Unix Users's Guide

May 13, 2020

COLLABORATORS			
	TITLE : Unix Users's Guide		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Jason W. Bacon	May 13, 2020	

Contents

1 Using Unix **1**

1.1 Keep It Simple, Stupid . . . 1

1.2 What is Unix? . . . 2

1.2.1 Aw, man... I Have to Learn Another System? . . . 2

1.2.2 Operating System or Religion? . . . 4

1.2.3 The Unix Standard API . . . 7

1.2.4 Shake Out the Bugs . . . 8

1.2.5 The Unix Standard UI . . . 9

1.2.6 Freedom of Choice . . . 9

1.2.7 Fast, Stable and Secure . . . 9

1.2.8 Sharing Resources . . . 10

1.3 Self-test . . . 10

1.4 Unix User Interfaces . . . 10

1.4.1 Graphical User Interfaces (GUIs) . . . 10

1.4.2 X11 on Mac OS X . . . 13

1.4.3 Command Line Interfaces (CLIs): Unix Shells . . . 13

1.4.4 Terminals . . . 15

1.4.5 Basic Shell Use . . . 16

1.4.6 Self-test . . . 17

1.5 Still Need Windows? Don't Panic! . . . 18

1.5.1 Cygwin: Try This First . . . 18

1.5.2 Windows Subsystem for Linux: Another Compatibility Layer . . . 31

1.6 Logging In Remotely . . . 32

1.6.1 Unix to Unix . . . 32

1.6.2 Windows to Unix . . . 33

Cygwin . . . 33

PuTTY . . . 33

1.6.3 Terminal Types . . . 36

1.6.4 Self-test . . . 36

1.7 Unix Command Basics . . . 37

1.7.1 Self-test . . . 38
1.8 Basic Shell Tools . . . 38
1.8.1 Common Unix Shells . . . 38
1.8.2 Command History . . . 39
1.8.3 Auto-completion . . . 40
1.8.4 Command-line Editing . . . 40
1.8.5 Globbing (File Specifications) . . . 40
1.8.6 Self-test . . . 41
1.9 Processes . . . 41
1.9.1 Self-test . . . 42
1.10 The Unix File system . . . 42
1.10.1 Unix Files . . . 42
Text vs Binary Files . . . 42
Unix vs. Windows Text Files . . . 43
1.10.2 File system Organization . . . 43
Basic Concepts . . . 43
Absolute Path Names . . . 44
Current Working Directory . . . 45
Relative Path Names . . . 45
Avoid Absolute Path Names . . . 46
Special Directory Names . . . 47
1.10.3 Ownership and Permissions . . . 47
Overview . . . 47
Viewing Permissions . . . 48
Setting Permissions . . . 48
1.10.4 Self-test . . . 49
1.11 Unix Commands and the Shell . . . 51
1.11.1 Internal Commands . . . 51
1.11.2 External Commands . . . 51
1.11.3 Getting Help . . . 52
1.11.4 A Basic Set of Unix Commands . . . 53
File and Directory Management . . . 53
Shell Internal Commands . . . 55
Text File Processing . . . 56
Text Editors . . . 57
Networking . . . 57
Identity and Access Management . . . 58
Terminal Control . . . 58
1.11.5 Self-test . . . 58

1.12 Unix Command Quick Reference . . . 59
1.13 POSIX and Extensions . . . 59
1.14 File Transfer . . . 61
1.14.1 File Transfers from Unix . . . 61
1.14.2 File Transfer from Windows without Cygwin . . . 63
1.14.3 Self-test . . . 64
1.15 Environment Variables . . . 64
1.15.1 Self-test . . . 65
1.16 Shell Variables . . . 66
1.16.1 Self-test . . . 66
1.17 More Shell Tools . . . 66
1.17.1 Redirection and Pipes . . . 66
Device Independence . . . 66
Redirection . . . 68
Special Files in /dev . . . 70
Pipes . . . 70
1.17.2 Subshells . . . 72
1.17.3 Self-test . . . 72
1.18 Process Control . . . 72
1.18.1 External Commands . . . 73
1.18.2 Special Key Combinations . . . 73
1.18.3 Internal Shell Commands and Symbols . . . 73
1.18.4 Self-test . . . 74
1.19 Remote Graphics . . . 75
1.19.1 Configuration Steps Common to all Operating Systems . . . 75
1.19.2 Graphical Programs on Windows with Cygwin . . . 76
Installation . . . 76
Configuration . . . 77
Start-up . . . 77
1.20 Where to Learn More . . . 77

2 Unix Shell Scripting 78

2.1 What is a Shell Script? . . . 78
2.1.1 Self-test . . . 78
2.2 Scripts vs Programs . . . 78
2.2.1 Self-test . . . 79
2.3 Why Write Shell Scripts? . . . 79
2.3.1 Efficiency and Accuracy . . . 79
Self-test . . . 79

2.3.2 Documentation . . . 80
Self-test . . . 80
2.3.3 Why Unix Shell Scripts? . . . 80
Self-test . . . 80
2.3.4 Self-test . . . 80
2.4 Which Shell? . . . 81
2.4.1 Common Shells . . . 81
2.4.2 Self-test . . . 81
2.5 Writing and Running Shell Scripts . . . 81
2.5.1 Self-test . . . 85
2.6 Shell Start-up Scripts . . . 85
2.6.1 Self-test . . . 86
2.7 Sourcing Scripts . . . 87
2.7.1 Self-test . . . 87
2.8 Scripting Constructs . . . 87
2.9 Strings . . . 87
2.10 Output . . . 88
2.10.1 Self-test . . . 89
2.11 Shell and Environment Variables . . . 89
2.11.1 Assignment Statements . . . 90
2.11.2 Variable References . . . 91
2.11.3 Using Variables for Code Quality . . . 92
2.11.4 Output Capture . . . 93
2.11.5 Self-test . . . 93
2.12 Hard and Soft Quotes . . . 94
2.12.1 Self-test . . . 95
2.13 User Input . . . 95
2.13.1 Self-test . . . 96
2.14 Conditional Execution . . . 96
2.14.1 Command Exit Status . . . 96
2.14.2 If-then-else Statements . . . 97
Bourne Shell Family . . . 97
C shell Family . . . 101
2.14.3 Conditional Operators . . . 102
2.14.4 Case and Switch Statements . . . 104
2.14.5 Self-test . . . 106
2.15 Loops . . . 107
2.15.1 For and Foreach . . . 107
2.15.2 While Loops . . . 108

2.15.3 Self-test . . . 109
2.16 Generalizing Your Code . . . 110
2.16.1 Hard-coding: Failure to Generalize . . . 110
2.16.2 Generalizing with User Input . . . 110
2.16.3 Generalizing with Command-line Arguments . . . 111
Bourne Shell Family . . . 111
C shell Family . . . 111
2.16.4 Self-test . . . 112
2.17 Scripting an Analysis Pipeline . . . 112
2.17.1 What's an Analysis Pipeline? . . . 112
2.17.2 Where do Pipelines Come From? . . . 112
2.17.3 Implementing Your Own Pipeline . . . 112
2.17.4 An Example Genomics Pipeline . . . 113
2.18 Functions and Calling other Scripts . . . 115
2.18.1 Bourne Shell Functions . . . 115
2.18.2 C Shell Separate Scripts . . . 116
2.18.3 Self-test . . . 117
2.19 Alias . . . 117
2.20 Shell Flags and Variables . . . 117
2.21 Arrays . . . 119
2.22 Good and Bad Practices . . . 119
2.23 Here Documents . . . 120
2.24 Common Unix Tools Used in Scripts . . . 121
2.24.1 Grep . . . 121
2.24.2 Stream Editors . . . 122
2.24.3 Tabular Data Tools . . . 123
2.24.4 Sort/Uniq . . . 124
2.24.5 Perl, Python, and other Scripting Languages . . . 125

3 Index **127**

List of Figures

1.1 Sample of a Unix File system 44

List of Tables

1.1 Partial List of Unix Operating Systems . . . 3
1.2 Pkgsrc Build Times . . . 18
1.3 Default Key Bindings in some Shells . . . 40
1.4 Globbing Symbols . . . 40
1.5 Special Directory Symbols . . . 47
1.6 Common hot keys in **more** . . . 52
1.7 Unix Commands . . . 60
1.8 Common Extensions . . . 61
1.9 Standard Streams . . . 68
1.10 Redirection Operators . . . 68
1.11 Pipe Operators . . . 71

2.1 Conventional script file name extensions . . . 82
2.2 Shell Start Up Scripts . . . 85
2.3 Printf Format Specifiers . . . 89
2.4 Special Character Sequences . . . 89
2.5 Test Relational Operators . . . 99
2.6 . . . 100
2.7 C Shell Relational Operators . . . 102
2.8 Shell Conditional Operators . . . 103
2.9 . . . 121

Chapter 1

Using Unix

Before You Begin
If you think the word "Unix" refers to Sumerian servants specially "trained" to guard a harem, you've come to the right place. This chapter is designed as a tutorial for users with little or no Unix experience.
If you are following this guide as part of an ungraded workshop, please feel free to work together on the exercises in this text. It would be very helpful if experienced users could assist less experienced users during the "practice breaks" in order to keep the class moving forward and avoid leaving anyone behind.

1.1 Keep It Simple, Stupid

Most people make most things far more complicated than they need to be. Engineers and scientists, especially so.

Aside
A normal person says "If it ain't broke, don't fix it."
An engineer says "If it ain't broke, it doesn't have enough features yet."

We achieve more when we make things simple for ourselves.

We achieve less when we make things complicated.

Most people choose the latter.

The original Unix designers were an exception. Unix is designed to be as simple and elegant as possible. Some things may not seem intuitive at first, but this is probably because the first idea you would come up with is more convoluted than the Unix way. The Unix developers had the wisdom to constantly look for more elegant ways to achieve their goals instead of the most amazing ones or the first one that worked.

Learning the Unix way will therefore make you a wiser and happier computer user. I speak from experience.

Complexity is the product of carelessness or ego, and simplicity is the product of a wise and clear thinker.

Aside
"Simplicity is the ultimate sophistication."
-- Leonardo da Vinci

Unix is not hard to learn. You may have gotten the impression that it's a complicated system meant for geniuses while listening to geniuses talk about it. Don't let them fool you, though. The genius ego compels every genius to make things sound really hard, so you'll think they're smarter than you.

Another challenge with learning anything these days is filtering out all the noise on the Internet. Most tutorials on any given subject are incomplete and many contain misinformation or bad advice. As a result, new users are often pointed in the wrong direction and hit a dead end before long. One of the goals of this guide is a show a simple, sustainable, portable, and expandable approach to using Unix systems. This will reduce your learning curve by an order of magnitude.

Unix has grown immensely since it was created, but the reality is, you don't need to know a lot in order to use Unix effectively. The average Unix user can learn almost everything they'll need to know in a day or two. You can become more sophisticated over time if you want, but most Unix users don't really need to. It may be better to stick to the KISS principal (Keep It Simple, Stupid) and focus on becoming resourceful using the basic tools, rather than accumulating a lot of knowledge that you'll rarely use.

Aside Einstein was once asked how many feet are in a mile. His reply: "I don't know, why should I fill my brain with facts I can find in two minutes in any standard reference book?"

Unix is designed to be as simple as possible and to allow you to work as fast as possible, by staying out of your way. Many other systems will slow you down by requiring you to use cumbersome user interfaces or spend time learning new proprietary methods. As you become a master of Unix, your productivity will be limited only by the speed of the hardware and programs you run.

If something is proving difficult to do under Unix, you're probably going about it wrong. There is almost always an easier way, and if there isn't, then you probably shouldn't be trying to do what you're trying to do. If it were a wise thing to do, some Unix developer would have invented an elegant solution by now. Adapt to the wisdom of those who traveled this road before you, and life will become simpler.

1.2 What is Unix?

1.2.1 Aw, man... I Have to Learn Another System?

Well, yeah, but it's the last time, I promise. As you'll see in the sections that follow, once you've learned to use Unix, you'll be able to use your new skills on virtually any computer. Over time you'll get better and better at it, and never have to start over from scratch again. Read on to find out why.

If you plan to do computational research, you have two choices:

- Learn to use Unix.
- Rely on the charity of others.

Most scientific software runs only on Unix and very little of it will ever have a graphical or other user interface that allows you to run it without knowing Unix.

There have been many attempts to provide access to scientific software via web interfaces, but most of them die out after a short time. Many are created by graduate students who move on and those created by more seasoned professionals usually become to much of a burden to sustain.

Hence, in order to be independent in your research computing, you must know how to use Unix in the traditional way. This is the reality of research computing. It's much easier to adapt yourself to reality than to adapt reality to yourself. This chapter will help you become proficient enough to survive and even flourish on your own.

Unix began as the trade name of an operating system developed at AT&T Bell Labs around 1970. It quickly became the model on which most subsequent operating systems have been based. Eventually, "Unix" came into common use to refer to any operating system mimicking the original Unix, much like Band-Aid is now used to refer to any adhesive bandage purchased in a drug store.

Over time, formal standards were developed to promote compatibility between the various Unix-like operating systems, and eventually, Unix ceased to be a trade name. Today, the name Unix officially refers to a set of standards to which most operating systems conform.

Look around the room and you will see many standards that make our lives easier. (Wall outlets, keyboards, USB ports, light bulb sockets, etc.) All of these standards make it possible to buy interchangeable devices from competing companies. This competition forces the companies to offer better value. They need to offer a lower price and/or better quality than their competition in order to stay in business.

In a nutshell, Unix is every operating system you're likely to use except Microsoft Windows. Table 1.1 provides links to many Unix-compatible operating systems. This is not a comprehensive list. Many more Unix-like systems can be found by searching the web.

Name	Type	URL
AIX (IBM)	Commercial	https://en.wikipedia.org/wiki/IBM_AIX
CentOS GNU/Linux	Free	https://en.wikipedia.org/wiki/CentOS
Debian GNU/Linux	Free	https://en.wikipedia.org/wiki/Debian
DragonFly BSD	Free	https://en.wikipedia.org/wiki/DragonFly_BSD
FreeBSD	Free	https://en.wikipedia.org/wiki/FreeBSD
GhostBSD	Free	https://en.wikipedia.org/wiki/GhostBSD
HP-UX	Commercial	https://en.wikipedia.org/wiki/HP-UX
illumos	Free	https://en.wikipedia.org/wiki/Illumos
JunOS (Juniper Networks)	Commercial	https://en.wikipedia.org/wiki/Junos
Linux Mint	Free	https://en.wikipedia.org/wiki/Linux_Mint
MidnightBSD	Free	https://en.wikipedia.org/wiki/MirOS_BSD
MirOS BSD	Free	https://en.wikipedia.org/wiki/MirOS_BSD
NetBSD	Free	https://en.wikipedia.org/wiki/NetBSD
OpenBSD	Free	https://en.wikipedia.org/wiki/OpenBSD
OpenIndiana	Free	https://en.wikipedia.org/wiki/OpenIndiana
OS X (Apple Macintosh)	Commercial	https://en.wikipedia.org/wiki/OS_X
Redhat Enterprise Linux	Commercial	https://en.wikipedia.org/wiki/Red_Hat_Enterprise_Linux
Slackware Linux	Free	https://en.wikipedia.org/wiki/Slackware
SmartOS	Free	https://en.wikipedia.org/wiki/SmartOS
Solaris	Commercial	https://en.wikipedia.org/wiki/Solaris_(operating_system)
SUSE Enterprise Linux	Commercial	https://en.wikipedia.org/wiki/SUSE_Linux_Enterprise_Desktop
Trident	Free	https://en.wikipedia.org/wiki/TrueOS
TrueOS (Server OS)	Free	https://en.wikipedia.org/wiki/TrueOS
Ubuntu Linux (See also Kubuntu, Lubuntu, Xubuntu)	Free	https://en.wikipedia.org/wiki/Ubuntu_(operating_system)

Table 1.1: Partial List of Unix Operating Systems

Note Apple's Mac OS X has many proprietary extensions, including Apple's own user interface, but is almost fully Unix-compatible and can be used much like any other Unix system by simply choosing not to use the Apple extensions.

Unix is historically connected with other standards such as X/Open XPG and POSIX (Portable Operating System Interface based on Unix). The Unix-related standards are fact the *only* open standards in existence for operating systems. For the official definition of Unix and associated standards, see the Open Group website: http://www.unix.org/what_is_unix.html.

The Unix standards serve the same purpose as all standards; to foster collaboration, give the consumer freedom of choice, reduce unnecessary learning time, and annoy developers who would rather ignore what everyone else is doing and reinvent the wheel at their employer's expense to gratify their own egos.

Unix standards make things interchangeable in the same way as many other standards, such as power plugs, USB, cell phone SIM cards, etc.

They allow us to become operating system agnostic nomads, readily switching from one Unix system to another as our needs or situation dictate.

Note
When you develop programs for any Unix-compatible operating system, those programs can be easily used by people running any other Unix-compatible system. Most Unix programs can even be used on a Microsoft Windows system with the aid of a *compatibility layer* such as Cygwin (See Section 1.5.1).
I once knew a mechanic whose wife was concerned about how often he changed jobs. His response: "That's why tool-boxes have wheels." Likewise, you can write programs with "wheels" that can easily be taken elsewhere to run on other Unix-compatible systems, or you can bolt your programs to the floor of a proprietary system like MS Windows and abandone them when it's time to move.

Note Once you've learned to use one Unix system, you're ready to use any of them. Hence, Unix is the last system you'll ever need to learn!

Unix systems run on everything from your cell phone to the world's largest supercomputers. Unix is the basis for Apple's iOS, the Android mobile OS, embedded systems such as networking equipment and robotics controllers, most PC operating systems, and many large mainframe systems.

Many Unix systems are completely free (as in free beer) and can run thousands of high quality free software packages.

It's a good idea to regularly use more than one Unix system. This will make you aware of how much they all have in common and what the subtle differences are.

1.2.2 Operating System or Religion?

Aside
Keep the company of those who seek the truth, and run from those who have found it.
-- Vaclav Havel

The more confident someone is in their views, the less they probably know about the subject. As we gain life experience and wisdom, we become less certain about everything and more comfortable with that uncertainty.

What looks like confidence is usually a symptom of ignorance of our own ignorance, generally fueled by ego.

If you discuss operating systems at length with most people, you will discover, as the ancient philosopher Socrates did while discussing many topics with "experts", that their views are not based on broad knowledge and objective comparison. Before taking advice from anyone, it's a good idea to find out how much they really know and what role emotion and ego play in their preferences. Most people quickly become attached to their views and expend more effort clining to them than it would actually take to verify or refute them.

The whole point of the Unix standard, like any other standard, is freedom of choice.

However, you won't have any trouble finding evangelists for a particular brand of Unix-compatible operating system on whom this point is lost. "Discussions" about the merits of various Unix implementations often involve arrogant pontification and emotional outbursts, and even occasional cussing.

If you step back and ask yourself what kind of person gets emotionally attached to a piece of software, you'll realize whose advice you should value and whose you should not. Rational people will keep an open mind and calmly discuss the objective measures of an OS, such as performance, reliability, security, easy of maintenance, specific capabilities, etc. They will also back up their opinions with facts rather than try to bully you into agreeing with their views.

If someone tells you that a particular operating system "isn't worth using", "is way behind the times", or "sucks wads", rather than asking you what you need and objectively discussing alternatives, this is probably someone whose advice you can safely ignore.

Evangelists are typically pretty easy to spot. They will somehow magically assess your needs without asking you a single question and proceed to explain (often aggressively) why you should be using their favorite operating system or programming language. They tend to have limited or no experience with other alternatives. This is easy to expose with a minute or two of

questioning. So this OS is garbage? How many years of experience to you have with it? For what types of problems have you attempted to use it?

Ultimately, the system that most easily runs your programs to your satisfaction is the best one for you. That could turn out to be BSD, Cygwin, Linux, Mac OS X, or any other.

Someone who knows what they're doing and truly wants to help you will always begin by asking you a series of questions in order to better understand your needs. "What program(s) do you need to run?", "Do they require any special hardware?", "Do you need to run any commercial software, or just open source?", etc. They will then consider multiple alternatives and inform you about the capabilities of each one that might match your needs.

Unfortunately, many people in science and engineering are not so rational or knowledgeable, but driven by ego. They want to look smart or be cool by using the latest trendy technologies, even if those technologies do nothing to meet their needs. In the minds of the ego-driven, new technologies become solutions looking for problems.

