats the last command that began with the letter m
make
cc -O2 -g   -c -o main.o main.c
cc -O2 -g   -c -o mutils.o mutils.c
```

Suffix rules for C++,

```
CXX = g++
COMPILE.cpp = $(COMPILE.cc)
COMPILE.cc = $(CXX) $(CXXFLAGS) $(CPPFLAGS) -c
```

Thus, we can modify $(CPPFLAGS) to specify g++ compiler command line options.

13.4.7 Handling Header Files

Notice that mutils.h is not listed in the makefile. Header files are not compiled separately, but .c files do depend on them, e.g., if we change mutils.h then we have to recompile main.c because main.c includes mutils.h (in other words, changing the text of mutils.h is equivalent to changing the text of any .c file which includes mutils.h).

Our makefile does not work correctly then because if I change mutils.h and perform a make, then make tells me that myprog is up to date. So how do we specify that changes to .h files should cause .c files that depend on them to be recompiled? We modify the pre-requisites for the .o file to include the .h file(s) as well as the .c file(s),

```
----------------------------------------------------------------------------
#!/bin/bash
# File: makefile

CFLAGS=-g -O2

myprog: main.o mutils.o
	gcc main.o mutils.o -o myprog

main.o: main.c mutils.h
mutils.o: mutils.c mutils.h
----------------------------------------------------------------------------
```

The line starting main.o basically says that main.o depends on main.c and mutils.h. A change to one or both of those files would cause main.c to be recompiled to produce an up-to-date main.o.

But, wouldn't having to remember and list which .h files a .o file depends on get tedious for a large project? (Trust me, it would.) What if I add one or more new #include header files to a .c source code file, or remove one or more header files? Sounds like a headache. Never fear, gcc to the rescue. The -M option to gcc will generate a dependency rule listing all of the .h files that a .o file depends on,

```
$ gcc -M main.c
main.o: main.c /usr/include/stdio.h /usr/include/_ansi.h \
  /usr/include/newlib.h /usr/include/sys/config.h \
  /usr/include/machine/ieeefp.h /usr/include/cygwin/config.h \
  /usr/lib/gcc/i686-pc-cygwin/3.4.4/include/stddef.h \
  /usr/lib/gcc/i686-pc-cygwin/3.4.4/include/stdarg.h \
  /usr/include/sys/reent.h /usr/include/_ansi.h
  /usr/include/sys/_types.h \
  /usr/include/sys/lock.h /usr/include/sys/types.h \
  /usr/include/machine/_types.h /usr/include/machine/types.h \
  /usr/include/sys/features.h /usr/include/cygwin/types.h \
  /usr/include/sys/sysmacros.h /usr/include/stdint.h \
  /usr/include/endian.h /usr/include/sys/stdio.h
  /usr/include/sys/cdefs.h \
  mutils.h
```

Note that system header files not do change that often (usually only when a newer version of the compiler is installed). Hence, they do not need to be listed in the makefile. Use the -MM command line option to gcc to omit system header files,

```
$ gcc -MM main.c
main.o: main.c mutils.h
```

Our new modified makefile is,

```
---------------------------------------------------------------------------
#!/bin/bash
# File: makefile

CFLAGS  = -g -O2

SOURCES = main.c   \
          mutils.c

myprog: main.o mutils.o
	gcc main.o mutils.o -o myprog

%.d: %.c
	rm -f $@; gcc -MM $< > $@
```

```
include $(SOURCES:.c=.d)
-----------------------------------------------------------------------------
```

The %.d: %.c rule says that a file with a .d extension depends on a file with a .c extension, i.e., foo.d depends on foo.c. The automatic variable $@ refers to the target dependency (the .d file) and $< to the prerequisite dependency (the .c file). When this rule is applied to a .c file, the corresponding .d file is deleted (because of the "rm -f $@" command. Then "gcc -MM" is performed on the .c file and the output is redirected to the .d file (this is the "> $@" part). For main.c and mutils.c this would create the files main.d and mutils.d,

```
$ cat main.d
main.o: main.c mutils.h
$ cat mutils.d
mutils.o mutils.d : mutils.c
```

The SOURCES= line defines a macro variable which contains the names of all of the .c files in the project. The include line states that in the $(SOURCES) macro variable, every filename that ends with .c should be changed to .d. Then everyone of those .d files are included in the makefile. This is equivalent to writing the line,