Some recent "we're not cool unless we use this" fads include Hadoop, cgroups, solid state disk drives (SSDs), ultrathin laptops, GPUs for scientific computing, parallel file systems, machine learning, Bayesian networks, containers, etc. All of these technologies are very useful under the right circumstances, but many people waste time and money trying to apply them to ordinary tasks that don't benefit from them at all, and may actually suffer due to the high cost and added man-hours wasted on them.

For example, SSDs are roughly three times faster than magnetic disks. However, they cost a lot more for the same capacity. Does the added speed they provide actually allow you to accomplish something that could not be done if your program took a little longer to run on a magnetic disk? How much does an SSD actually reduce your run time? If your software is mainly limited by CPU speed, you won't see any measurable reduction in run-time by switching to an SSD.

The vastly larger capacity/cost ratio of a magnetic disk might be of more value to you. SSDs also burn out over time, as they have a limited number of write cycles. An SSD on a computer with constant heavy disk activity will improve performance, but the SSD might be dead in a year or two. Magnetic disks on average actually last longer, despite being more physically fragile than SSDs.

All scientists and engineers are capable of logical thought, but outside peer-reviewed publications, many only use it to rationalize what they want to believe despite all evidence to the contrary.

Aside

"I don't understand why some people wash their bath towels. When I get out of the shower, I'm the cleanest object in my house. In theory, those towels should be getting cleaner every time they touch me... Are towels supposed to bend?"
-- Wally (Dilbert)

Of course, the "more is always better" fallacy is not limited to science and engineering. Many people waste vast amounts of time and money on things that have little or no real impact on their lives (a new set of golf clubs, a racing jersey for weekend bike rides, four wheel drive, the latest iPhone, etc.) Avoid falling into this trap and life will be simpler, more productive, and more relaxing.

My personal recommendations for running Unix software (for now, these could change in the distant future) are listed below. Note that these recommendations are meant to indicate only what is optimal to minimize your wasted effort, not what is necessary to succeed in your work. All of these systems are somewhat interchangeable with each other and the many other Unix based systems available, so deviating from these recommendations will not lead to castastrophe.

More details on choosing a Unix platform are provided in [?].

- Servers running mostly open source software: FreeBSD.

  FreeBSD is extremely fast, reliable, and secure. Software management is very easy with FreeBSD ports, which offers over 33,000 distinct and usually very recent software packages (not counting different builds of the same software). The ports system supports installation via either generic binary packages, or you can just as easily build from source with custom options or optimizations for your specific CPU. With the Linux compatibility module, FreeBSD can directly run most Linux closed-source programs with no performance penalty and a little added effort and resources.

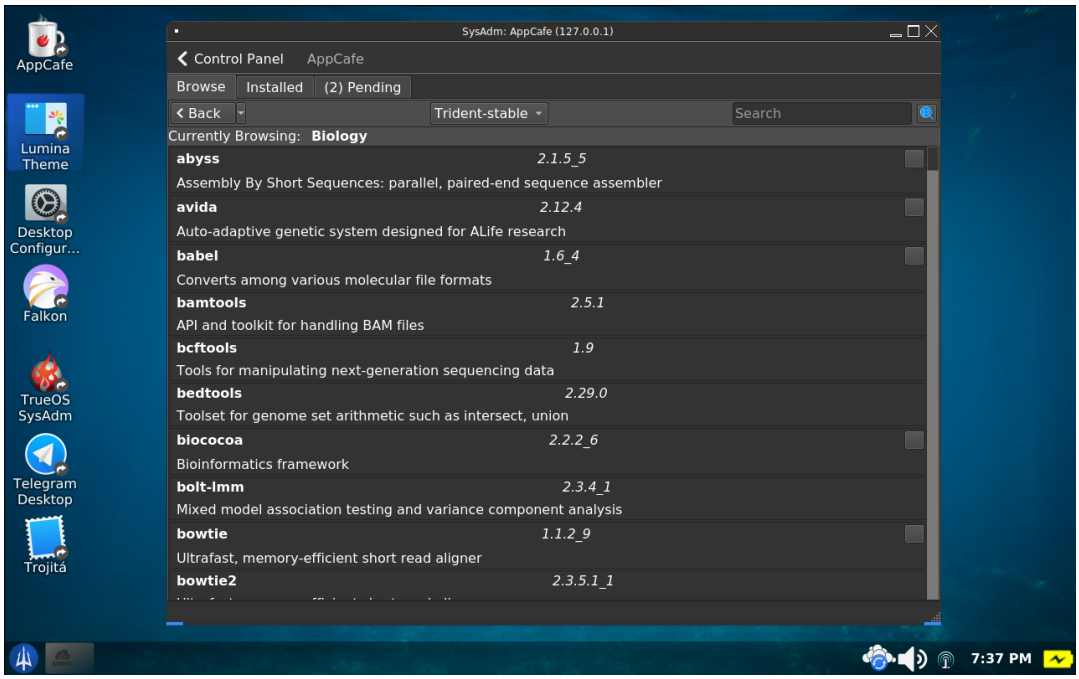
- Servers running mainly or commercial applications or CUDA GPU software: Enterprise Linux (CentOS, RHEL, Scientific Linux, SUSE).

These systems are designed for better reliability, security, and long-term binary compatibility than bleeding-edge Linux systems. They are the only platforms besides MS Windows and Mac OS X supported by many commercial software vendors. While you may be able to get some commercial engineering software running on Ubuntu or Mint, it is often difficult and the company will not provide support. Packages in the native Yum repository of enterprise Linux are generally outdated, but more recent open source software can be installed using a separate add-on package manager such as pkgsrc.

- An average Joe who wants to browse the web, use a word processor, etc.: Debian, GhostBSD, Trident, Ubuntu, or similar open source Unix system with graphical installer and management tools, or Macintosh.

  These systems make it easy to install software packages and system updates, with minimal risk of breakage that a non computer wizard would know how to fix.

Debian Linux



Trident Desktop OS

- Someone who uses mostly Windows-based software, but needs a basic Unix environment for software development or connecting to other Unix systems: A Windows PC with Cygwin.

  Cygwin is free, entirely open source, and very easy to install in about 10 minutes on most Windows systems. It has some performance bottlenecks, fewer packages than a real Unix system running on the same machine, and a few other limitations, but it's more than adequate for the needs of many typical users.

  WSL (Windows Services for Linux) is an alternative to Cygwin which is binary compatible with a real Linux system such as Debian, but it lacks graphical capabilities, is slower than Cygwin (see Table 1.2), and not entirely open source, leaving you at the mercy of Microsoft if you become dependent on it.

1.2.3 The Unix Standard API

Programmers cost money. This is a problem in the computer industry for which we haven't found a solution. Even if we keep them locked in a basement for days at a time and only pay them for half the hours they work (which many of them oddly find perfectly acceptable), they still need to be fed and watered occasionally, and paid a reasonably competitive salary so they won't leave and go work in someone else's basement.

Unix systems adhere to an *application program interface* (API) standard, which means that programs written for one Unix-based system can be run on any other with little or no modification, even if it runs on completely different hardware. For example, programs written for an Intel/AMD-based Linux system will also run a PowerPC based Mac, a Solaris system running on a Sparc processor, or FreeBSD on an ARM processor.

An API defines a set of functions (subprograms) used to request services from the operating system, such as opening a file, allocating memory, running another program, etc.

These functions are the same on all Unix systems, but some of them are different on Windows and other non-standard systems. For example, to open a file in a C program on any Unix system, one could use the fopen() function:

```
FILE *fopen(const char *filename, const char *mode);
```

Microsoft compilers also support fopen(), but also provide another function for the same purpose that won't work on other systems:

```
errno_t fopen_s(FILE** pFile, const char *filename, const char *mode);
```

Note Microsoft claims that fopen_s() is more secure, which is debatable. Note however, that even if this is true, the existing fopen() function itself could have been made more secure rather than creating a separate, non-standard function that does the same thing.

Here are a few other standard Unix functions that can be used in programs written in C and most other compiled languages. These functions can be used on any Unix system, regardless of the type of hardware running it. Some of these may also work in Windows, but for others, Windows uses a completely different function to achieve the same goal.

```
chdir()         // Change current working directory
execl()         // Load and run another program
mkdir()         // Create a directory
unlink()        // Remove a file
sleep()         // Pause execution of the process
DisplayWidth()  // Get the width of the screen
```

Because the Unix API is platform-independent, it is also possible to compile and run most Unix programs on Windows with the aid of a *compatibility layer*, software that bridges the difference between two platforms. (See Section 1.5.1 for details.) It is not generally possible to compile and run Windows software on Unix, however, since Windows has many PC-specific features.

Since programs written for Unix can be run on almost any computer, including Windows computers, they will probably never have to be rewritten in order to run somewhere else!

Programs written for non-Unix platforms will only run on that platform, and will have to be rewritten (at least partially) in order to run on any other system. This leads to an enormous waste of man-hours that could have gone into creating something new.

They may also become obsolete as they proprietary systems for which they were written evolve. For example, most programs written for MS DOS and Windows 3.x are no longer in use today, while programs written for Unix around that same time will still work on modern Unix systems.

Of course, if you had a lot of fun writing a program the first time, you may *want* to do it again. In that case, you won't want to use Unix, since it would take away at least half your fun.

1.2.4 Shake Out the Bugs

Another advantage of programming on standardized platforms is the ability to easily do more thorough testing.

Compiling and running a program on multiple operating systems and with multiple compilers will almost always expose bugs that you were unaware of while running it on the original development system. The same bug will have different effects on different operating systems, with different compilers or interpreters, or with different compile options (e.g. with and without optimization).

For example, an errant array subscript or pointer might cause corruption in a non-critical memory location in some environments, while causing the program to crash in others.

A program may seem to be fine when you compile it with Clang and run it on your Mac, but may not compile, or may crash when compiled with GCC on a Linux machine.

Finding bugs *now* may save you from the stressful situation of tracking them down under time pressure later, with an approaching grant deadline. A bug that was invisible on your Mac for the test cases you've used could also show up on your Mac later, when you run the program with different inputs.

Developing for the Unix API makes it easy to test on various operating systems and with different compilers. There are many free BSD and Linux based systems, as well as free compilers such as Clang and GCC. Most of them can be run in a virtual machine ([?]), so you don't even need to have another computer for the sake of program testing.

Take advantage of this easy opportunity to stay ahead of program bugs, so they don't lead to missed deadlines down the road.

1.2.5 The Unix Standard UI

Let's face it: Most people don't like to learn new things. At least not about computers. Unix can help with this, too.

All Unix systems support the same basic set of commands, which conform to standards so that they behave the same way everywhere. So, if you learn to use FreeBSD, most of that knowledge will directly apply to Linux, Mac OS X, Solaris, etc.

Another part of the original Unix design philosophy was to do everything in the simplest way possible. As you learn Unix, you will likely find some of its features befuddling at first. However, upon closer examination, you will often come to appreciate the elegance of the Unix solution to a difficult problem. If you're observant enough, you'll learn to apply this Zen-like simplicity to your own work, and maybe even your everyday life. Who knows, mastering Unix could even help you attain enlightenment someday.

You will also gradually recognize a great deal of consistency between various Unix commands and functions. For example, many Unix commands support a `-v` (verbose) flag to indicate more verbose output, as well as a `-q` (quiet) flag to indicate no unnecessary output. Over time, you will develop an intuitive feel for Unix commands, become adept at correctly guessing how things work, and feel downright God-like at times.

Unix documentation also follows a few standard formats, which users quickly get used to, making it easier to learn new things about commands on any Unix system.

The consistency provided by Unix standards minimizes the amount of knowledge Unix users need in order to effectively utilize the numerous Unix systems available.

In a nutshell, the time and effort you spend learning any Unix system will make it easy to use any other in the future. You need only learn Unix once, and you'll be proficient with many different implementations such as FreeBSD, Linux, and Mac OS X.

1.2.6 Freedom of Choice

Unix standards are designed to give the user as much freedom of choice as possible. Unix users can run their programs on virtually any type or brand of hardware, and switch at will when a better or cheaper option appears.

As a case in point, until the early 1990's, most Unix systems were high-end workstations or minicomputers costing $10,000 or more. Many of the same programs that ran on those systems are now running on commodity PCs and even laptops that cost a few hundred dollars. In fact, at this very moment I'm editing this chapter on an old ThinkPad someone else threw away, which is now running FreeBSD with the XFCE desktop environment.

Another of the main design goals of Unix is to stay out of the user's way. With freedom comes responsibility, though. A common quip about Unix is that it gives us the freedom to shoot ourselves in the foot. Unix does a lot to protect users from each other, but very little to protect users from themselves. This usually leads to some mistakes by new users, but most users quickly become conditioned to be more careful and come to prefer the freedom Unix offers over more restrictive, cumbersome systems.

1.2.7 Fast, Stable and Secure

Since Unix systems compete directly with each other to win and retain users running the same programs, developers are highly motivated to optimize objective measures of the system such as performance, stability, and security.

Most Unix systems operate near the maximum speed of the hardware on which they run. Unix systems typically respond faster than other systems on the same hardware and run intensive programs in less time. Many Unix systems require far fewer resources than non-Unix systems, leaving more disk and memory for use by your programs.

Unix systems may run for months or even years without being rebooted. Software installations almost never require a reboot, and even most security updates can be applied without rebooting. As a professional systems manager, I run the risk of forgetting that some of my Unix systems exist because I haven't touched them for so long. I'm occasionally reminded when a sleep-deprived graduate student nods off at his desk and knocks the monitor to the floor with a nervous twitch.

Stability is critical for research computing, where computational models often run for weeks or months. Users of non-Unix operating systems often have to choose between killing a process that has been running for weeks and neglecting critical security updates that require a reboot.

Very few viruses or other malware programs exist for Unix systems. This is in part due to the inherently better security of Unix systems and in part due to a strong tradition in the Unix community of discouraging users from engaging in risky practices such as running programs under an administrator account and installing software from pop-ups on the web.

1.2.8 Sharing Resources

Your mom probably told you that it's nice to share, but did you know it's also more efficient?

One of the major problems for researchers in computational science is managing their own computers.

Most researchers aren't very good at installing operating systems, managing software, apply security updates, etc., nor do they want to be. Unfortunately, they often have to do these things in order to conduct computational research.Computers managed by a tag-team of researchers usually end up full of junk software, out-of-date, full of security issues, and infected with malware.

Some universities have staff to help with research computing support, but most do not. Good IT guys are expensive and hard to find, so it takes a critical mass of demand from researchers to motivate the creation of a research computing support group.

Since Unix is designed from the ground up to be accessed remotely, Unix creates an opportunity to serve researchers' needs far more cost-effectively than individual computers for each researcher. A single Unix machine on a modern PC can support dozens or even hundreds of users at the same time.

IT staff managing one or a few central Unix hosts can serve the needs of many researchers, freeing the researchers to focus on what they do best. All the researchers need is a computer that can connect to the central Unix host, and the systems managers of the host can take care of all the hard stuff.

1.3 Self-test

1. Is Unix an operating system? Why or why not?
2. Which mainstream operating systems are Unix compatible and which are not?
3. What is the advantage of the Unix API over the APIs of non-Unix operating systems?
4. What is the advantage of the Unix UIs over the UIs of non-Unix operating systems?
5. What is the major design goal of the Unix standard?

1.4 Unix User Interfaces

A *user interface*, or *UI*, refers to the software that allows a person to interact with the computer. The UI provides the look and feel of the system, and determines how easily and efficiently it can be used. (Note that ease of use and efficiency are not the same!)

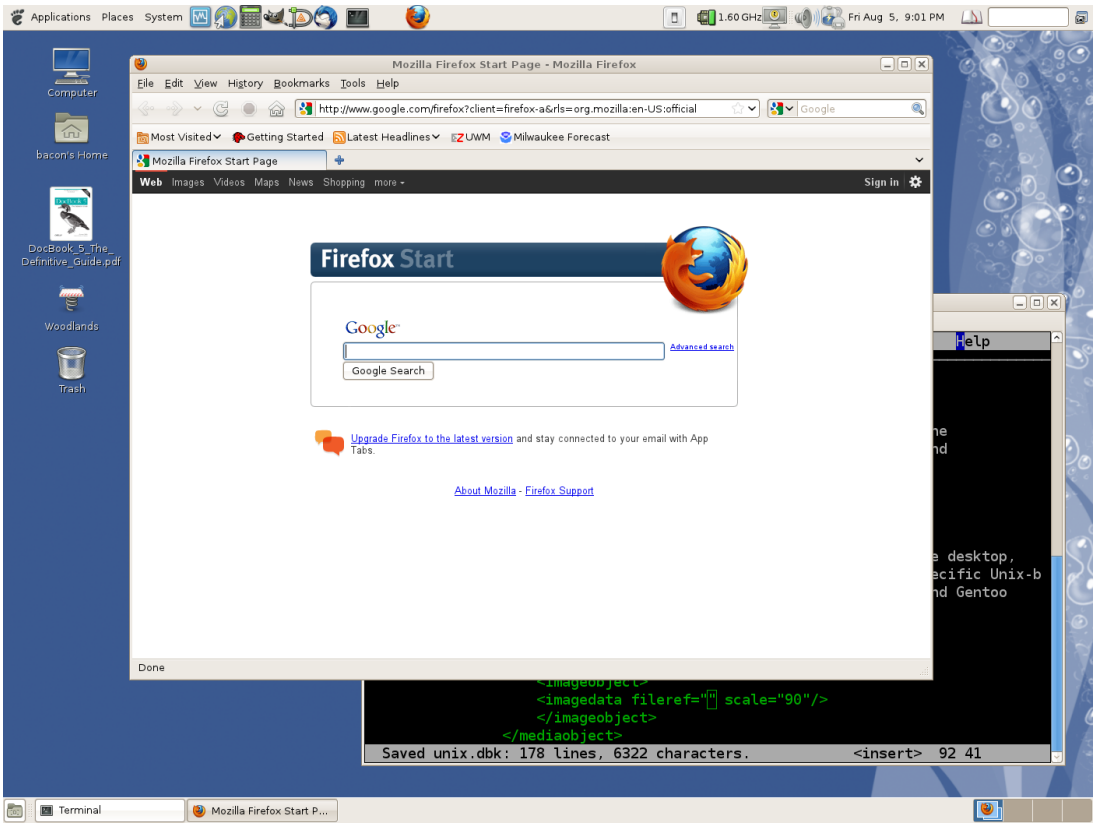
The term "Windows" refers to a a specific proprietary operating system, and implies all of the features of that system including the API and the UI. When people think of Windows, they think of the Start menu, the Control Panel, etc.

Likewise, "Macintosh" refers to a specific product and invokes images of the "Dock" and a menu bar at the top of the screen vs. attached to a window.

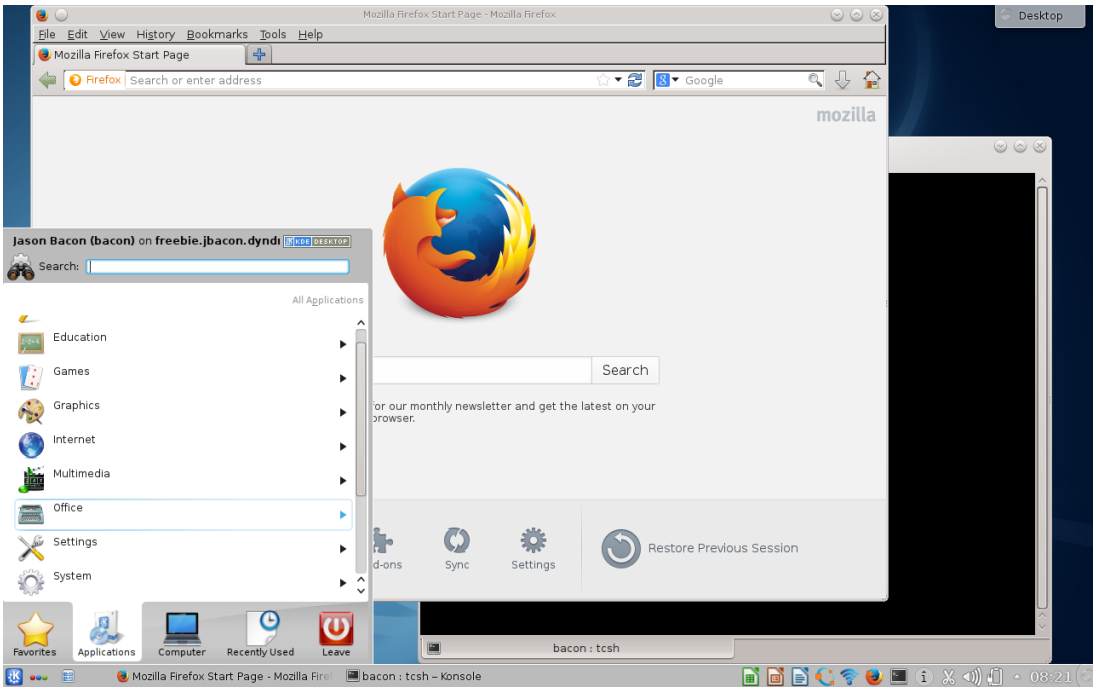
The term "Unix", on the other hand, implies an API, but does not imply a specific UI. There are many UIs available for Unix systems. In fact, a computer running Unix can have many UIs installed, and each user can choose the one they want when the log in.