```
include main.d mutils.d
```

but is more flexible, because now if we add a new C source code file to the project, all we have to do is add the name of the source code file to the SOURCES= line. Now that we know that trick, we can make another change,

```
-----------------------------------------------------------------------------
#!/bin/bash
# File: makefile

# The name of the executable file (called a binary).
BINARY = myprog

# Options for the C compiler.
CFLAGS = -ansi -c -g -O0 -Wall

# a list of the .c files in the project.
SOURCES = main.c    \
          mutils.c

# A list of the .o files in the project. Replaces all occurrences
# of .c in the SOURCES macro with .o, i.e., OBJECTS will be
# assigned: main.o mutils.o.
OBJECTS = $(SOURCES:.c=.o)

# The binary depends on the object files.
# Link the object files to produce the binary
$(BINARY): $(OBJECTS)
	gcc $(OBJECTS) -o $(BINARY)
```

```
# Even though there is an implicit rule which states that the .o
# files depend on the .c files, I like to be explicit about it.
# Compile a .c file to produce a .o file.
%.o: %.c
	gcc $(CFLAGS) $< -o $@

# The .d files depend on the .c files.
# Create a .d file by running gcc –MM.
%.d: %.c
	rm -f $@; gcc -MM $< > $@

# include the contents of all of the .d files.
include $(SOURCES:.c=.d)
----------------------------------------------------------------------------
```

Now, that's a pretty nice looking makefile isn't it?

13.4.8 Phony Targets

A phony target is a target that is not built, but rather is placed in the makefile to do something useful, e.g.,

```
----------------------------------------------------------------------------
#!/bin/bash
# File: makefile

# The name of the executable file (called a binary).
BINARY = myprog

# Options for the C compiler.
CFLAGS = -ansi -c -g -O0 -Wall

# a list of the .c files in the project.
SOURCES = main.c    \
          mutils.c

# A list of the .o files in the project. Replaces all occurrences
# of .c in the SOURCES macro with .o, i.e., OBJECTS will be
# assigned: main.o mutils.o.
OBJECTS = $(SOURCES:.c=.o)

# The binary depends on the object files
# Link the object files to produce the binary
$(BINARY): $(OBJECTS)
	gcc $(OBJECTS) -o $(BINARY)

# Even though there is an implicit rule which states that the .o
# files depend on the. c files, I like to be explicit about it.
# Compile a .c file to produce a .o file.
%.o: %.c
	gcc $(CFLAGS) $< -o $@
```

```
# The .d files depend on the .c files.
# Create a .d file by running gcc –MM.
%.d: %.c
	rm -f $@; gcc -MM $< > $@

# include the contents of all of the .d files.
include $(SOURCES:.c=.d)

# "clean" is a phony target
.PHONY: clean
clean:
	rm -f $(OBJECTS)  # delete all of the .o files.
	rm -f *.d         # delete all of the .d files.
	rm -f myprog      # delete the binary.
-----------------------------------------------------------------------------
```

Now run,

```
$ make clean
rm -f main.o mutils.o
rm -f main.d mutils.d
rm -f myprog
```

After cleaning, every source code file will be recompiled and the resulting object files will be linked with the library files to build the target binary. You don't need to do a clean every time you want to build your project, because that would defeat the purpose of using a makefile in the first place.

Open-source programs that you might download from the Internet and build on your GNU/Linux box often come with makefiles provided. A common way to build the software and install it is to perform the commands,

```
$ ./configure
$ make
$ make install
```

where *configure* is a shell script that actually creates the *Makefile* file and install is a phony target which causes make to copy/move the executable and other installation files to their final destinations (e.g., to /usr/local/bin).

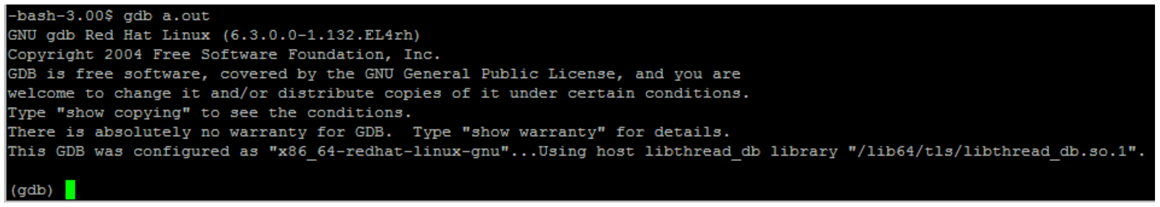
As I mentioned earlier, make is a very complicated program. We have only scratched the surface of what it can do here, but I urge you to read the online tutorials and manuals and learn more about it. Someday you will make your employer happy that you did.

14 Debugging

14.1 Text-Based Debugging with GDB

14.1.1 Launching and Quitting gdb

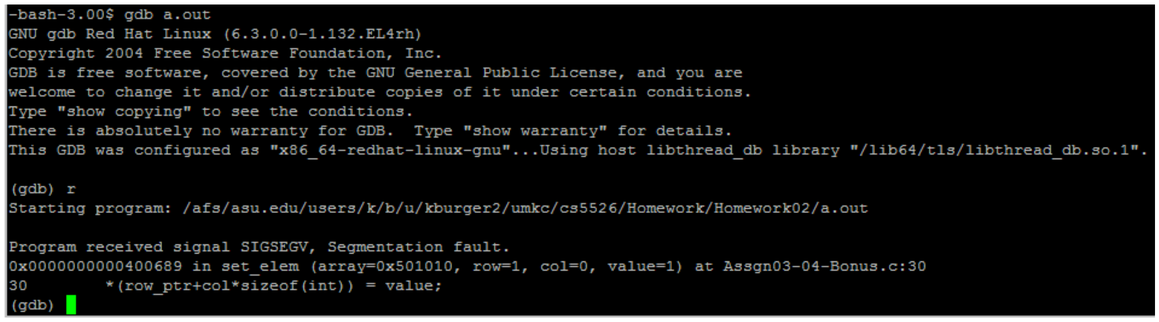
To start **gdb** and debug the executable a.out, at the command prompt type **gdb a.out**. Note that in order to debug a.out you must have built a.out with the gcc -g option; it also best if you do not compile with optimization, i.e., use -O0. You will see a screen similar to the following. The gdb prompt is (gdb) which means it is waiting for input.


```
-bash-3.00$ gdb a.out
GNU gdb Red Hat Linux (6.3.0.0-1.132.EL4rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...Using host libthread_db library "/lib64/tls/libthread_db.so.1".

(gdb)
```

To leave gdb, use the command **quit** (abbreviated **q**) or hit Ctrl+D.

14.1.2 Running Your Program

To run your program non-stop, type run or just r. Your program will either run to completion or crash. If it runs to completion and produces the correct output, then what are you using the debugger for? On the other hand, mine crashed with a segmentation fault,


```
-bash-3.00$ gdb a.out
GNU gdb Red Hat Linux (6.3.0.0-1.132.EL4rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...Using host libthread_db library "/lib64/tls/libthread_db.so.1".