1.4.1 Graphical User Interfaces (GUIs)

Unlike Microsoft Windows, which has a unique look and feel, there are many different GUIs (pronounced goo-ey) available for Unix. Some of the more popular ones include KDE, Gnome, XFCE, LXDE, OpenBox, CDE, and Java Desktop.

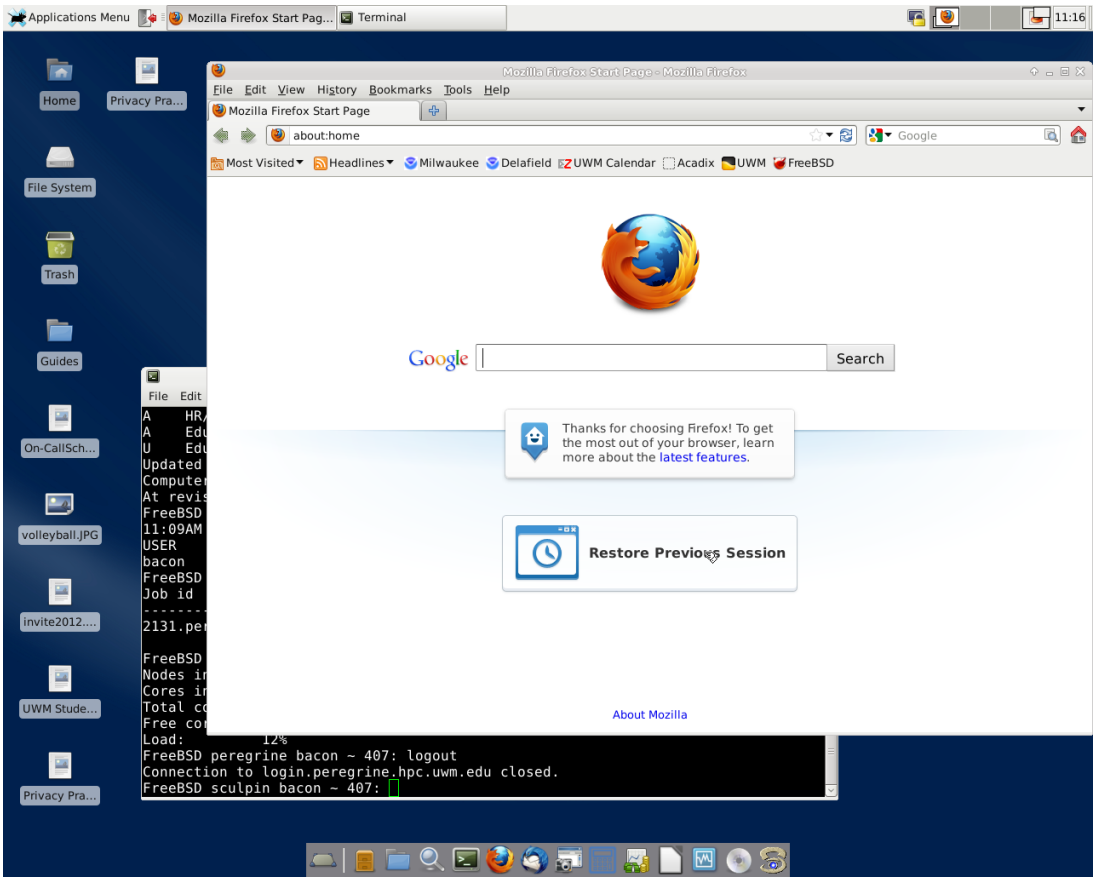


A FreeBSD system running Gnome desktop.



A FreeBSD system running KDE desktop.

A FreeBSD system running Lumina desktop.



A FreeBSD system running XFCE desktop.

Practice Break

If you have access to a Unix GUI, log into your Unix system via the GUI interface now.

All Unix GUIs are built on top of the X11 networked graphics API. As a result, all Unix systems have the inherent ability to display graphics on other Unix systems. I.e., you can remotely log into another Unix computer over a network and run graphical programs that display output wherever you're sitting.

Note This is not the same as a remote desktop system, which only mirrors the console display on a remote system. Unix systems allow multiple users in different locations to run graphical programs independent of each other. In other words, Unix supports multiple independent graphical displays on remote computers.

Most Unix GUIs support multiple *virtual desktops*, also known as *workspaces*. Virtual desktops allow a single monitor to support multiple separate desktop images. It's like having multiple monitors without the expense and clutter. The user can switch between virtual desktops by clicking on a panel of thumbnail images, or in some cases by simply moving the mouse over the edge of the screen.

1.4.2 X11 on Mac OS X

Mac OS X is Unix compatible, derived largely from FreeBSD and the Mach kernel project, with components from GNU and other Unix-based projects.

It differs from more traditional Unix systems like BSD and Linux, though, in that it runs Apple's proprietary graphical API and GUI. Native OS X programs don't use the X11 API, but OS X can also run X11-based programs. See Section 1.19 for instructions on enabling X11 for Mac.

1.4.3 Command Line Interfaces (CLIs): Unix Shells

There are two basic type of user interfaces:

- Menu-driven, where choices are displayed on the screen and the user selects one.
- Command-driven, where the user types commands that they have memorized.

A GUI is a type of menu-driven interface, where the menu items are graphical icons. Some menu systems are simply text.

While modern Unix systems have GUIs, much work is still done via the command line, or *shell*.

Menu-driven systems are much easier to use if you're new to the system or use it infrequently, but can become cumbersome for everyday use. For example, an ATM (automatic teller machine) with a command-driven interface would likely be unpopular among banking customers.

If a user needs access to dozens or hundreds of features, they cannot all be displayed on the screen as icons at the same time. Hence, it will be necessary to navigate through multiple levels of menus or screens to find the functionality you need. Even if they could be displayed all at once, it would be a nightmare to find the one you want among such clutter.

Because of these limitations, most GUIs support *hot keys*, special key combinations that can be used to access certain features without navigating the menus. Hot keys are often shown in menus alongside the features they activate. For example, Command+q can be used on Mac OS X to terminate most graphical applications.

It is also difficult to automate tasks in a menu-driven system. Some systems have this capability, but most do not, and the method of automating is different for each system.

Perhaps the most important drawback of menu-driven systems is non-existence. Programming a menu system, and especially a GUI, requires a lot of grunt-work and testing. As a result, the vast majority of open source software does not and never will have a GUI interface. Open source developers generally don't have the time or programming skills to build and maintain a comprehensive GUI interface.

Caution
If you lack command-line skills, you will be limited to using a small fraction of available open source software. In the tight competition for research grants, those who can use the command-line will usually win.

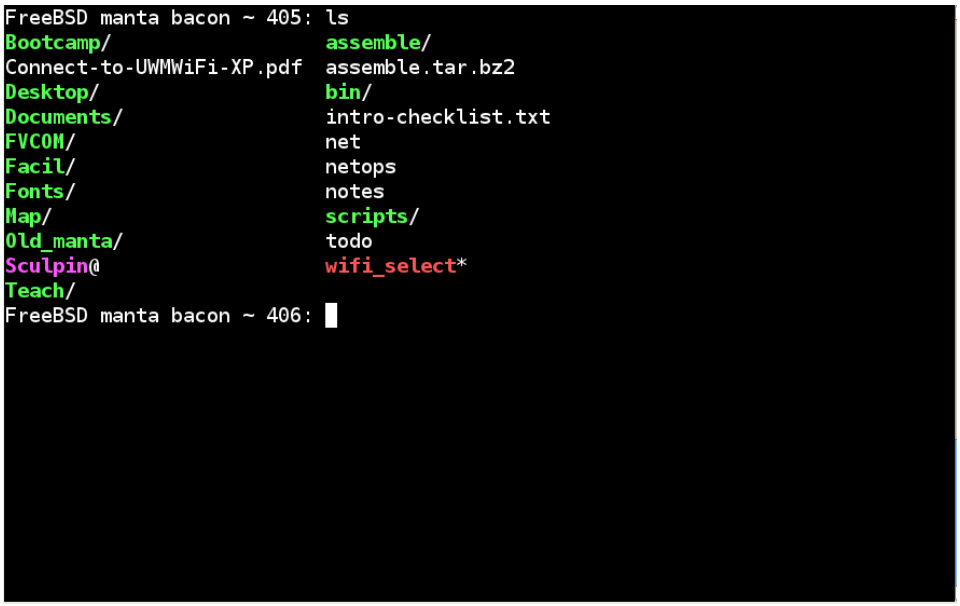
A command line interface, on the other hand, provides instant access to an unlimited number of commands and is easy to automate. We can simply store a sequence of commands in a file, called a *script*.

A command line interface requires some learning, since we need to memorize some commands in order to use it efficiently. However, we usually need only learn a few commands to get started, and once the basics are learned, a command line interface allows for much greater efficiency and flexibility than a GUI.

The small investment in learning a command line interface can have a huge payoff, and yet many people try to avoid it. The result is usually an enormous amount of wasted effort dealing with limited and poorly designed custom user interfaces before eventually realizing that things would have been much easier had they learned to use the command line in the first place. It's amazing how much effort people put into avoiding effort...

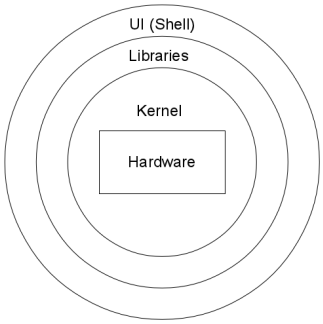
A *shell* is a program that provides the command line interface. It inputs commands from the user, interprets them, and executes them.

Using a shell, you type a command, press enter, and the command is immediately executed.



Unix Shell

The term shell comes from the view of Unix as three layers of software wrapped around the hardware:



A 3-layer Model of Unix

- The innermost layer, which handles all hardware interaction for Unix programs, is called the *kernel*, named after the core of a seed. The Unix kernel effectively hides the hardware from user programs and provides a standard API. This is what allows Unix programs to run on different kinds of computers.
- The middle layer, the libraries, provide a wealth of standard functionality for Unix programmers to utilize. The libraries are like a huge box of Legos that can be used to build all kinds of sophisticated programs.

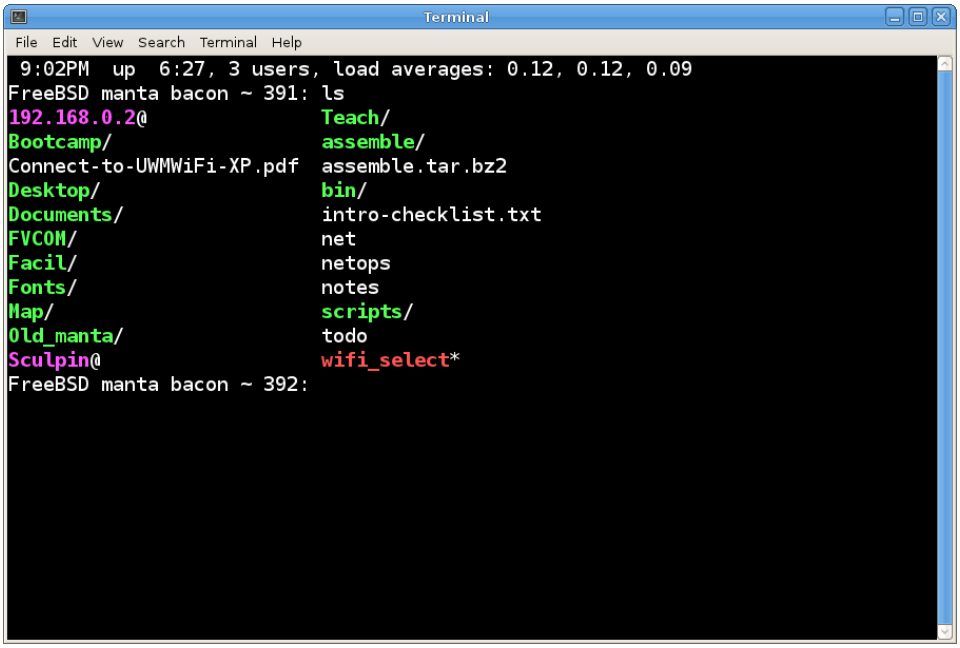
- The outermost layer, the CLI, is called a shell.

1.4.4 Terminals

All that is needed to use a Unix shell is a keyboard and a screen. In the olden days, these were provided by a simple hardware device called a *terminal*, which connected a keyboard and screen to the system through a simple communication cable. These terminals typically did not have a mouse or any graphics capabilities. They usually had a text-only screen of 80 columns by 24 lines, and offered limited capabilities such as moving the cursor, scrolling the screen, and perhaps a limited number of colors.

Digital VT320 terminal

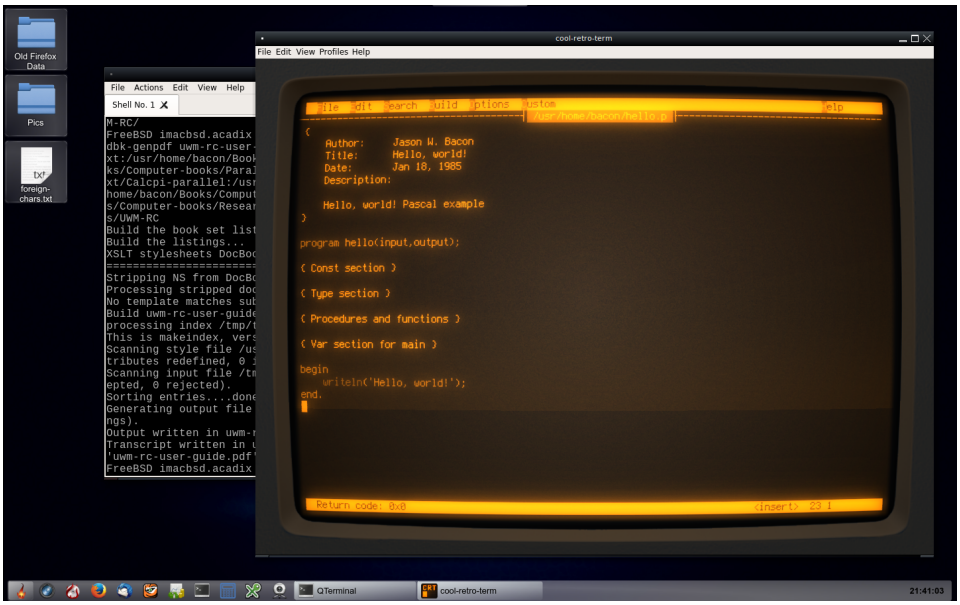
Hardware terminals lost popularity with the advent of cheap personal computers, which can perform the function of a terminal, as well as running programs of their own. Terminals have been largely replaced by *terminal emulators*. A terminal emulator is a simple program that emulates an old style terminal within a window on your desktop.



A Terminal emulator.

All Unix systems come with a terminal emulator program. There are also free terminal emulators for Windows, which are discussed in Section 1.6.

For purists who *really* want to emulate a terminal, there's *Cool Retro Terminal* (CRT for short, which also happens to stand for cathode ray tube). This emulator comes complete with screen distortion and jitter to provide a genuine nostalgic 1970s experience.



Cool Retro Terminal

1.4.5 Basic Shell Use

Once you're logged in and have a shell running in your terminal window, you're ready to start entering Unix commands.

The shell displays a *prompt* (such as "FreeBSD manta bacon ~ 392:" in the image above) to indicate that it's waiting for you to enter the next command. The shell prompt can be customized by each user, so it may be different on each Unix system you use.

Note
For clarity, we primarily use the following to indicate a shell prompt in this text:

```
shell-prompt:
```

The actual shell prompt you see may be different on each individual Unix system you use.
Including the hostname, user name, and other information in a shell prompt is a common practice and very helpful for users who are logged into multiple Unix systems at the same time from the same desktop. This discourages users from running a command on the wrong computer!
Section 1.16 explains how to set your shell prompt in common Unix shells.

The shell prompt in the terminal window above is "FreeBSD manta bacon ~ 392:". At the first prompt, the user entered the command **ls**, which lists the files in the current directory. After **ls** finished, the shell printed another prompt.

To enter a Unix command, you type the command on a single line, edit if necessary (using arrow keys to move around), and press **Enter** or **Return**.

Note
Modern Unix shells allow commands to be extensively edited. Assuming your terminal type is properly identified by the Unix system, you can use the left and right arrow keys to move around, backspace and delete to remove characters (Ctrl+h serves as a backspace in some cases), and other key combinations to remove words, the rest of the line, etc. Learning the editing capabilities of your shell will make you a much faster Unix user, so it's a great investment of a small amount of time.

Practice Break

Log into your Unix system, open a terminal if necessary, and run the following commands:

```
shell-prompt: ls
shell-prompt: ls /
shell-prompt: ls -al
shell-prompt: mkdir -p Data/IRC
shell-prompt: cd Data/IRC
shell-prompt: nano sample.txt

Type in some text, then save the file (press Ctrl+o), and exit nano
(press Ctrl+x).

shell-prompt: ls
shell-prompt: cat sample.txt
shell-prompt: wc sample.txt
shell-prompt: whoami
shell-prompt: hostname
shell-prompt: uname
shell-prompt: date
shell-prompt: cal
shell-prompt: cal nov 2018
shell-prompt: bc -l
scale=50
sqrt(2)
8^2
2^8
a=1
b=2
c=1
(-b+sqrt(b^2-4*a*c))/2*a
2*a
quit

shell-prompt: w
shell-prompt: ls /bin
```

1.4.6 Self-test

1. What is a UI?
2. What is a GUI?
3. What is the difference between Unix and other operating systems with respect to the GUI?
4. What is a CLI?
5. What are the advantages and disadvantages of a CLI vs. a GUI?
6. What is a shell?
7. What is a kernel?
8. What are libraries?
9. What is a terminal?
10. Do people still use hardware terminals today? Explain.
11. What is a shell prompt?

1.5 Still Need Windows? Don't Panic!

For those who need to run software that is only available for Windows, or those who simply haven't tried anything else yet, there are options for getting to know Unix while still using Windows for your daily work.

There are virtual machines (see [?]) that allow us to run Windows and Unix on the same computer, at the same time.

There are also compatibility layers such as Cygwin and Windows Services for Linux (WSL), that allow Unix software to run on Windows.

A compatibility layer is generally easier to install, but as of this writing, both Cygwin and WSL have serious performance limitations in some areas. Purely computational software will run about as fast as it would on a real Unix system, but software that performs a lot of file input/output or other system calls can be much slower than a real Unix system, even one running in a virtual machine.

For example, installing the pkgsrc package manager from scratch, which involves running many Unix scripts and programs, required the times shown in Table 1.2. WSL, Cygwin, and the Hyper-V virtual machine were all run on the same Windows 10 host with a 2.6 GHz Core i7 processor and 4 GiB RAM. The native FreeBSD and Linux builds were run on identical 3.0 GHz Xeon servers with 16 GiB RAM, much older than the Core i7 Windows machine.

Platform	**Time**
WSL	104 minutes
Cygwin	71 minutes
FreeBSD Virtual Machine under Hyper-V	21 minutes
CentOS Linux (3.0 GHz Xeon)	6 minutes, 16 seconds
FreeBSD (3.0 GHz Xeon)	5 minutes, 57 seconds

Table 1.2: Pkgsrc Build Times

I highly recommend Cygwin or WSL as a light-duty Unix environment under Windows, for connecting to other Unix systems or developing small Unix programs. For serious Unix development or heavy computation, obtaining a real Unix system, even under a virtual machine, would be a wise investment of your time.