(gdb) r
Starting program: /afs/asu.edu/users/k/b/u/kburger2/umkc/cs5526/Homework/Homework02/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400689 in set_elem (array=0x501010, row=1, col=0, value=1) at Assgn03-04-Bonus.c:30
30          *(row_ptr+col*sizeof(int)) = value;
(gdb)
```

The normal way to generate a segmentation fault is to try to read from or write to memory that does not belong to you. Seems simple enough to avoid that, but when you're using pointers, crazy things can happen. The dump tells me which program I was running, ".../Homework02/a.out". I was executing code in a function named set_elem() and you can see the input parameters to the function were array which was assigned the value 0x501010, row which was 1, col which was 0, and value which was 1. This code is on line 30 of the source code file Assgn03-04-Bonus.c. The very last line in the display (starting with 30) is the source code statement on line 30 which caused the exception.

Now that I know the line number of the statement which caused the exception, I can go back to my source code and examine it to see if I can find the bug. Note that just because that is the line we were executing when the segmentation fault occurred does not mean that that is the line where the actually bug lies. C is an extremely powerful and exceedingly dangerous language. It is entirely possible that some other statement somewhere accidentally modified this block of memory I am now accessing and trashed it. Hence, the bug may not be on this line of code but somewhere else. These bugs are hard as heck to find; I once spent two 8-hr work days tracking one down.

14.1.3 Listing Source Code

Gdb is a source code debugger, which means you can display your source code as you are debugging and also view the assembly language code. You can step through your program line by line executing instructions and viewing the contents of registers and memory. This is extremely powerful when attempting to locate bugs. To print source code lines, use one of the variants of the list command (abbreviated l; that is, lowercase ell):

`list linenum`
Prints lines centered around line number *linenum*. The number of lines is controlled by **set listsize *count***. The default value for count is 10. To see the current value of *count*, use **show listsize**.

`list` ***`function`***
Prints lines centered around the beginning of *function*. If *count* is ten, five lines before the function will be printed, and the first five lines of the function will be printed.

`list`
Prints more lines following the lines printed by the previous list command, i.e., scroll down.

`list -`
Prints more lines preceding the lines printed by the previous list command, i.e., scroll up.

`list` ***`first,last`***
Prints lines from *first* to *last*, e.g., list 1,100.

`list` ***`file`*****`:`*****`number`***
Prints line *number* from source file *file*.

`list` ***`file`*****`:`*****`function`***
Prints lines centered around function *function* from source file *file*.

14.1.4 Examining Assembly Language Code

Your C or C++ source code is compiled into assembly language. You can examine this assembly language code in gdb. To see the assembly language code you use the **disassemble** (or **disas**) command.

`disas`
Disassembles code centered around the current PC.

`disas` ***`addr1`***
Displays disassembled code starting at memory address *addr1*.

`disas` ***`addr1,addr2`***
Displays disassembled code from memory address *addr1* to memory address *addr2*.

To find the memory address of the beginning of a line of source code, use **info line** ***line***, e.g., **info line 30**. Gdb will respond with the starting memory address of the assembly language code for the statement and the ending memory address (a single HLL statement may compile down to multiple assembly language instructions). The command **x/i** will begin displaying instructions or you can use **disas** ***addr1*****,** ***addr2*** to display the entire range. Pressing x repeatedly after x/i will display one instruction at a time.

14.1.5 Examining Data and Variables

When your program is running, or when it crashes, you may want to see what values were in certain memory locations at the time of the crash. This is done with the **print** or p command.

`p` ***`file::var`***
Will print the value stored in global variable *var* defined in *file*.

`p` ***`var`***
Will print the value of the variable *var* if it is in scope. If it is not in scope, then strange things may happen.

14.1.6 Examining Memory

Sometimes you want to look at memory directly as a bunch of bytes than as variables. For example, I might want to know what values are stored at memory locations 0x501010 through 0x501030. The command to do this is **x** (for examine).

`x` ***`addr`***
Display the value at memory location *addr*.

`x/5 addr`
Display five words (a word is 4-bytes) starting at *addr*. In general use x/*n* to display *n* words.

`x/3d addr`
Display three words starting at *addr*. The d means display the values as signed decimal numbers. The default is to display in hex (you would use x for that). Use u for unsigned integers, o for octal, t for binary, c for character, f for float, and s for string. Use i to interpret the words as machine language instructions.

`x/4b 0x501010`
Display four bytes starting at 0x501010 in hex format.

`x/4t 0x501010`
Display four bytes starting at 0x501010 in binary format.

`x/fg 0x501010`
Displays a double (8-bytes) at 0x501010. g is for giant (a double to gdb is a giant float).

`x/5fg 0x501010`
Displays five doubles start at 0x501010.

Continuing to hit x will display successive memory locations.

14.1.7 Examining Array Variables

If *a* is an array then **p a** will print the address in memory of the beginning of the array. Remember that the name of an array variable is equivalent to the address in memory of the beginning of the array. To display the entire array use the command **p *array@len*** where *len* is the number of words in *array*, e.g., if a is an array of 10 integers, then use p*a@10 to display the 10 integers in a.

14.1.8 Setting Watches or Automatic Displays

If you want to view the contents of a variable or memory location each time your program stops (e.g., if you are single-stepping through the program), then add the variable or address to the automatic display list: display var or display addr. Each display in the list is numbered starting at 1.

`display/5t 0x501010`
Displays five words starting at 0x501010 in binary format every time the program stops.

`display/i $pc`
Displays the machine language instruction pointed to by the PC register. This is the instruction to be executed next.

`delete display 1`
Delete display number 1.

`disable display 2`
Disable display number 2 but don't delete it.

`enable display 2`
Enable display number 2.

`info display`
Display the display list.

14.1.9 Changing the Values of Variables

Use **set var *varname*=*value*** if the variable is in scope. If the variable is not in scope, you cannot do this.

14.1.10 Examining Registers

To display the register names and their values, use **info registers**. To print the program counter register, use **p $pc**. The command **x/i $pc** will print the instruction pointed to by $pc. The stack register is referred to by $sp. To set a register use **set *regname*=*value***, e.g., set rcx=0x100.

14.1.11 Stepping

Normally we use a debugger because our program has bugs in it that we need to locate and fix. In these cases, you often want to step through the program executing one line at a time, examining the values of variables and memory, until you find the point where something bad is happening. When you load your program in gdb, if you hit r or run it will run non-stop until it finishes or crashes.

To step through the program one line at a time, use the **step** or **s** command—but only after typing **start** to start program execution. Step will execute one source code line and then stop (remember that each source code line is compiled into one or more assembly language instructions). The actual bug will be found during execution of one of these assembly language instructions.

`step` ***`n`***
Abbreviated s *n*. Execute *n* source code instructions and then stop.

`next`
Abbreviated **n**. Executes the next source code instruction, but jumps over function calls. Normally step will step into function calls. Next does not—ever. This is useful if you know the function you are calling does not have any bugs in it, and you just want the function to execute so you can get on with your life.

`next` ***n***
Or n *n*. Like **step *n*** but doesn't execute into functions.

`finish`
Run until the function you are in returns, then stop and display the return value.

`until`
Typically used after you begin stepping through a loop. It will run nonstop until the end of the loop. Typing **until** will continue execution until the loop terminates, whereas **next** would execute the loop again by going back to the beginning.

`stepi`
Abbreviated **si**. Execute one machine language instruction. Using display/i $pc in conjunction with si is an easy way to display each machine language instruction before it is executed, and then execute the instruction and stop. That is, you would type display/i $pc once, and then you could do si, si, ... to single-step through the program executing one machine language instruction at a time. Note that stepi will step into functions.

`si` ***n***
Or, stepi *n*. Execute *n* assembly language statements. Will step into functions.

`nexti`
Abbreviated **ni**. Execute one assembly language statement and execute over function calls (i.e., don't step into it).

`ni` ***n***
Execute *n* assembly language statements executing over function calls.

Once you enter one of these commands, for example, s or si, continuing to hit ENTER will execute the same command again.

14.1.12 Breakpoints

A **breakpoint** is a marker which tells the debugger to stop at that marker. For example, you can set a breakpoint on a certain source code line, and then run your program with r. When that line is reached during execution, the debugger will stop. You can then examine the state of memory and variables to try and find the bug. Then you could single step from that point on, or you could continue running until the next breakpoint is reached, e.g., if the line containing the breakpoint is reached again during execution, the debugger will break again.

Breakpoints can be set on source code lines or machine language instructions. Often it is helpful to identify the function where the bug is, then to find the line within the function, and then to finally identify the machine language instruction containing the bug if need be. Breakpoints are set with the **break** or **b** command. Breakpoints are disabled with the **disable** command. Break points are enabled with the **enable** command. Breakpoints are deleted with the **delete** (**d**) command. Information on breakpoints is obtained with the **info** (**i**) command.

`continue`
Abbreviated **c**. Continue execution after a breakpoint is triggered until the next breakpoint.