1.5.1 Cygwin: Try This First

Cygwin is a *compatibility layer*, another layer of software on top of Windows that translates the Unix API to the Windows API. As such, performance is not as good as a native Unix system on the same hardware, but it's more than adequate for many purposes.

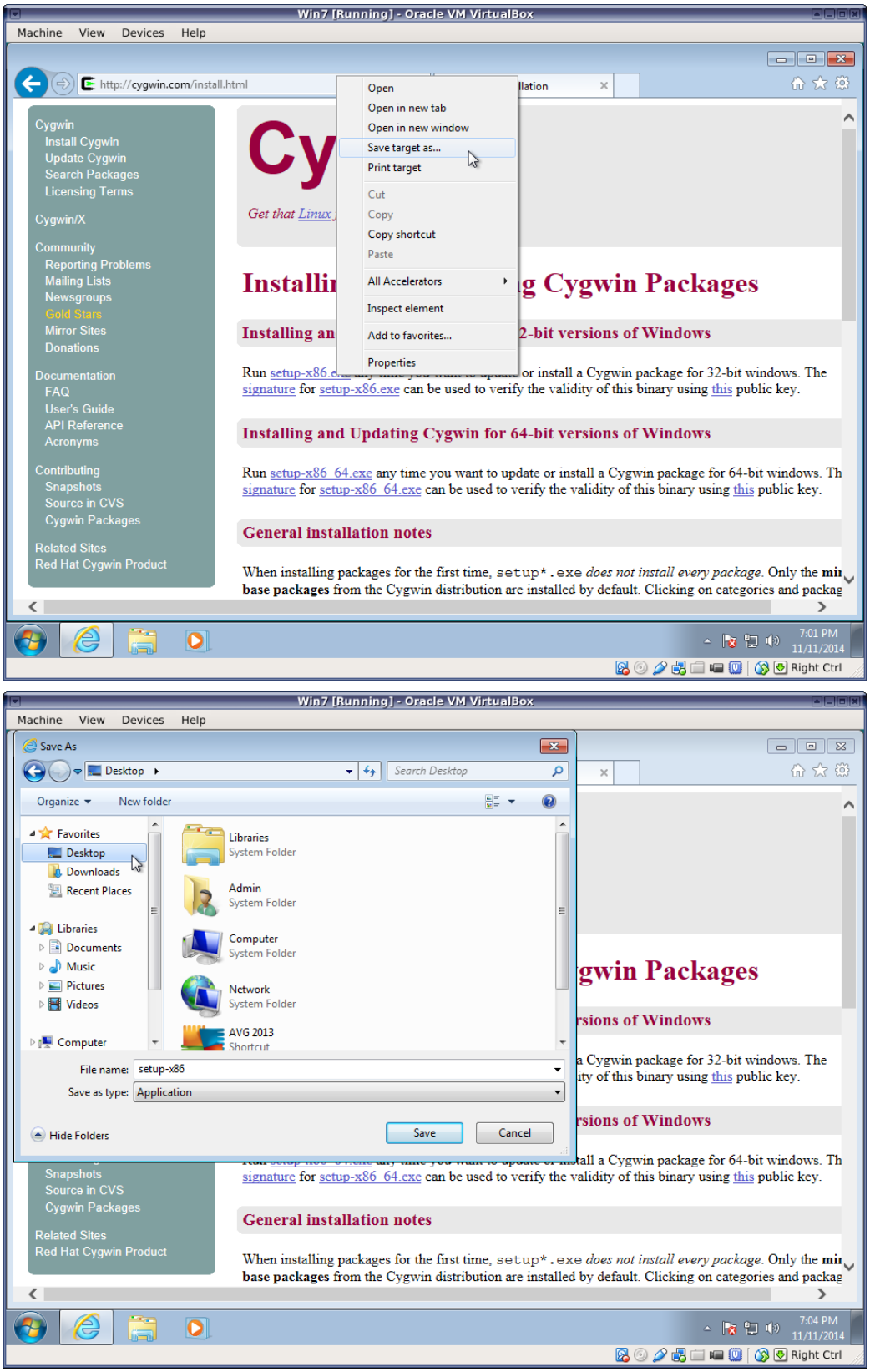
Cygwin may not be ideal for heavy-duty data analysis where optimal performance is required, but it is an excellent system for basic development and testing of Unix code and for interfacing with other Unix systems.

Installation will take about 10 minutes on a modern machine with a fast Internet connection. It won't break your Windows configuration, since it is completely self-contained in its own directory. Given that it's so easy to install and free of risk, there's no point wasting time wondering whether you should use Cygwin, a virtual machine, or some other method to get a Unix environment on your Windows PC. Try Cygwin first and if it fails to meet your needs, try something else.

Cygwin is a free collection of Unix software, including many system tools from Linux and other Unix-compatible systems, ported to Windows. It can be installed on any typical Windows machine in a few minutes and allows users to experience a Unix user interface as well as run many popular Unix programs right on the Windows desktop.

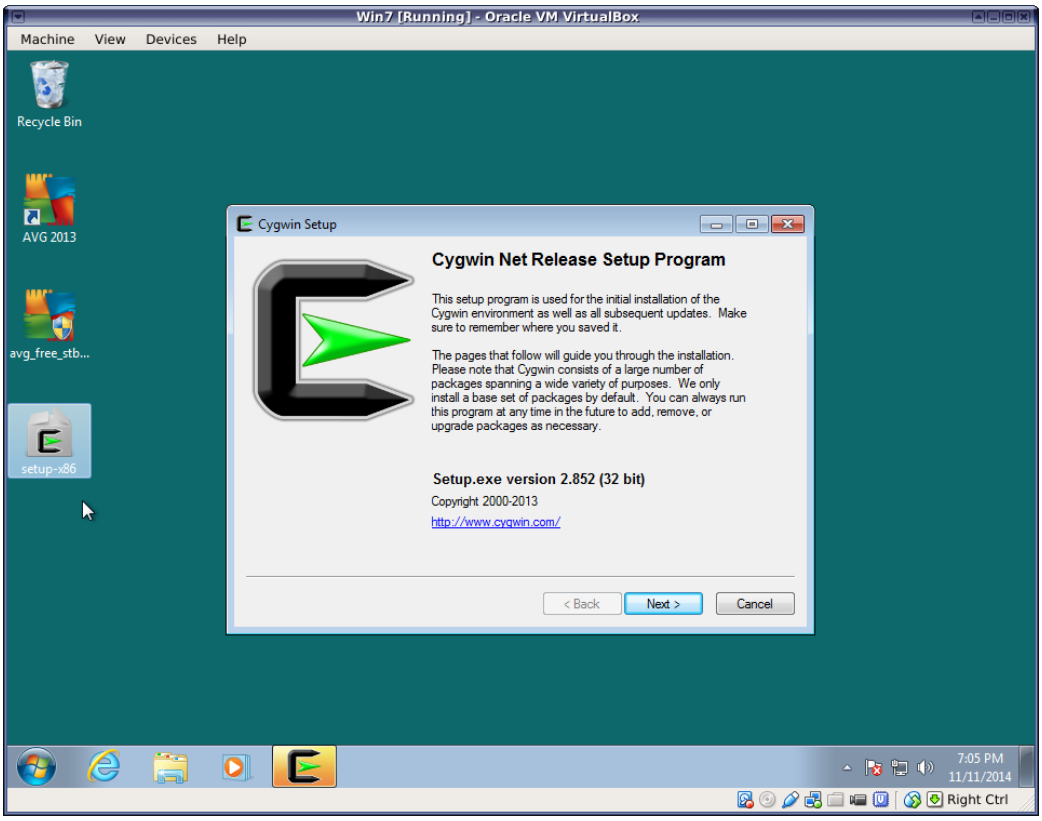
Installing Cygwin is quick and easy:

1. Download **setup-x86.exe** (32-bit Windows) or **setup-x86_64.exe** (64-bit Windows) from http://www.cygwin.com and save a copy on your desktop or some other convenient location. You will need this program to install additional packages in the future.

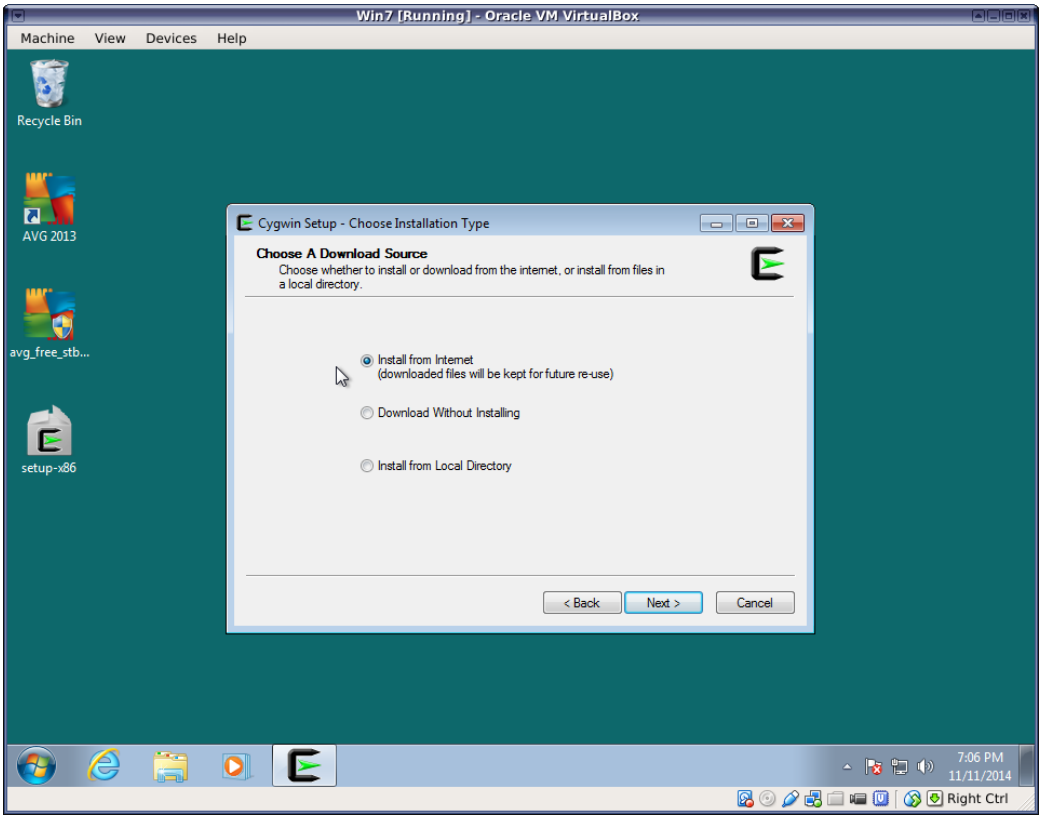


2. Run **setup-x86.exe** or **setup-x86_64.exe** and follow the instructions on the screen.

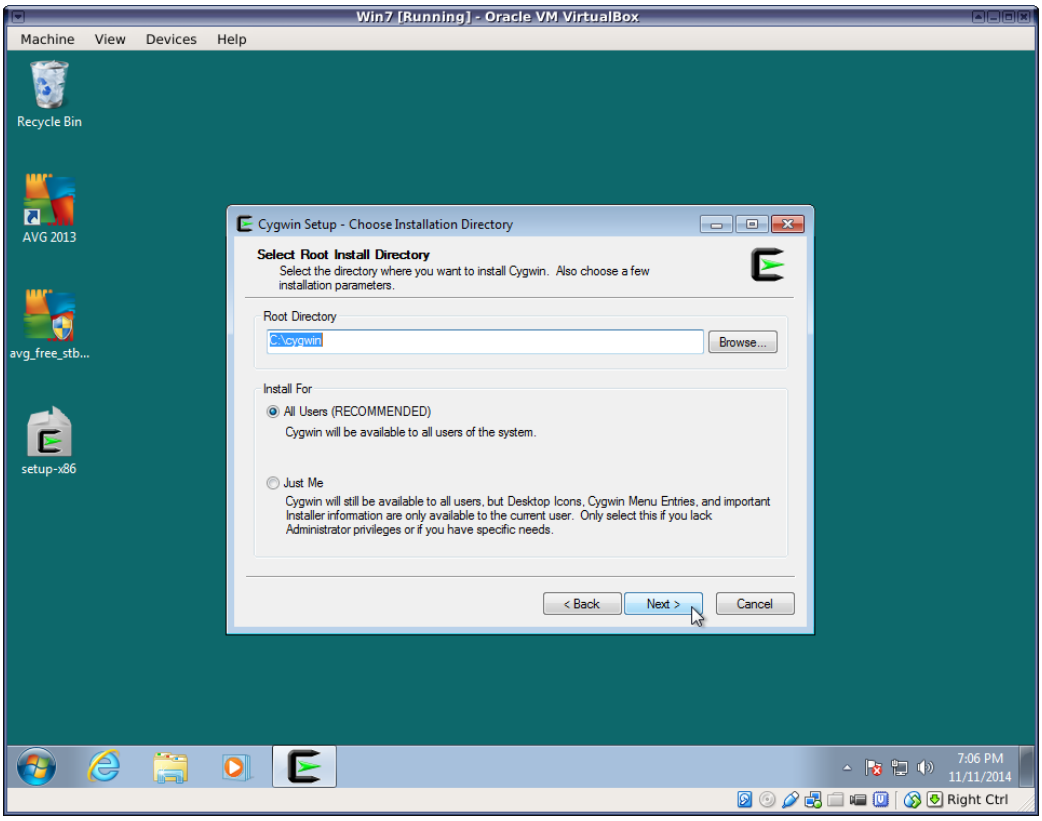
   Unless you know what you're doing, accept the default answers to most questions. Some exceptions are noted below.



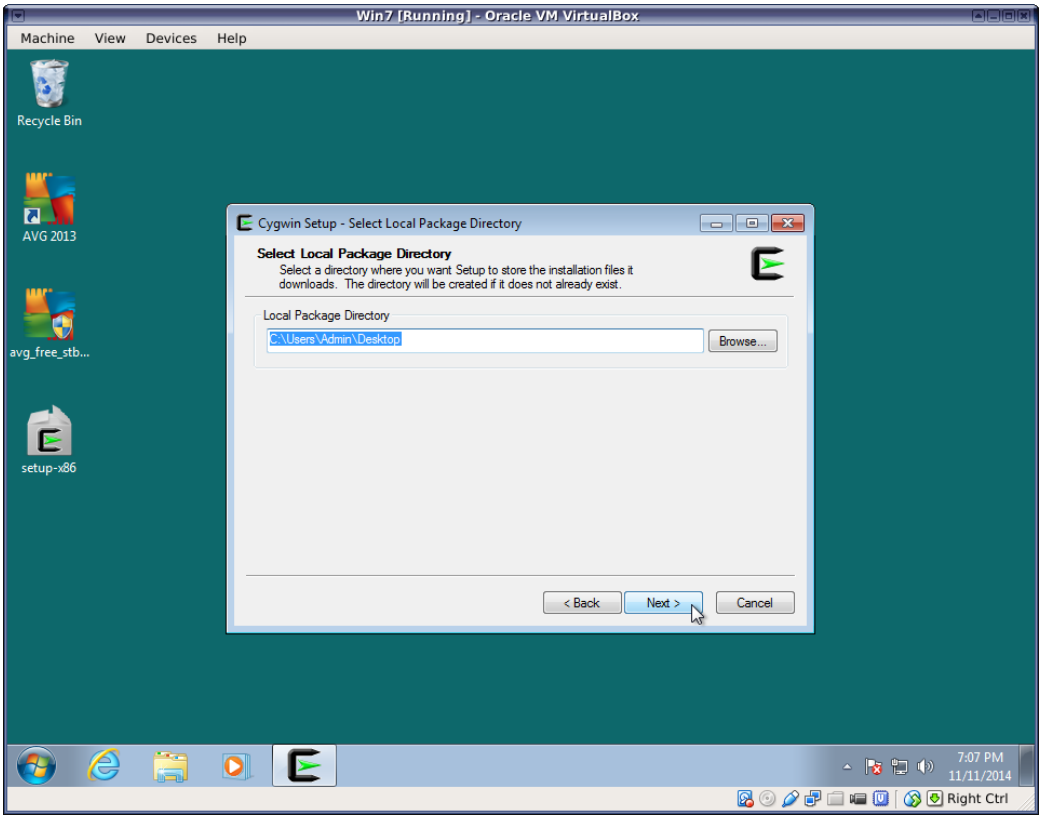
3. Unless you know what you're doing, simply choose "Install from Internet".



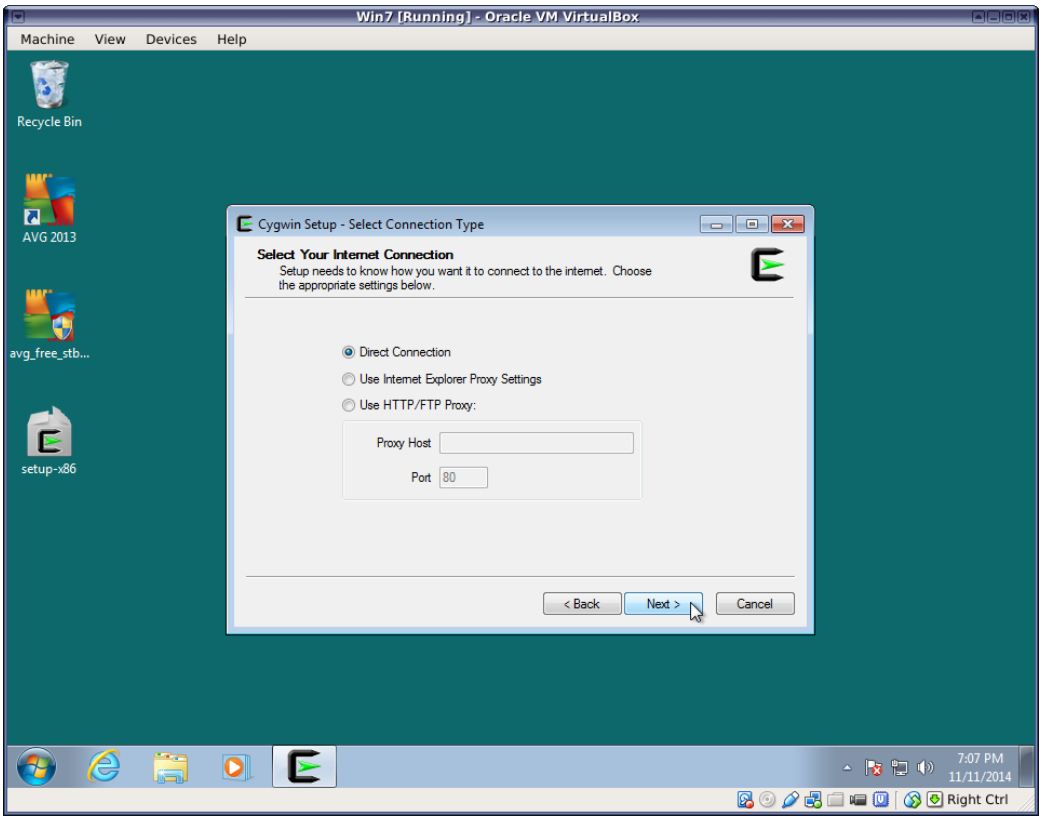
4. Select where you want to install the Cygwin files and whether to install for all users of this Windows machine.