`b` ***`function`***
Set a breakpoint at the entry point of *function*.

`b` ***`linenum`***
Set a breakpoint at source code line *linenum* in the current source code file.

`b` ***`file`*****`:`*****`linenum`***
Set a breakpoint at *linenum* in source code file *file*.

`b` ***`file`*****`:`*****`function`***
Set a breakpoint at *function* in source code file *file*.

`b *`***`addr`***
Set a breakpoint at memory address *addr*.

`tbreak` ***`args`***
The argument *args* are the same as for the break commands discussed above, but in this case the breakpoint is temporary, i.e., the debugger will break once but then the breakpoint is deleted.

`rbreak` ***`rexpr`***
Argument *rexpr* is a regular expression. This command will set a breakpoint on all functions matching the *rexpr* pattern. Since the regular expression . matches every function, **rbreak .** will set a breakpoint on every function in your program. Then you can hit c to continue execution until the next function gets called.

`info break`
Abbreviated **i b**. Displays a table of breakpoints.

`disable` ***`bps`***
Abbreviated **dis**. Disable all breakpoints in the range, e.g., disable 2 would disable breakpoint number 2 (see info break). disable 2 3 5 would disable breakpoints 2, 3, and 5. Disable [2-5] would disable breakpoints 2, 3, 4, and 5. If *bps* is not given then all breakpoints are disabled.

`enable` ***`bps`***
Abbreviated **en**. Arguments are similar to **disable**. Enables breakpoints.

`en` ***`bps`*** **`once`**
Enables the breakpoints once, then once each breakpoint is hit, disables the breakpoint again.

`en` ***`bps`*** **`delete`**
Enables the breakpoints once, then once each breakpoint is hit, deletes the breakpoint.

`delete` ***`bps`***
Abbreviated **d**. Arguments are similar to **disable** and **enable**. Deletes the breakpoints.

`condition` ***`bnum expr`***
Abbreviated **cond**. Sets a condition on breakpoint number *bnum*. The condition might specify that the value of a variable local to a function be equal to a certain value, e.g., if we do **b foo** to set a breakpoint (numbered 2) on function foo(), and foo() has a local variable named x, then we can set a condition on the breakpoint for foo(): **cond 2 x==4** says the breakpoint 2 is triggered only if x is equal to 4.

`cond` ***`bnum`***
Remove the condition on breakpoint number *bnum*.

`ignore` ***`bnum count`***
An especially useful thing is to be able to break on a breakpoint only after it has been triggered *n* times. For example, set a breakpoint on function foo(), but specify that the breakpoint is to be triggered only on the fifth call to foo(). This is done by **ignore *bnum count***. For example, **ig 2 5** if the breakpoint on function foo() is breakpoint number 2.

14.1.13 Watchpoints

A **watchpoint** is similar to a breakpoint. You instruct the debugger to stop execution if the value of variable changes (we could call these data breakpoints).

`watch` ***`foo`***
Watches variable *foo* and breaks if it is written to.

`rwatch` ***`foo`***
Breaks if *foo* is read.

`awatch` ***`foo`***
Breaks if *foo* is read or written (**a** means access).

`info wat`
Displays a table of watchpoints.

The expression to watch does not have to be a simple variable. We could tell gdb that we want it to watch a certain region of memory and break when that region is read from or written to.

`watch 0x501010`
Watch memory location 0x501010 for a write.

`watch 0x500-0x600`
Watch memory range 0x500 through 0x600 for writes.

14.1.14 Miscellaneous Gdb Commands

The command **shell** (**sh**) can be used to execute a shell command without leaving gdb, e.g., try **shell ls**. The command **shell make** will rebuild the project in the current directory. If the make succeeds, you can reload the executable using the command **file *binary***.

To log the output of a debugging session, use the **set logging on** command. The default log file is gdb.txt. Use **set logging file *file*** to change the logging file from gdb.txt to *file*. By default gdb appends to the logging file; use **set logging overwrite** on to switch to overwrite mode. Use **set logging overwrite off** to switch back to append mode.

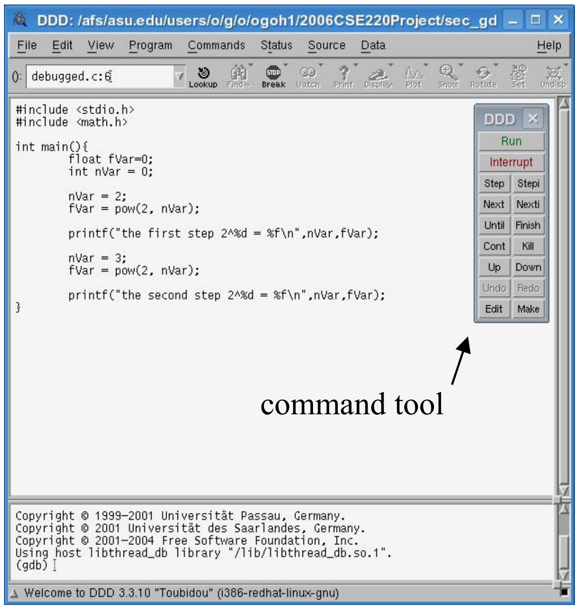
Gdb will repeat the previous command if you just press the ENTER key. For example, if you are single-stepping through the program using si, just type si once, and from then on you can just keep hitting the ENTER key to continue single stepping.

By default, gdb assumes numbers you enter are in decimal. Use 0x to specify hex. To change the input radix to hex permanently, use **set input-radix 16**. From now until the end of your debugging session, gdb will assume numbers you enter are in hex without you having to type 0x.

Getting help is accomplished by using the **help** command or Google.

14.2 GUI Debugging with DDD

Gdb is a powerful debugger, but it's interface is relatively primitive compared to today's GUI's. It is possible to write a graphical front-end for gdb which interacts with gdb on the back-end by submitting text commands in response to the user's actions with the mouse and keyboard while interacting with the GUI. One such program is DDD, the Data Display Debugger[109].


DDD: /afs/asu.edu/users/o/g/o/ogoh1/2006CSE220Project/sec_gd
File Edit View Program Commands Status Source Data Help
(): debugged.c:6
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make
command tool

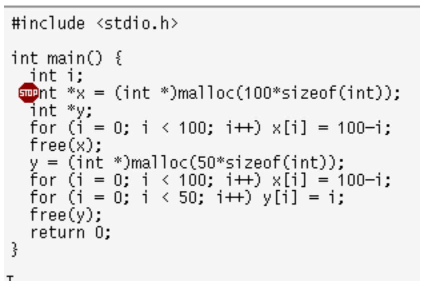
```
#include <stdio.h>
#include <math.h>