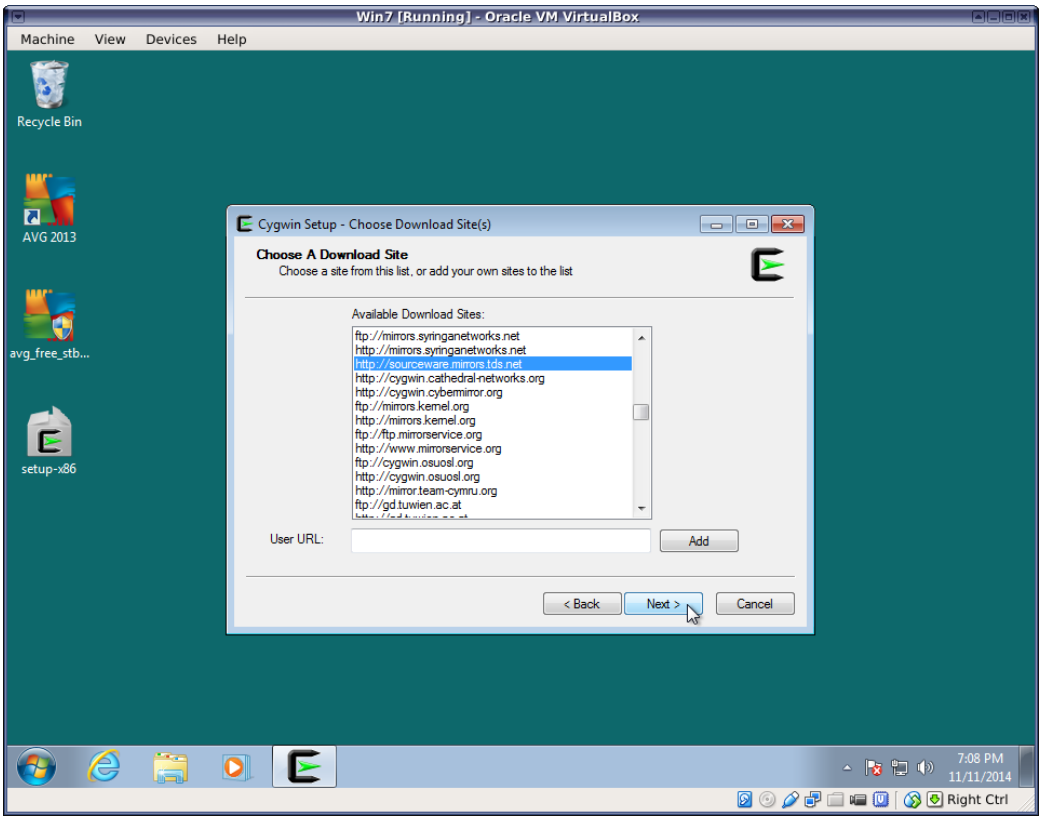
5. Select where to save downloaded packages. Again, the default location should work for most users.



6. Select a network connection type.



7. Select a download site. It is very important here to select a site near you. Choosing a site far away can cause downloads to be incredibly slow. You may have to search the web to determine the location of each URL. This information is unfortunately not presented by the setup utility.



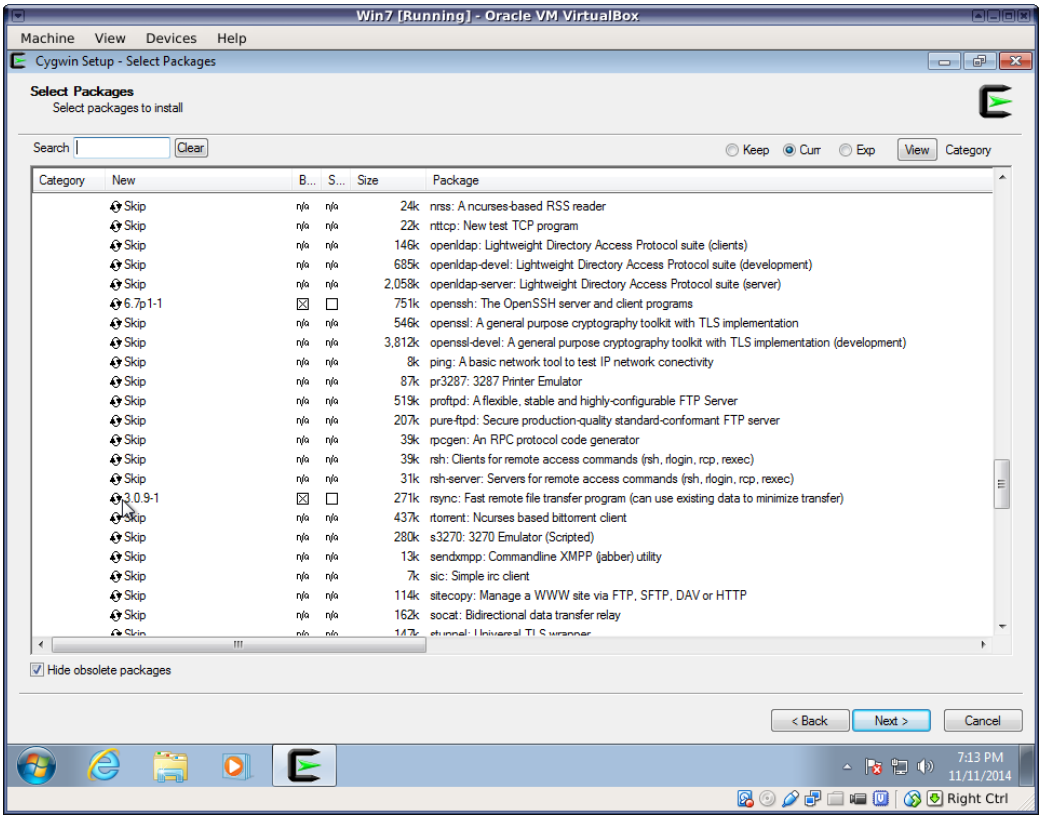
8. When you reach the package selection screen, select at least the following packages in addition to the basic installation:

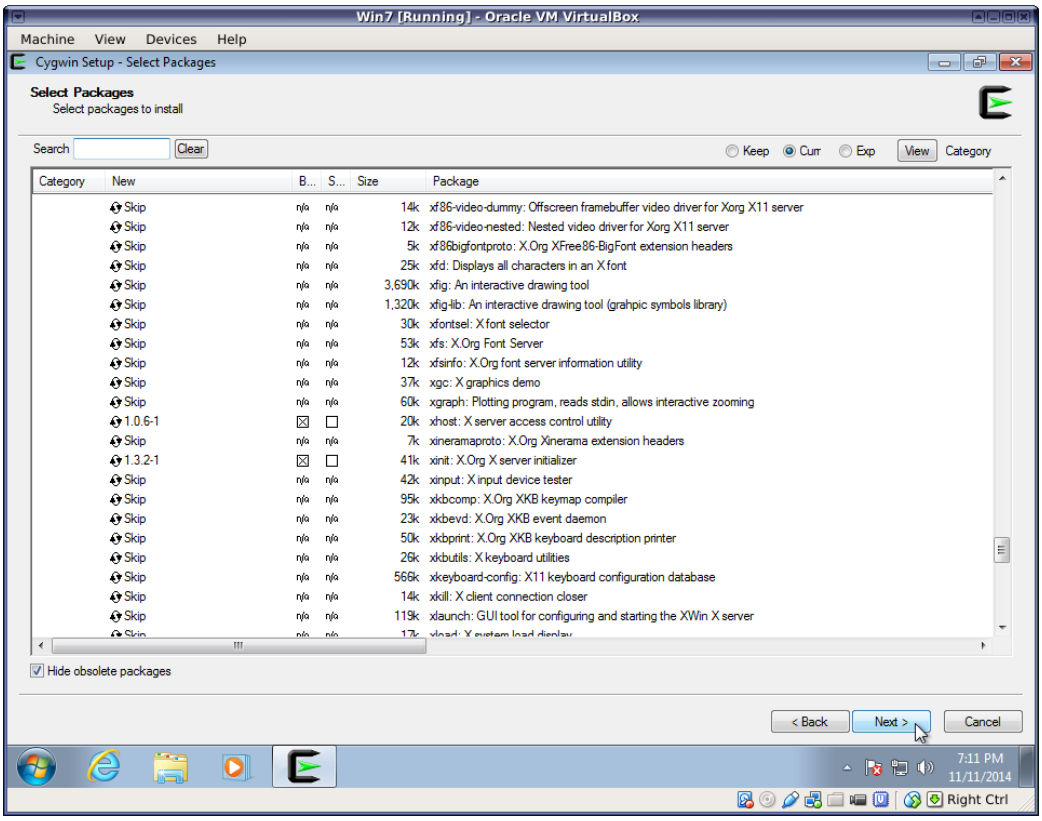
- net/openssh
- net/rsync
- x11/xhost
- x11/xinit

This will install the **ssh** command as well as an X11 server, which will allow you to run graphical Unix programs on your Windows desktop. You may not need graphical capabilities immediately, but they will likely come in handy down the road.

The rsync package is especially useful if you'll be transferring large amounts of data back and forth between your Windows machine and remote servers.

Click on the package categories displayed in order to expand them and see the packages under them.



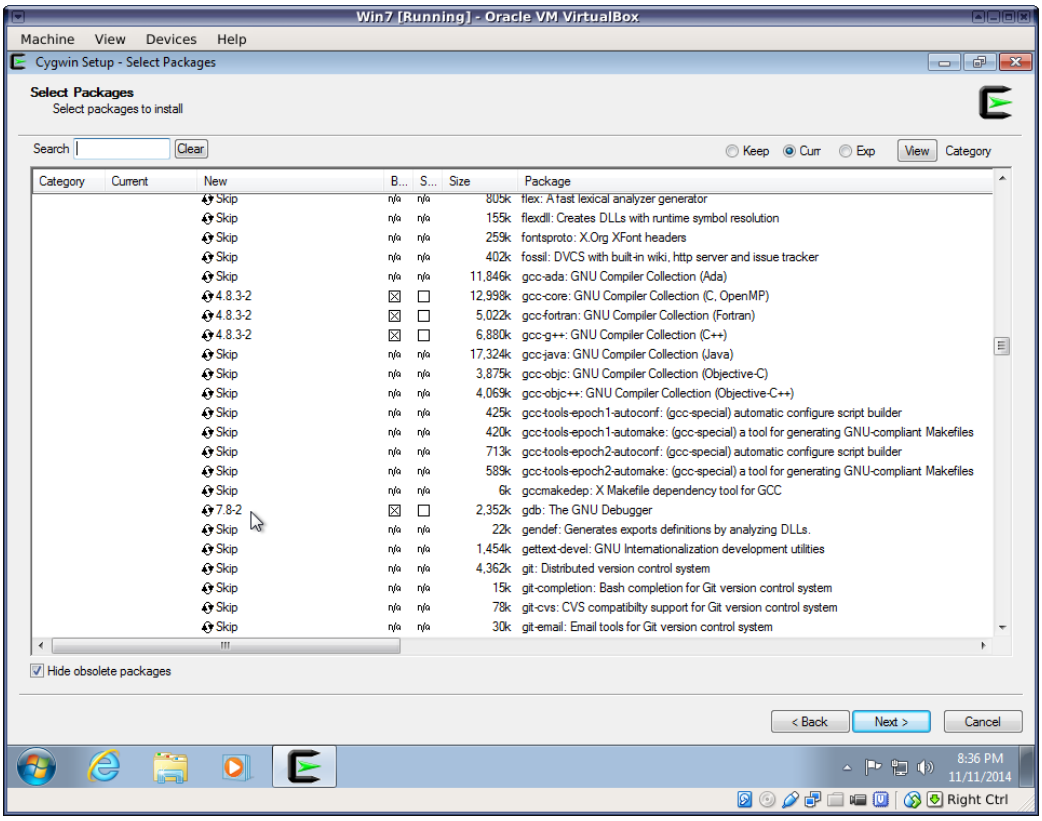


Cygwin can also enable you do do Unix program development on your Windows machine. There are many packages providing Unix development tools such as compilers and editors, as well as libraries. The following is a small sample of common development packages:

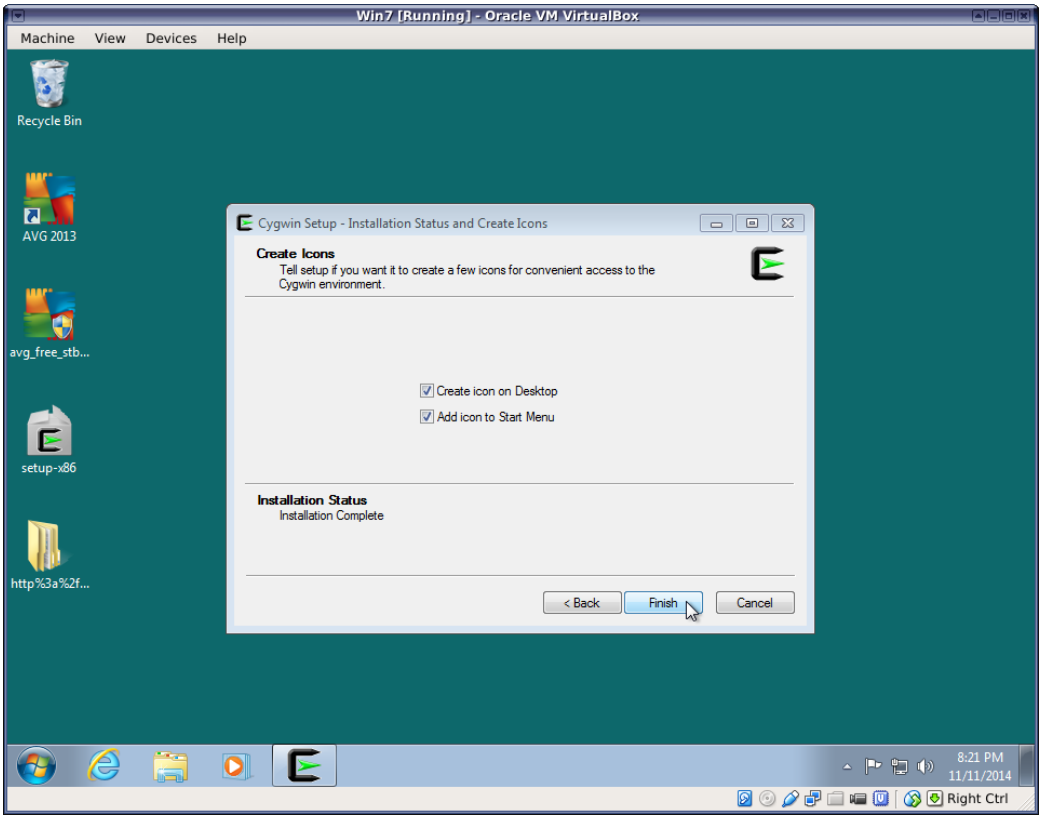
Note

Many of these programs are easier to install and update than their counterparts with a standard Windows interface. By running them under Cygwin, you are also practicing use of the Unix interface, which will make things easy for you when need to run them on a cluster or other Unix host that is more powerful than your PC.

- devel/clang (C/C++/ObjC compiler)
- devel/clang-analyzer (Development and debugging tool)
- devel/gcc-core (GNU Compiler Collection C compiler)
- devel/gcc-g++
- devel/gcc-gfortran
- devel/make (GNU make utility)
- editors/emacs (Text editor)
- editors/gvim (Text editor)
- editors/nano (Text editor)
- libs/openmpi (Distributed parallel programming tools)
- math/libopenblas (Basic Linear Algebra System libraries)
- math/lapack (Linear Algebra PACKage libraries)
- math/octave (Open source linear algebra system compatible with Matlab(r))
- math/R (Open source statistical language)

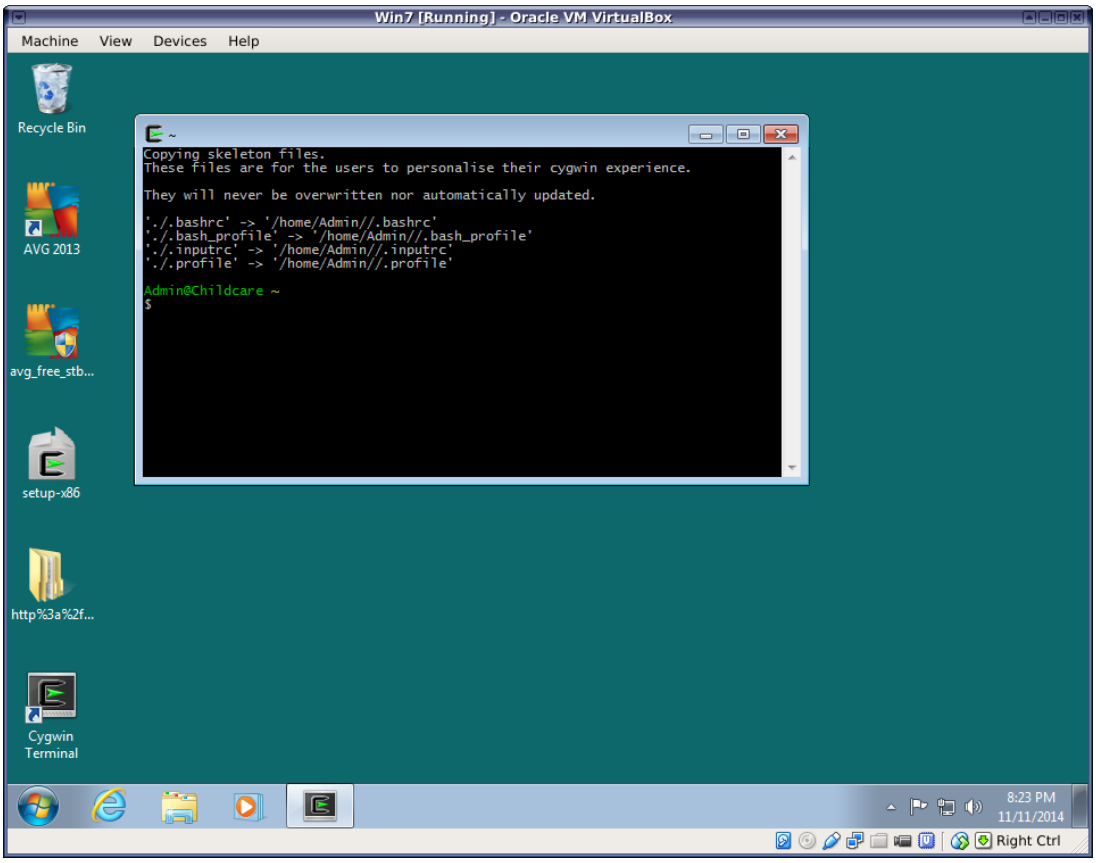


9. Most users will want to accept the default action of adding an icon to their desktop and to the Windows Start menu.

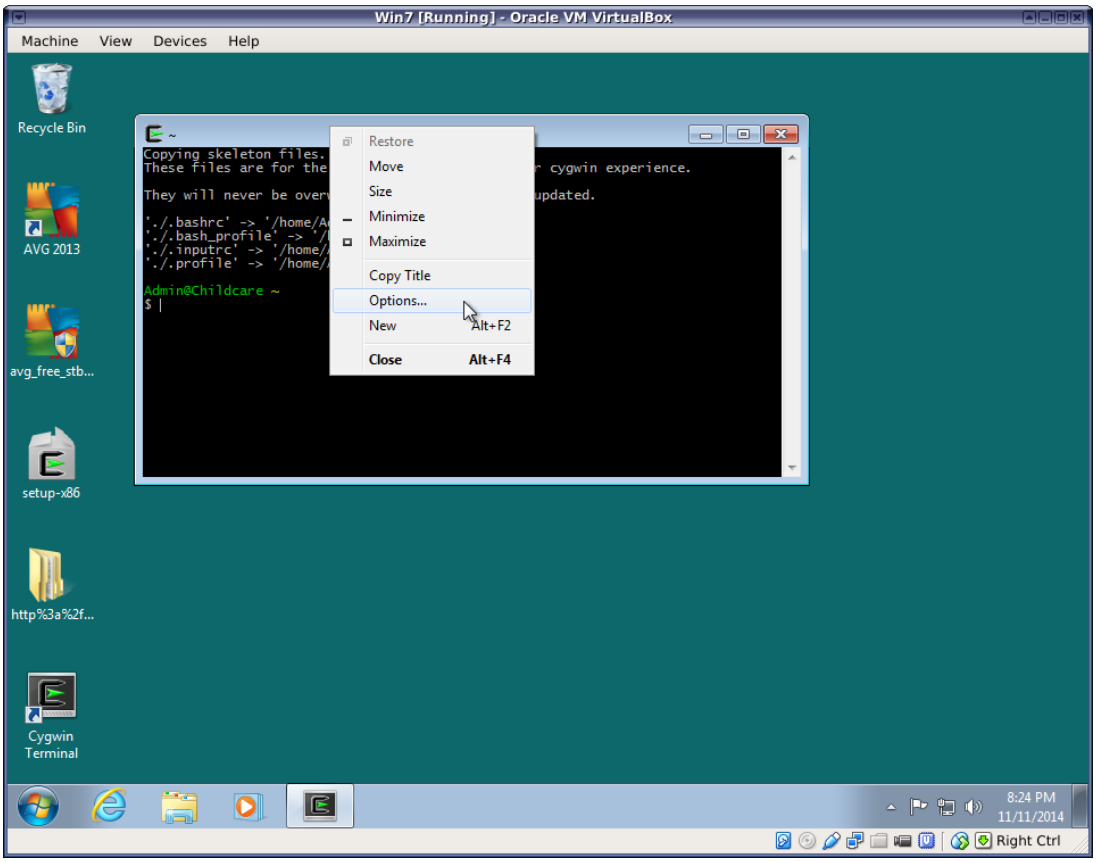


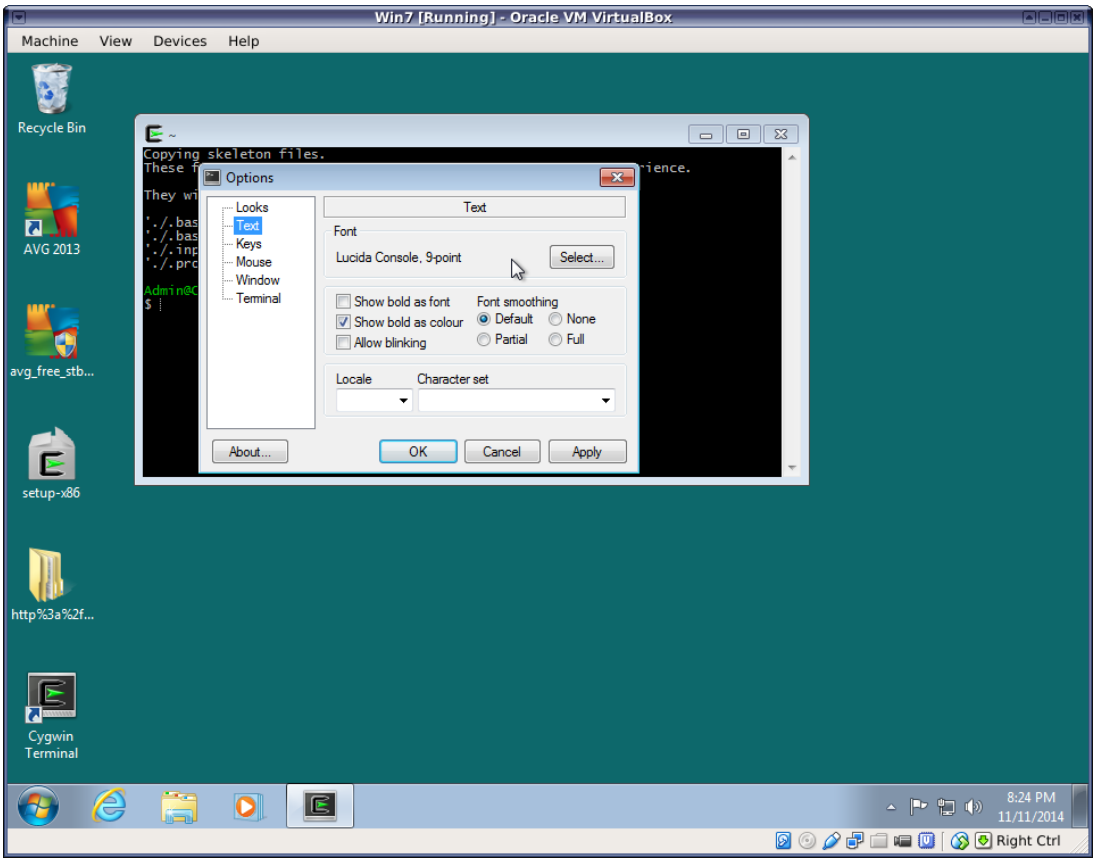
When the installation is complete, you will find Cygwin and Cygwin/X folders in your Windows program menu.

For a basic Terminal emulator, just run the Cygwin terminal:



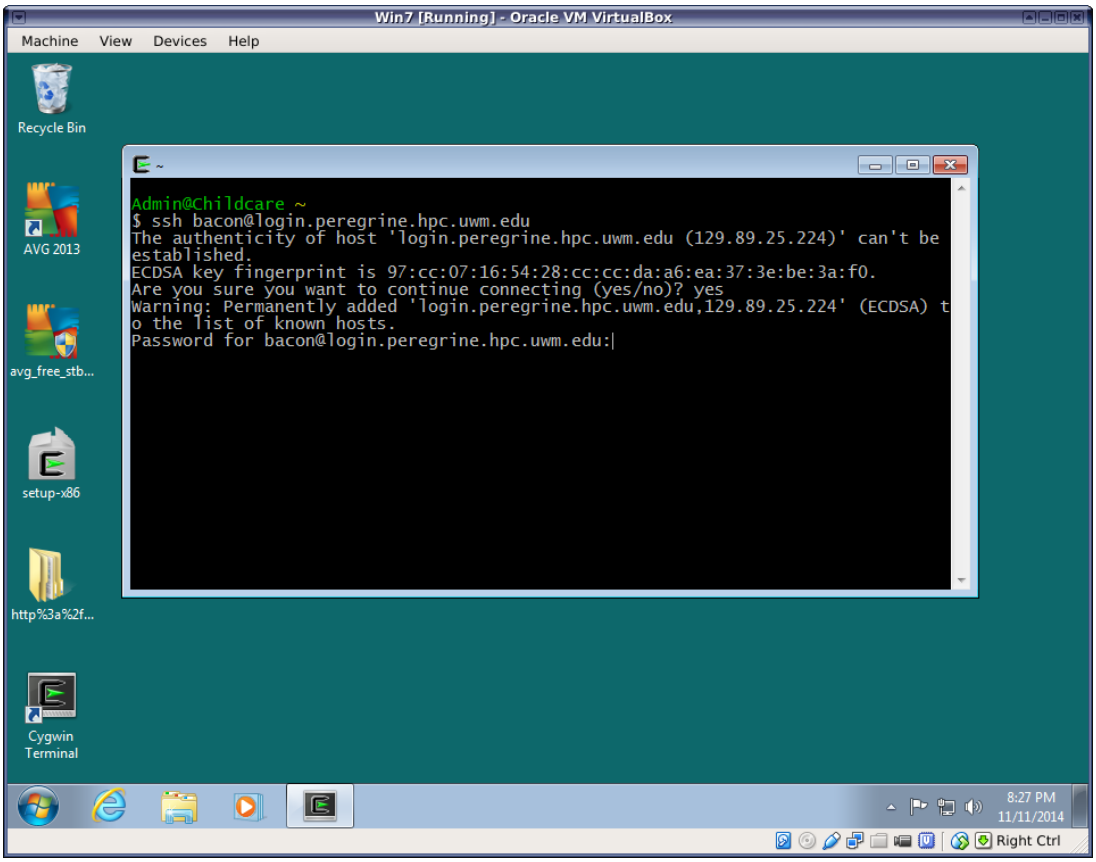
If you'd like to change the font size or colors of the Cygwin terminal emulator, just right-click on the title bar of the window:





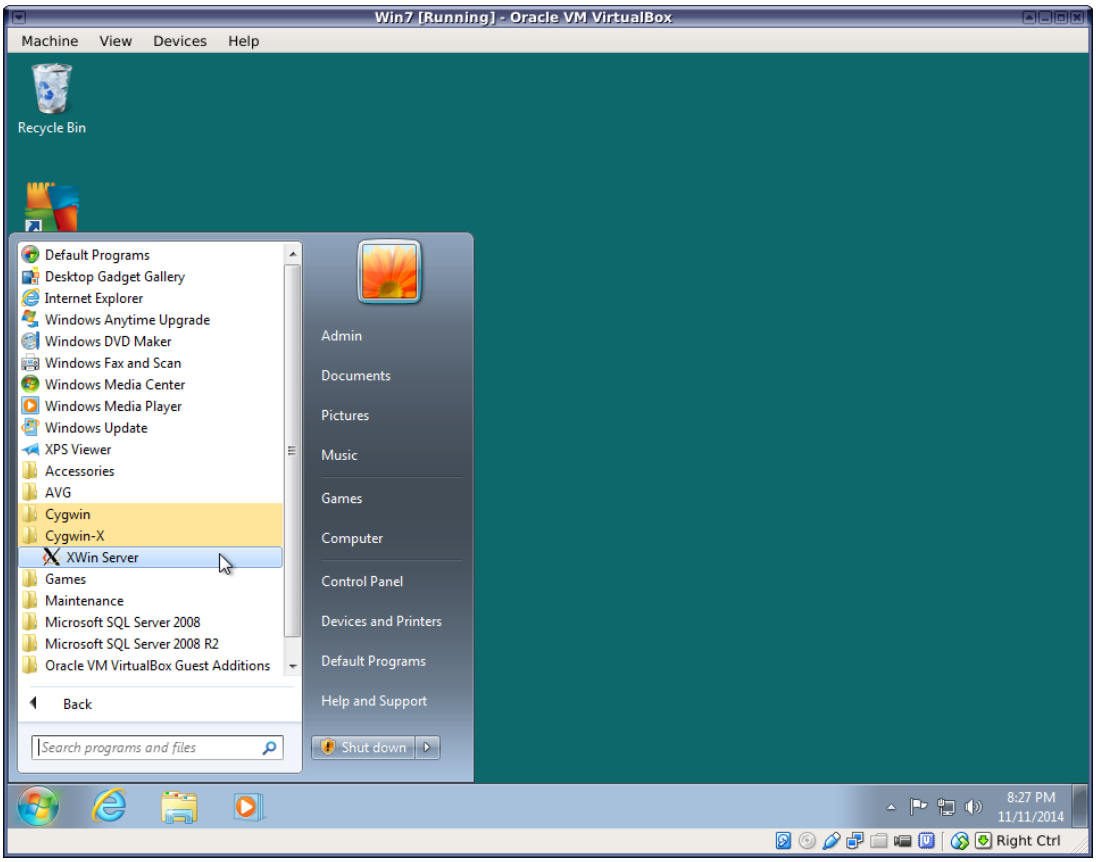
Within the Cygwin terminal window, you are now running a "bash" Unix shell and can run most common Unix commands such as "ls", "pwd", etc.

If you selected the openssh package during the Cygwin installation, you can now remotely log into other Unix machines, such as the clusters, over the network:



Note If you forgot to select the openssh package, just run the Cygwin setup program again. The packages you select when running it again will be added to your current installation.

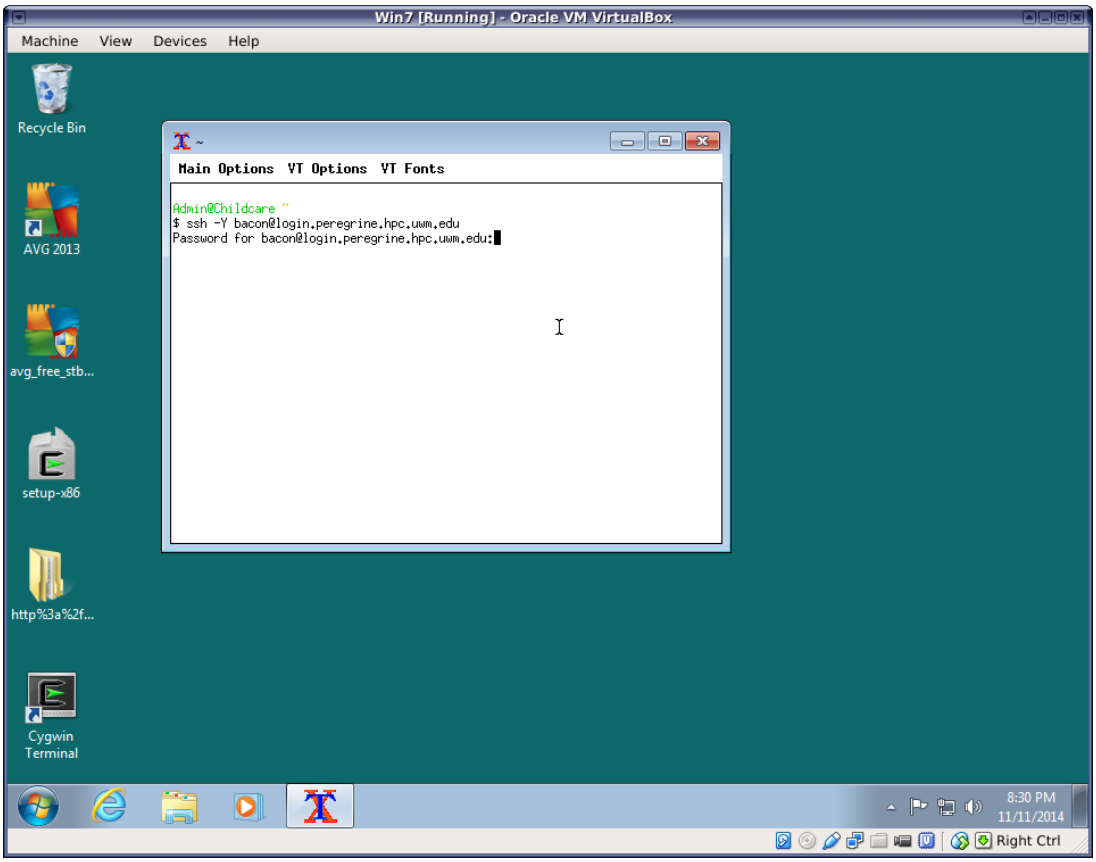
If you want to run Unix graphical applications, either on your Windows machine or on a remote Unix system, run the Cygwin/X application:



Note Doing graphics over a network may require a fast connection. If you are logging in from home or over a wireless connection, you may experience very sluggish rendering of windows from the remote host.

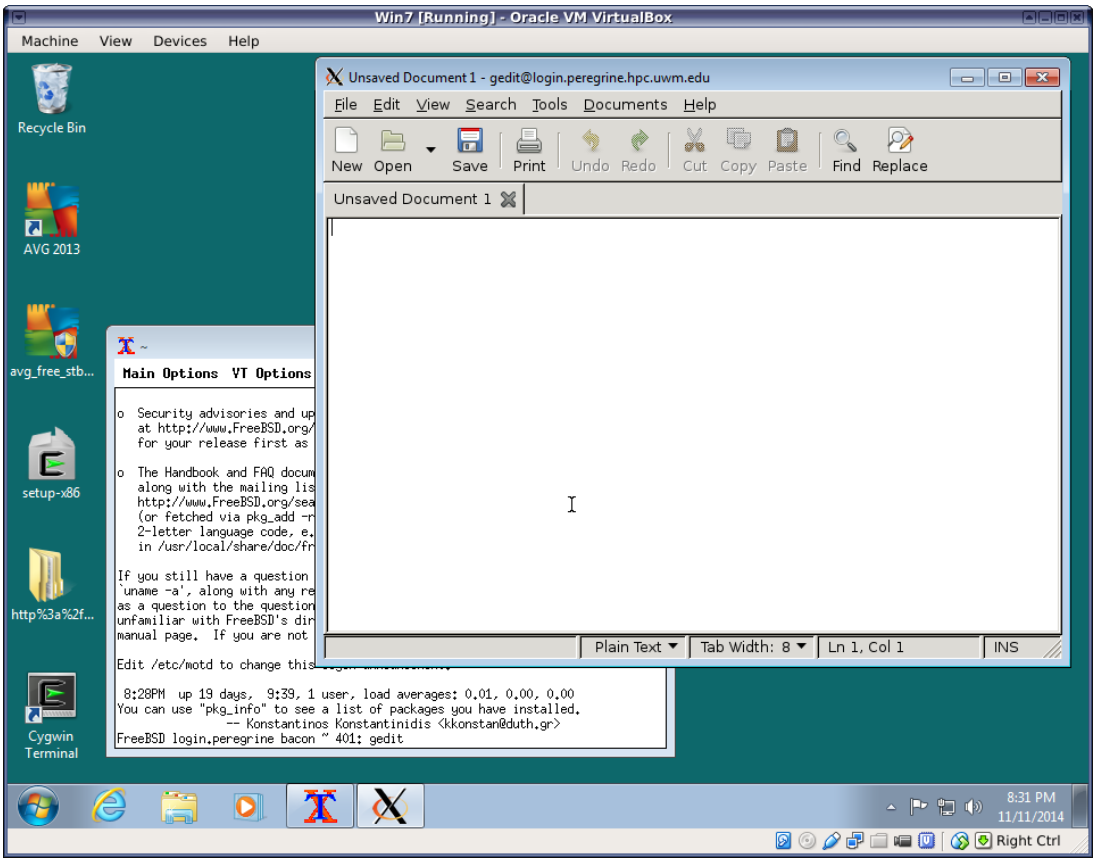
Depending on your Cygwin setup, this might automatically open a terminal emulator called "xterm", which is essentially the same as the standard Cygwin terminal, although it has a different appearance. You can use it to run all the same commands you would in the standard Cygwin terminal, including ssh. You may need to use the -X or -Y flag with ssh to enable some remote graphical programs.

Unlike Cygwin Terminal, the xterm supplied with Cygwin/X is preconfigured to support graphical applications. See Section 1.19.2 for details.



Caution Use of the -X and -Y flags could compromise the security of your Windows system by allowing malicious programs on the remote host to display phony windows on your PC. Use them *only* when logging into a trusted host.

Once you are logged into the remote host from the Cygwin/X xterm, you should be able to run graphical Unix programs.



You can also run graphical applications from the standard Cygwin terminal if you update your start up script. If you are using bash (the Cygwin default shell), add the following line to your .bashrc file:

```
export DISPLAY=unix:0.0
```

You will need to run **source .bashrc** or restart your bash shell after making this change.

If you are using T-shell, the line should read as follows in your .cshrc or .tcshrc:

```
setenv DISPLAY unix:0.0
```

>

Again, Cygwin is not the ideal way to run Unix programs on or from a Windows machine, but it is a very quick and easy way to gain access to a basic Unix environment and many Unix tools. Subsequent sections provide information about other options besides Cygwin for those with more sophisticated needs.

1.5.2 Windows Subsystem for Linux: Another Compatibility Layer

Windows Subsystem for Linux (WSL) is the latest in a chain of Unix compatibility systems provided by Microsoft.

It allows Windows to run a subset of a Linux environment. As of this writing, the user can choose from a few different Linux distributions such as Ubuntu, Debian, SUSE, or Kali.

Differences from Cygwin:

- WSL runs actual Linux binaries (executables), whereas Cygwin allows the user to compile Unix programs into native Windows executables. Programs build under WSL can be run on a compatible Linux distribution and vice-versa. They cannot be run on Windows outside WSL. Programs compiled under Cygwin can in some cases be run under Windows outside Cygwin, but Cygwin cannot run binaries from a real Linux system. Which one you prefer depends on your specific goals. For many people, including most of us who just want to develop or run scientific programs, it makes no difference.

- WSL provides direct access to the native package collection of the chosen Linux distribution. For example, WSL users running the Debian app can install software directly from the Debian project using **apt-get**, just as they would on a real Debian system. The Debian package collection is much larger than Cygwin's, so if Cygwin does not have a package for software you need, WSL might be a good option.
- As of this writing, WSL only supports command-line applications, not Unix graphical programs. It is possible to run graphical programs from WSL, but it requires installing a Windows-based X11 server from another project (such as Cygwin) and then installing the necessary packages within the WSL app. If your goal is to quickly and easily install and run graphical Unix programs, Cygwin is probably a better option.
- Cygwin is an independent open source project, while WSL is a Microsoft product. There are pros and cons to each. Microsoft could terminate support for WSL as it has done with previous Unix compatibility products, if it no longer appears to be in the company's interest to support it. The Cygwin project will only cease if and when there is too little interest from the user community.

1.6 Logging In Remotely

Virtually all Unix systems allow users to log in and run programs over a network from other locations. This feature is intrinsic to Unix systems, and only disabled on certain proprietary or embedded installations. It is possible to use both GUIs and CLIs in this fashion, although GUIs may not work well over slow connections such as a typical home Internet service. Different graphical programs have vastly different video speed demands. Some will work fine over a DSL connection, while others will not work well even over the fastest network.

The command line interface, on the other hand, works comfortably on even the slowest network connections.

Logging into a Unix CLI from a remote location is usually done using *Secure Shell (SSH)*.