int main(){
        float fVar=0;
        int nVar = 0;

        nVar = 2;
        fVar = pow(2, nVar);

        printf("the first step 2^%d = %f\n",nVar,fVar);

        nVar = 3;
        fVar = pow(2, nVar);

        printf("the second step 2^%d = %f\n",nVar,fVar);
}
```

```
Copyright © 1999–2001 Universität Passau, Germany.
Copyright © 2001 Universität des Saarlandes, Germany.
Copyright © 2001–2004 Free Software Foundation, Inc.
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb)
```

Welcome to DDD 3.3.10 "Toubidou" (i386-redhat-linux-gnu)

Since DDD is a graphical program, you cannot run it while logged in to General in text mode. You *can* run it on the GNU/Linux systems in the BYENG 214 lab, or, if you are running Cygwin/X on your Windows machine, then you can open a ssh connection to General, and run it from there; Last time I checked, ASU has DDD version 3.3.9 installed. The latest version on the DDD website is 3.3.12. (Of course if you're running Cygwin, there is also a port of DDD for Cygwin that you would probably be better off installing; it is the latest version 3.3.12). If you installed Linux Mint as discussed in Chapter 2, ddd can be found in the Development (universe) category in Synaptic.

In DDD you can interact with gdb through the console window at the bottom of the DDD window. The **command tool** is turned on by selecting View | Command Tool.



```
#include <stdio.h>

int main() {
   int i;
   int *x = (int *)malloc(100*sizeof(int));
   int *y;
   for (i = 0; i < 100; i++) x[i] = 100-i;
   free(x);
   y = (int *)malloc(50*sizeof(int));
   for (i = 0; i < 100; i++) x[i] = 100-i;
   for (i = 0; i < 50; i++) y[i] = i;
   free(y);
   return 0;
}
```

To set a breakpoint, move the mouse pointer to the left margin of the source line where you want to set a breakpoint, and right-click. Select Set Breakpoint from the popup menu. A stop sign icon will be displayed on the line.

To start running the program, click Program | Run from the main menu, or click Run on the command tool. This will run the program non-stop. To start running and stop on the first line of main(), type start in the console window, or set a breakpoint on the first line of the code where you want to stop.

To inspect the value of a variable, right click on it and select Print. It's value will be printed in the console window. To add the variable to the display, right-click on it, and

109 http://www.gnu.org/software/ddd

select Display. If you don't see the display pane, click View | Data Window on the main menu. If you hold the mouse pointer over a variable, after .5 seconds a popup box will appear with the value of the variable.

Clicking View | Machine Code Window will show you the disassembly listing. You can set breakpoints here as in the source code listing window.

To see the contents of the registers, click Status | Registers.

To display line numbers in the source code listing window, click Source | Display Line Numbers.

If you click Source | Edit Source, vi will be loaded with your source code. You can make changes to the code, save it in vi, exit vi, and the code will be reloaded into DDD.

If you want to remake the project without exiting DDD, you can type **make** in the gdb console window to make your project (assuming you have a makefile). The issue the **file *binary*** command to reload the remade binary file.

To view information about the variables of the current stack frame, click Data | Status Displays. Then click Local Variables of Current Stack Frame. The local variables will be displayed in the data window and will be updated as the program is being executed step-by-step.

To examine memory, click Data | Memory. In the box that pops up you can select the format for the display (e.g., decimal, binary, hex, char), the number of words to display, and the starting memory address. Clicking print will dump the memory contents to the gdb console window. Clicking display will display the memory contents in the data window.

Well, that's it for this basic introduction to gdb and DDD. For more info, consult the online documentation.