Caution Older protocols such as rlogin, rsh, and telnet, should no longer be used due to their lack of security. These protocols transport passwords over the Internet in unencrypted form, so people who manage the gateway computers they pass through can easily read them.

1.6.1 Unix to Unix

If you want to remotely log in from one Unix system to another, you can simply use the **ssh** command from the command line. The general syntax of the **ssh** command is:

```
ssh [flags] login-id@hostname
```

Note If you plan to run graphical programs on the remote Unix system, you may need to include the `-X` (enable X11 forwarding) or `-Y` (enable trusted X11 forwarding) flag in your **ssh** command.

Caution Use `-X` or `-Y` only when connecting to trusted computers, i.e. those managed by you or someone you trust. These options allow the remote system to access your display, which can pose a security risk.

Examples:

```
shell-prompt: ssh joe@unixdev1.hpc.uwm.edu
shell-prompt: ssh joe@login.peregrine.hpc.uwm.edu
```

Note For licensing reasons, **ssh** may not be included in basic Linux installations, but it can be very easily added via the package management system of most Linux distributions.

Once logged in, you can easily open additional terminals from the command-line if you know the name of the terminal emulator. Simply type the name of the terminal emulator, followed by an '&' to put it in the background. (See Section 1.18.3 for a full explanation of background jobs.) Some common terminal emulators are xterm, rxvt, gnome-terminal, xfce-terminal, konsole, and Terminal.

```
shell-prompt: xfce-terminal &
```

1.6.2 Windows to Unix

If you're connecting to a Unix system from a Windows system, you will need to install some additional software.

Cygwin

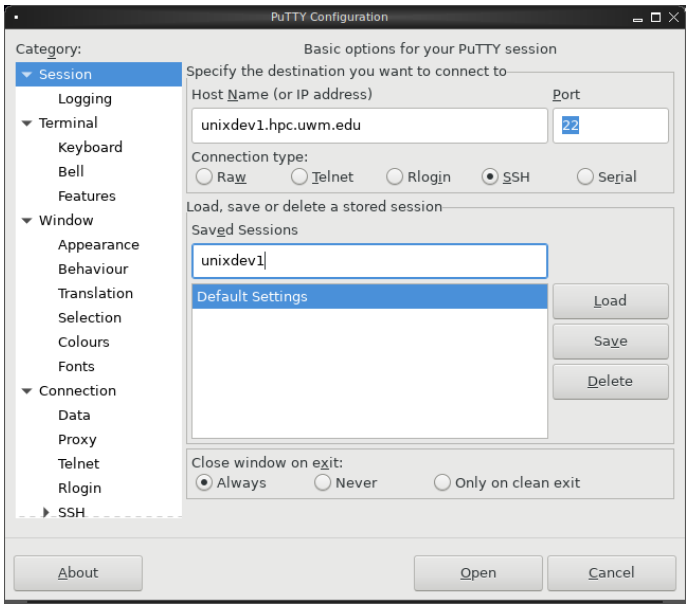
The Cygwin Unix-compatibility system is free, quick and easy to install, and equips a Windows computer with most common Unix commands, including a Unix-style Terminal emulator. Once Cygwin is installed, you can open a Cygwin terminal on your Windows desktop and use the **ssh** command as shown above.

The Cygwin installation is very quick and easy and is described in Section 1.5.1.

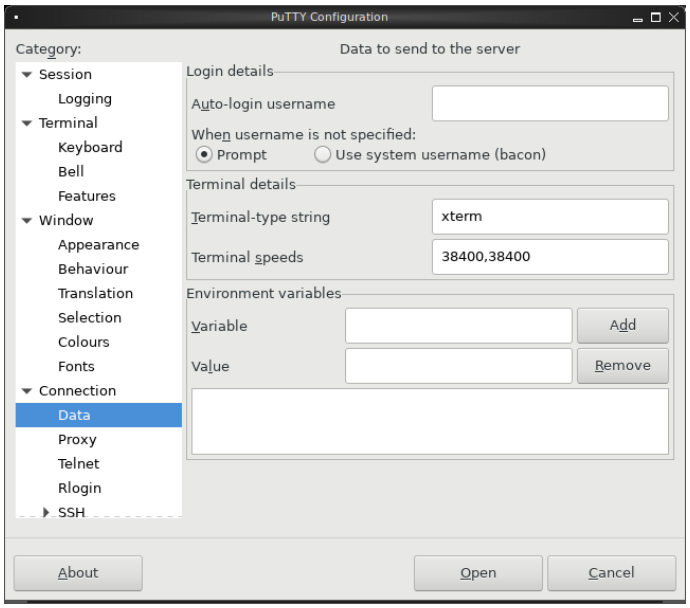
PuTTY

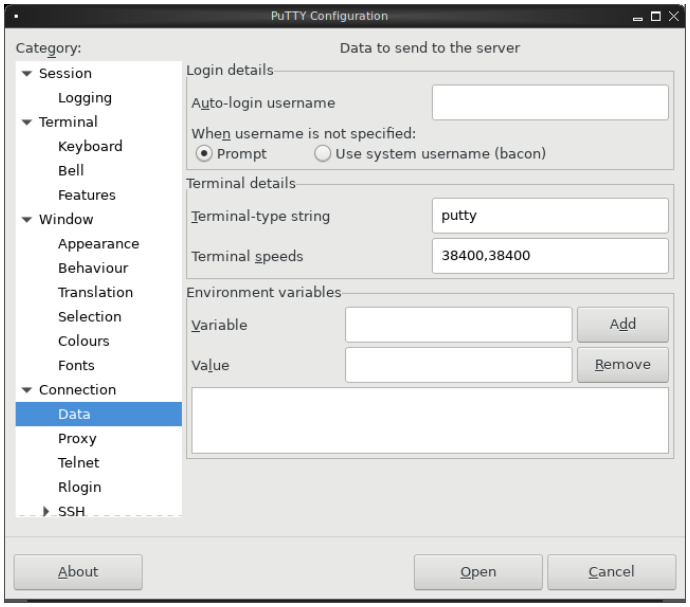
A more limited method for remotely accessing Unix systems is to install a stand-alone terminal emulator, such as PuTTY, http://www.chiark.greenend.org.uk/~sgtatham/putty/. PuTTY has a built-in **ssh** client, and a graphical dialog box for connecting to a remote machine. To connect via **ssh**, simply select the **ssh** radio button, enter the hostname of the computer you want to connect to, and click the Open button.

Connections to specific machines can be saved. First, enter the host name in the "Host Name" box and a name of your choice in the "Saved Sessions" box:

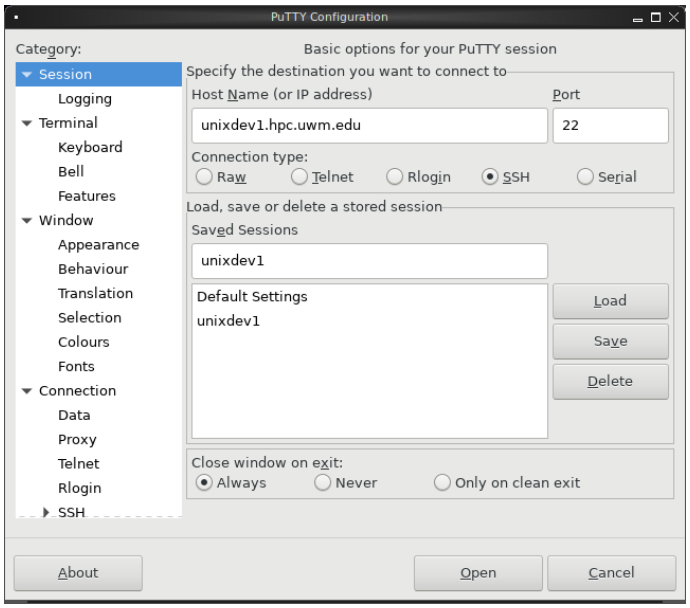


In order for terminal-based programs to function properly, they must know what type of terminal or terminal emulator you are using. Most terminal emulators are based on the "xterm" terminal type, and programs will mostly work if you tell the remote system you are using an xterm. Some special keys and graphic characters may not work properly, though. For best results with PuTTY, go to the "Data" section under "Connection" and change the terminal type that PuTTY reports from "xterm" to "putty":

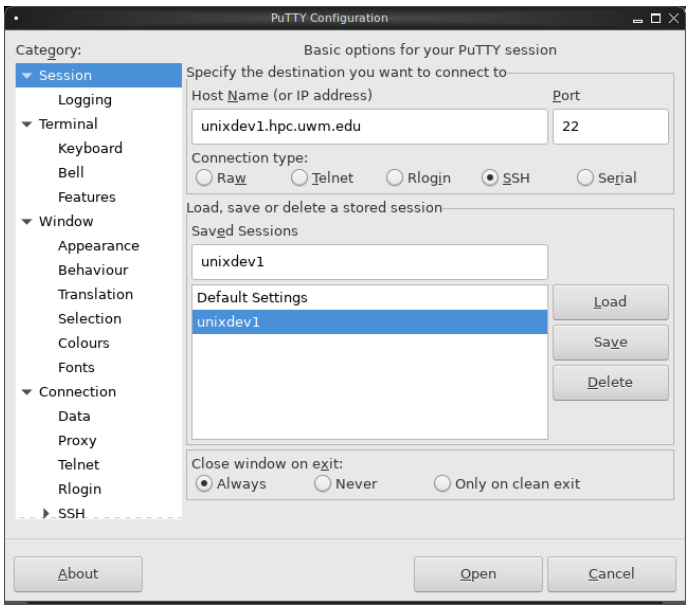




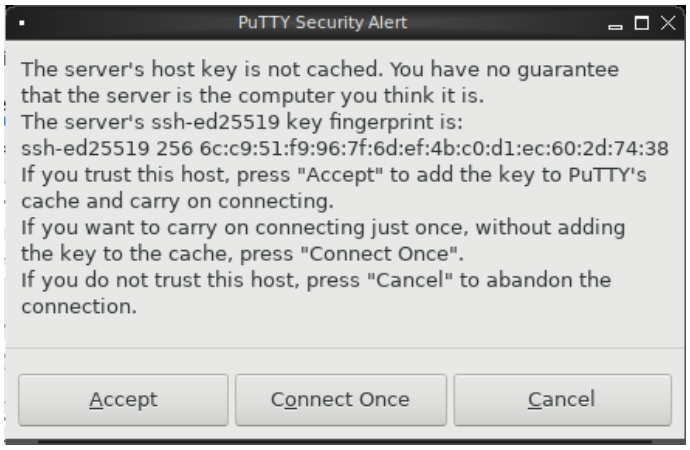
Then go back to the "Session" screen and click "Save" to add this connection to the saved sessions.



Once you've saved the session, you can click on the name once and then click "Open", or just double click on the name.



The first time you connect to a system using SSH protocol, you will be asked whether you really trust the host to which you are connecting. If you trust it, click "Yes" or "Accept" or "Connect Once". If not, then don't connect to the host.



1.6.3 Terminal Types

In some cases, you may be asked to specify a terminal type when you log in:

```
TERM=(unknown)
```

Terminal features such as cursor movement and color changes are triggered by sending special codes (characters or character combinations) to the terminal. Pressing keys on the terminal sends codes from the terminal to the computer.

Different types of terminals use different key and screen control codes. PuTTY and most other terminal emulators emulate an "xterm" terminal, so if asked, just type the string "xterm" (without the quotes).

If you fail to set the terminal type, some programs such as text editors will not function. They may garble the screen and fail to recognize special keys such as arrows, page-up, etc. Programs such as **ls** that simple output a line and then go to the next will generally work fine even if TERM is not set.

You can set the terminal type after logging in, but the methods for doing this vary according to which shell you use, so you may just want to log out and remember to set the terminal type when you log back in.

Practice Break

Remotely log into another Unix system using the **ssh** command or PuTTY. Then try starting the **vi** editor:

```
shell-prompt: ls
shell-prompt: ls /
shell-prompt: ls -al
shell-prompt: mkdir -p Data/IRC
shell-prompt: cd Data/IRC
shell-prompt: nano sample.txt

Type in some text, then save the file (press Ctrl+o), and exit nano
(press Ctrl+x).

shell-prompt: ls
shell-prompt: cat sample.txt
shell-prompt: wc sample.txt
shell-prompt: whoami
shell-prompt: hostname
shell-prompt: uname
shell-prompt: date
shell-prompt: cal
shell-prompt: cal nov 2018
shell-prompt: bc -l
scale=50
sqrt(2)
8^2
2^8
a=1
b=2
c=1
(-b+sqrt(b^2-4*a*c))/2*a
2*a
quit

shell-prompt: w
shell-prompt: ls /bin
```

1.6.4 Self-test

1. What extra software must you install on Unix systems to allow logging into another Unix system over a network? Explain.

2. What extra software must you install on Unix systems to enable the use of graphical programs over a network? Explain.
3. What is **ssh**? How does it differ from **rsh** and **telnet**?
4. Can Windows users run **ssh**? Explain.
5. Can Windows users run graphical programs on a remote Unix system? Explain.
6. What is the purpose of the TERM environment variable? What will happen if it is not set correctly?

1.7 Unix Command Basics

A Unix command is built from a command name and optionally one or more command line *arguments*. Arguments can be either *flags* or data.

```
ls -a -l /etc /var
^^ ^^^^^ ^^^^^^^^^
|   |     |
|   |     Data Arguments
|   Flags
Command name
```

- The *command name* identifies the program to run. For example, the **ls** command names a program that lists the contents of a directory.
- Most commands have optional *flags* (sometimes called *options*) that control the general behavior of the command. By convention, flags begin with a '-' character, just to help the reader distinguish between flags and arguments.

  Note Unix systems do not enforce this, but very few commands violate it. Unlike voluntary taxation, voluntary environmental regulations, or voluntary speed limits, the voluntary Unix conventions garner a very high level of conformance. Unix programmers tend to understand the benefits of conventions and don't have to be forced to follow them.

  The flags in the example above have the following meaning:

  `-a`: tells **ls** to show "hidden" files (files whose names begin with '.', which ls would not normally list).

  `-l`: tells **ls** to do a "long listing", which is to show lots of information about each file and directory instead of just the name.

  Single-letter flags can usually be combined, e.g. `-a -l` can be abbreviated as `-al`.

  Most newer Unix commands also support long flag names, mainly to improve readability of commands used in scripts. For example, in the Unix **zip** command, `-C` and `--preserve-case` are synonymous.

- Many commands also accept one or more data arguments, which provide input data to the command, or instruct it where to send output. Such arguments may be the actual input or they may be the names of files or directories that contain input or receive output. The `/etc` and `/var` arguments above are directories to be listed by **ls**. If no data arguments are given to **ls**, it lists the current working directory (described in Section 1.9).

For many Unix commands, the flags must come before the data arguments. A few commands require that certain flags appear in a specific order. Some commands allow flags and data arguments to appear in any order. Unix systems do not enforce any rules regarding arguments. How they behave is entirely up to the creator of the command. However, the vast majority of commands follow conventions, so there is a great deal of consistency in Unix command syntax.

The components of a Unix command are separated by white space (space or tab characters). Hence, if an argument contains any white space, it must be enclosed in quotes (single or double) so that it will not be interpreted as multiple separate arguments.

Example 1.1 White space in an Argument

Suppose you have a directory called `My Programs`, and you want to see what's in it. You might try the following:

```
shell-prompt: ls My Programs
```

The above command fails, because "My" and "Programs" are interpreted as two separate arguments. In fact, the **ls** command will look for two separate directories called "My" and "Programs". In this case, we must use quotes to bind the parts of the directory name together into a single argument. Either single or double quotes are accepted by all common Unix shells. The difference between single and double quotes is covered in Chapter 2.

```
shell-prompt: ls 'My Programs'
shell-prompt: ls "My Programs"
```

As an alternative to using quotes, we can *escape* the space by preceding it with a backslash ('\') character. This will save one keystroke if there is only one character to be escaped in the text.

```
shell-prompt: ls My\ Programs
```

Practice Break

Try the following commands:

```
shell-prompt: ls
shell-prompt: ls -al
shell-prompt: ls /etc
shell-prompt: ls -al /etc
shell-prompt: mkdir 'My Programs'
shell-prompt: ls My Programs
shell-prompt: ls "My Programs"
```

1.7.1 Self-test

1. What are the parts of a Unix command? What separates them?
2. What are command line flags?
3. What are data arguments?
4. What rules do Unix systems enforce regarding the placement of arguments?
5. How do we specify an argument that contains white space?

1.8 Basic Shell Tools

1.8.1 Common Unix Shells

There are many different shells available for Unix systems. This might sound daunting if you're new to Unix, but fortunately, like most Unix tools, all the common shells adhere to certain standards.

All of the common shells are derived from one of two early shells:

- Bourne shell (sh) is the de facto basic shell on all Unix systems, and is derived from the original Unix shell developed at AT&T.
- C shell (csh) offers mostly the same features as Bourne shell, but the two differ in the syntax of their scripting languages, which are discussed in Chapter 2. The C shell syntax is designed to be more intuitive and similar to the C language.

Most Unix commands are exactly the same regardless of which shell you are using. Differences will only become apparent when using more advanced command features or writing shell scripts, both of which we will cover later.

Common shells derived from Bourne shell include the following:

- Almquist shell (ash), used as the Bourne shell on many BSD systems.
- Korn shell (ksh), an extended Bourne shell with many added features for user-friendliness.
- Bourne again shell (bash) another extended Bourne shell from the GNU project with many added features for user-friendliness. Used as the Bourne shell on many Linux systems.
- Debian Almquist shell (dash), a reincarnation of ash which is used as the Bourne shell on Debian based Linux systems.

Common shells derived from C shell include the following:

- T shell (tcsh), and extended C shell with many added features for user-friendliness.
- Hamilton C shell, an extended C shell used primarily on Microsoft Windows.

Unix systems differ in which shells are included in the base installation, but most shells can be easily added to any Unix system using the system's package manager.

1.8.2 Command History

Most shells remember a configurable number of recent commands. This command history is saved to disk, so that you can still recall this session's commands next time you log in.

The exact mechanisms for recalling those commands varies from shell to shell, but some of the features common to all shells are described below.

Most modern shells support scrolling through recent commands by using the up-arrow and down-arrow keys. Among the popular shells, only very early shells like Bourne shell (sh) and C shell (csh) lack this ability.

Note This feature may not work if your TERM variable is not set properly, since the arrow keys send magic sequences that may differ among terminal types.

The **history** command lists the commands that the shell currently has in memory.

```
shell-prompt: history
```

A command consisting of an exclamation point (!) followed by any character string causes the shell to search for the most recently executed command that began with that string. This is particularly useful when you want to repeat a complicated command.

```
shell-prompt: find Programs -name '*.o' -exec rm -i '{}' \;
shell-prompt: !find
```

An exclamation point followed by a number runs the command with that history index:

```
shell-prompt: history
   385  13;42   more output.txt
   386  13:54   ls
   387  13:54   cat /etc/hosts
shell-prompt: !386
ls
Avi-admin/            Materials-Studio/     iperf-bsd
Backup@               New-cluster/          notes
Books/                Peregrine-admin/      octave-workspace
```

Tantalizing sneak preview: We can check the history for a particular pattern such as "find" as follows:

```
shell-prompt: history | grep find
```

More on the "| find" in Section 1.17.1.

1.8.3 Auto-completion

In most Unix shells, you need only type enough of a command or argument filename to uniquely identify it. At that point, pressing the TAB key will automatically fill in the rest for you. Try the following:

```
shell-prompt: touch sample.txt
shell-prompt: cat sam<Press the TAB key now>
```

If there are other files in your directory that begin with "sam", you may need to type a few additional characters before the TAB, like 'p' and 'l' before auto-completion will work.

1.8.4 Command-line Editing

Modern shells allow extensive editing of the command currently being entered. The key bindings for different editing features depend on the shell you are using and the current settings. Some shells offer a selection of different key bindings that correspond to Unix editors such as **vi** or **Emacs**.

See the documentation for your shell for full details. Below are some example default key bindings for shells such as **bash** and **tcsh**.

Key	**Action**
Left arrow	Move left
Right arrow	Move right
Ctrl+a	Beginning of line
Ctrl+e	End of line
Backspace or Ctrl+h	Delete left
Ctrl+d	Delete current

Table 1.3: Default Key Bindings in some Shells

1.8.5 Globbing (File Specifications)

There is often a need to specify a large number of files as command line arguments. Typing all of them would be tedious, so Unix shells provide a mechanism called *globbing* that allows short, simple patterns to match many file names.

These patterns are build from literal text and/or the special symbols called *wild cards* in Table 1.4.

Symbol	**Matches**
*	Any sequence of characters (including none) except a '.' in the first character of the filename.
?	Any single character, except a '.' in the first character of the filename.
[string]	Any character in string
[c1-c2]	Any character from c1 to c2, inclusive
{thing1,thing2}	Thing1 or thing2

Table 1.4: Globbing Symbols

Normally, the shell handles these special characters, expanding globbing patterns to a list of matching file names *before* the command is executed.

If you want an argument containing special globbing characters to be sent to a command in its raw form, it must be enclosed in quotes, or each special character must be escaped (preceded by a backslash, \).

Certain commands, such as **find** need to receive the pattern as an argument and attempt to do the matching themselves rather than have it done for them by the shell. Therefore, patterns to be used by the **find** command must be enclosed in quotes.

The exemption for a leading '.' prevents accidental matching of hidden files.

```
shell-prompt: ls *.txt     # Lists all files ending in ".txt"
shell-prompt: ls "*.txt"   # Fails, unless there is a file called '*.txt'
shell-prompt: ls '*.txt'   # Fails, unless there is a file called '*.txt'
shell-prompt: ls .*.txt    # Lists hidden files ending in ".txt"
shell-prompt: ls [A-Za-z]* # Lists all files and directories
         # whose name begins with a letter
shell-prompt: find . -name *.txt    # Fails
shell-prompt: find . -name '*.txt'  # List .txt files in all subdirectories
shell-prompt: ls *.{c,c++,f90}
```

1.8.6 Self-test

1. Which shells support modern interactive features such as scrolling through previous commands and command-line editing?
2. Show the simplest command to accomplish each of the following:
   (a) Show a list of recently executed commands.
   (b) Re-execute the most recent command that began with "ls".
3. Show the simplest command to accomplish each of the following:
   (a) Move all the files whose names end with ".c" from the current directory to the directory `./Prog1`.
   (b) Remove all the files whose names end with ".o".
   (c) List the contents of all files/directories in the current working directory whose names begin with a '.' followed by a capital letter or lower case letter, and end with a digit.

1.9 Processes

Before You Begin It is assumed the reader knows what Unix is. If not, please read Section 1.2 before proceeding.

A *process* in Unix terminology is the execution of a program.

Unix is a multitasking system, which means that many processes can be running at any given moment, i.e. there can be many active processes.

When you log in, the system creates a new process to run your shell program.

When you run a program (a command) from the shell, the shell creates a new process to run the program. Hence, you now have two processes running: the shell process and the command's process.

The process created by the shell to run your command is a *child* of the shell process.

Naturally, the shell process is called the *parent* of the command process.

Each process is uniquely identified by an integer serial number called the *process ID*, or *PID*.

Unix systems also keep track of each process's status and resource usage, such as memory, CPU time, etc. Information about your currently running processes can be easily viewed using the **ps** (process status) command:

```
shell-prompt: ps
 PID TTY          TIME CMD
7147 ttys000    0:00.14 -tcsh
7438 ttys000    0:01.13 ape notes.dbk unix.dbk
7736 ttys001    0:00.13 -tcsh
```

Practice Break
Run the **ps** command. What processes do you have running?

```
shell-prompt: ps
shell-prompt: ps -ax
```

Another useful tool is the **top** command, which monitors all processes in a system and displays system statistics and the top (most active) processes every few seconds. Note that since **top** is a full-terminal command, it will not function properly unless the TERM environment variable is set correctly.

Practice Break
Run the **top** command. What processes are using the most CPU time? Type 'q' to quit **top**.

```
shell-prompt: top
```

1.9.1 Self-test

1. What is a process?
2. When are processes created?
3. How does the Unix system distinguish between all the processes running at a given moment?
4. What command(s) will show you the processes that are currently running?

1.10 The Unix File system

1.10.1 Unix Files

A Unix *file* is simply a sequence of bytes (8-bit values) stored on a disk and given a unique name. The bytes in a file may be printable characters such as letters, digits, punctuation symbols, invisible *control characters* (which cause a printer or terminal to perform actions such as backspacing or scrolling), or other non-character data.

Text vs Binary Files

Files are often classified as either *text* or *binary* files. All of the bytes in a text file are interpreted as characters by the programs that read or write the file, while binary files may contain both character and non-character data.

Note that the Unix standard makes no distinction between one file and another based on their contents. Only individual programs care what is inside a file.

Practice Break
Try the following commands:

```
shell-prompt: cat .profile
```

What do you see? The `.profile` file is a text file, and **cat** is used here to echo it to the screen.
Now try the following:

```
shell-prompt: cat /bin/ls
```

What do you see? The file `/bin/ls` is not a text file. It contains binary program code, not characters. The **cat** command assumes that the file is a text file displays each character terminal. Binary files show up as a lot of garbage, and may even knock your terminal out of whack. If this happens, run the **reset** command to restore your terminal to its original state. In the rare case that **reset** does not fix the terminal, you can try running an editor such as **vi**, which will attempt to reset the terminal when starting or exiting, or simply log out and log back in using a fresh terminal window.

Unix vs. Windows Text Files

While it is the programs that interpret the contents of a file, there are some conventions regarding text file format that all Unix programs follow, so that they can all manipulate the same files. Unfortunately, DOS and Windows programs follow different conventions. Unix programs assume that text files terminate each line with a control character known as a *line feed* (also known as a *newline*), which is character 10 in the standard character sets. DOS and Windows programs tend to use both a *carriage return* (character 13) and a line feed.

To compound the problem, many Unix editors and tools also run on Windows (under Cygwin, for example). As a result, text files may end up with a mixture of line-terminations after being edited on both Windows and Unix.

Some programs are smart enough to properly handle either line termination convention. However, many others will misbehave if they encounter the "wrong" type of line termination.

The **dos2unix** and **unix2dos** commands can be used to clean up files that have been transferred between Unix and DOS/Windows. These programs convert text files between the DOS/Windows and Unix standards. If you've edited a text file on a non-Unix system, and are now using it on a Unix system, you can clean it up by running:

```
shell-prompt: dos2unix filename
```

The **dos2unix** and **unix2dos** commands are not standard with most Unix systems, but they are free programs that can easily be added.

Caution Note that **dos2unix** and **unix2dos** should only be used on text files. They should never be used on binary files, since the contents of a binary file are not meant to be interpreted as characters such as line feeds and carriage returns.

1.10.2 File system Organization

Basic Concepts

A Unix file system contains *files* and *directories*. A file is like a document, and a directory is like a folder that contains documents and/or other directories. The terms "directory" and "folder" are interchangeable, but "directory" is the standard term used in Unix.

Note
Unix file systems use case-sensitive file and directory names. I.e., `Temp` is not the same as `temp`, and both can coexist in the same directory.
Mac OS X is the only major Unix system that violates this convention. The standard OS X file system (called HFS) is case-preserving, but not case-sensitive. This means that if you call a file `Temp`, it will remember that the T is capital, but it can also be referred to as `temp`, `tEmp`, etc. Only one of these files can exist in a given directory at any one time.

A Unix file system can be visualized as a tree, with each file and directory contained within another directory.

Figure 1.1 shows a small portion of a typical Unix file system. On a real Unix system, there are usually thousands of files and directories. Directories are shown in green and files are in yellow.

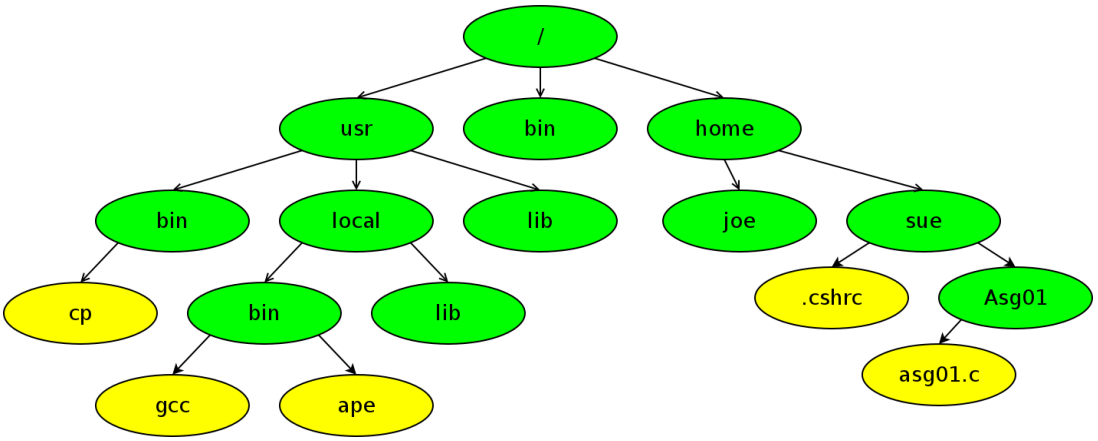


Figure 1.1: Sample of a Unix File system

The one directory that is not contained within any other is known as the *root directory*, whose name under Unix is `/`. There is exactly one root directory on every Unix system. Windows systems, on the other hand, have a root directory for each disk partition such as C: and D:.

The Cygwin compatibility layer works around the separate drive letters of Windows by unifying them under a common parent directory called /cygdrive. Hence, for Unix commands run under Cygwin, /cygdrive/c is equivalent to c:, /cygdrive/d is equivalent to d:, and so on. This allows Cygwin users to traverse multiple Windows drive letters with a single command starting in /cygdrive.

Unix file system trees are fairly standardized, but most have some variation. For instance, all Unix systems have a /bin and a /usr/bin, but not all of them have /home or /usr/local.

The root directory is the *parent* of /bin and /home and an *ancestor* of all other files and directories.

The /bin and /home directories are *subdirectories*, or *children* of /. Likewise, /home/joe and /home/sue are subdirectories of /home, and grandchildren of /.

All of the files in and under /home comprise a *subtree* of /home.

The children of a directory, all of its children, and so on, are known as *descendants* of the directory. All files and directories on a Unix system, except /, are descendants of /.

Each user has a *home directory*, which can be arbitrarily assigned, but is generally a child of /home on many Unix systems. The home directory can be referred to as `~` or `~username` in modern Unix shells.

In the example above, /home/joe is the home directory for user joe, and /home/sue is the home directory for user sue. This is the conventional location for home directories on BSD and Linux systems, which are two specific types of Unix. On a Mac OS X system, which is another brand of Unix, Joe's home directory would be /Users/joe instead of /home/joe. Most of the files owned by ordinary users are either in their home directory or one of its descendants.

Absolute Path Names

The *absolute path name*, also known as *full path name*, of a file or directory denotes the complete path from / (the root directory) to the file or directory of interest. For example, the absolute path name of Sue's .cshrc file is /home/sue/.cshrc, and the absolute path name of the ape command is /usr/local/bin/ape.

Note An absolute path name always begins with '/' or a '~'.

Practice Break
Try the following commands:

```
shell-prompt: ls
shell-prompt: ls /etc
shell-prompt: cat /etc/motd
```

Current Working Directory

Every Unix process has an attribute called the *current working directory*, or *CWD*. This is the directory that the process is currently "in". When you first log into a Unix system, the shell process's CWD is set to your home directory.

The **pwd** command prints the CWD of the shell process. The **cd** command changes the CWD of the shell process. Running **cd** with no arguments sets the CWD to your home directory.

Practice Break
Try the following commands:

```
shell-prompt: pwd
shell-prompt: cd /
shell-prompt: pwd
shell-prompt: cd
shell-prompt: pwd
```

Some commands, such as **ls**, use the CWD as a default if you don't provide a directory name on the command line. For example, if the CWD is `/home/joe`, then the following commands are the same:

```
shell-prompt: ls
shell-prompt: ls /home/joe
```

Relative Path Names

Whereas an absolute path name denotes the path from / to a file or directory, the *relative path name* denotes the path from the CWD to a file or directory.

Any path name that does not begin with a '/' or '~' is interpreted as a relative path name. The absolute path name is then derived by appending the relative path name to the CWD. For example, if the CWD is `/etc`, then the relative path name `motd` refers to the absolute path name `/etc/motd`.

absolute path name = CWD + "/" + relative path name

Note Relative path names are handled at the lowest level of the operating system, by the Unix kernel. This means that they can be used anywhere: in shell commands, in C or Fortran programs, etc.

When you run a program from the shell, the new process inherits the CWD from the shell. Hence, you can use relative path names as arguments in any Unix command, and they will use the CWD inherited from the shell. For example, the two **cat** commands below have the same effect.

```
shell-prompt: cd /etc          # Set shell's CWD to /etc
shell-prompt: cat motd         # Inherits CWD from shell
shell-prompt: cat /etc/motd
```

Wasting Time
The **cd** command is one of the most overused Unix commands. Many people use it where it is completely unnecessary and actually results in significantly more typing than needed. Don't use **cd** where you could have used the directory with another command. For example, the sequence of commands:

```
shell-prompt: cd /etc
shell-prompt: more hosts
shell-prompt: cd
```

The same effect could have been achieved much more easily using the following single command:

```
shell-prompt: more /etc/hosts
```

Note In almost all cases, absolute path names and relative path names are interchangeable. You can use either type of path name as a command line argument, or within a program.

Practice Break
Try the following commands:

```
shell-prompt: cd
shell-prompt: pwd
shell-prompt: cd /etc
shell-prompt: pwd
shell-prompt: cat motd
shell-prompt: cat /etc/motd
shell-prompt: cd
shell-prompt: pwd
shell-prompt: cat motd
```

Why does the last command result in an error?

Avoid Absolute Path Names

The relative path name is potentially much shorter than the absolute path name. Using relative path names also provides more flexibility.

Suppose you have a project contained in the directory /Users/joe/Thesis on your Mac workstation.

Now suppose you want to work on the same project on a cluster, where there is no /Users directory and you have to store it in /share1/joe/Thesis.

The absolute path name of every file and directory will be different on the cluster than it is on your Mac. This can cause major problems if you were using absolute path names in your scripts, programs, and makefiles. Statements like the following will have to be changed in order to run the program on a different computer.

```
infile = fopen("/Users/joe/Thesis/Inputs/input1.txt", "r");
```

```
sort /Users/joe/Thesis/Inputs/names.txt
```

No program should ever have to altered just to run it on a different computer.

While the absolute path names change when you move the Thesis directory, the path names relative to the Thesis directory remain the same. For this reason, absolute path names should be avoided unless absolutely necessary.

The statements below will work on any computer as long as the program or script is running with Thesis as the current working directory. It does not matter where the Thesis directory is located, so long as the Inputs directory is its child.

```
infile = fopen("Inputs/input1.txt", "r");
```

```
sort Inputs/names.txt
```

Special Directory Names

In addition to absolute path names and relative path names, there are a few special symbols for directories that are commonly referenced:

Symbol	Refers to
.	The current working directory
..	The parent of the current working directory
~	Your home directory
~user	user's home directory

Table 1.5: Special Directory Symbols

Practice Break
Try the following commands and see what they do:

```
shell-prompt: cd
shell-prompt: pwd
shell-prompt: ls
shell-prompt: ls .
shell-prompt: cp /etc/motd .
shell-prompt: cat motd
shell-prompt: cat ./motd
shell-prompt: ls ~
shell-prompt: ls /bin
shell-prompt: ls ..
shell-prompt: cd ..
shell-prompt: ls
shell-prompt: ls ~
shell-prompt: cd
```

1.10.3 Ownership and Permissions

Overview

Every file and directory on a Unix system has inherent access control features based on a simple scheme:

- Every file and directory has an individual owner and group owner.
- There are 3 types of permissions which are controlled separately from each other:
  - Read
  - Write (create or modify)
  - Execute (e.g. run a file if it's a program)
- Read, write, and execute permissions can be granted or denied separately for each of the following:
  - The individual owner

- The group owner
- All users on the system (a hypothetical group known as "world")

Execute permissions on a file mean that the file can be executed as a script or a program by typing its name. It does not mean that the file actually contains a script or a program: It is up to the owner of the file to set the execute permissions appropriately for each file.

Execute permissions on a directory mean that users in the category can **cd** into it. Users only need read permissions on a directory to list it or access a file within it, but they need execute permissions to make it the current working directory of their processes.

Unix systems provide this access using 9 on/off switches (bits) associated with each file.

Viewing Permissions

If you do a long listing of a file or directory, you will see the ownership and permissions:

```
shell-prompt: ls -l
drwx------   2 joe    users      512 Aug  7 07:52 Desktop/
drwxr-x---  39 joe    users     1536 Aug  9 22:21 Documents/
drwxr-xr-x   2 joe    users      512 Aug  9 22:25 Downloads/
-rw-r--r--   1 joe    users    82118 Aug  2 09:47 bootcamp.pdf
```

The leftmost column shows the type of object and the permissions for each user category.

A '-' in the leftmost character means a regular file, 'd' means a directory, 'l' means a link. etc. Running **man ls** will reveal all the codes.

The next three characters are, in order, read, write and execute permissions for the owner.

The next three after that are permissions for members of the owning group.

The next three are permissions for world.

A '-' in a permission bit column means that the permission is denied for that user or set of users and an 'r', 'w', or 'x' means that it is enabled.

The next two columns show the individual and group ownership of the file or directory. The other columns show the size, the date and time it was last modified, and name. In addition to the 'd' in the first column, directory names are followed by a '/'.

You can see above that Joe's `Desktop` directory is readable, writable, and executable for Joe, and completely inaccessible to everyone else.

Joe's `Documents` directory is readable, writable and executable for Joe, and readable and executable for members of the group "users". Users not in the group "users" cannot access the Documents directory at all.

Joe's `Downloads` directory is readable and executable to anyone who can log into the system.

The file `bootcamp.pdf` is readable by the world, but only writable by Joe. It is not executable by anyone, which makes sense because a PDF file is not a program.

Setting Permissions

Users cannot change individual ownership on a file, since this would allow them to subvert disk quotas by placing their files under someone else's name. Only the *superuser* can change the individual ownership of a file or directory.

Users can change the group ownership of a file to any group that they belong to using the **chgrp** command:

```
shell-prompt: chgrp group path [path ...]
```

All sharing of files on Unix systems is done by controlling group ownership and file permissions.

File permissions are changed using the **chmod** command:

```
shell-prompt: chmod permission-specification path [path ...]
```

The permission specification has a symbolic form, and a raw form, which is an octal number.

The basic symbolic form consists of any of the three user categories 'u' (user/owner), 'g' (group), and 'o' (other/world) followed by a '+' (enable) or '-' (disable), and finally one of the three permissions 'r', 'w', or 'x'.

Add read and execute permissions for group and world on the Documents directory:

```
shell-prompt: chmod go+rx Documents
```

Disable all permissions for world on the Documents directory and enable read for group:

```
shell-prompt: chmod o-rwx,g+r Documents
```

Disable write permission for everyone, including the owner, on bootcamp.pdf. (This can be used to reduce the chances of accidentally deleting an important file.)

```
shell-prompt: chmod ugo-w bootcamp.pdf
```

Run **man chmod** for additional information.

The raw form for permissions uses a 3-digit octal number to represent the 9 permission bits. This is a quick and convenient method for computer nerds who can do octal/binary conversions in their head.

```
shell-prompt: chmod 644 bootcamp.pdf   # 644 = 110100100 = rw-r--r--
shell-prompt: chmod 750 Documents      # 750 = 111101000 = rwxr-x---
```

Caution NEVER make any file or directory world-writable. Doing so allows any other user to modify it, which is a serious security risk. A malicious user could replace use this to install a Trojan Horse program under your name, for example.

Practice Break

Try the following commands, and try to predict the output of each **ls** before you run it.

```
shell-prompt: touch testfile
shell-prompt: ls -l
shell-prompt: chmod go-rwx testfile
shell-prompt: ls -l
shell-prompt: chmod o+rw testfile
shell-prompt: ls -l
shell-prompt: chmod g+rwx testfile
shell-prompt: ls -l
shell-prompt: rm testfile
```

Now set permissions on testfile so that it is readable, writable, and executable by you, only readable by the group, and inaccessible to everyone else.

1.10.4 Self-test

1. What is a Unix file?
2. Explain the difference between a text file and a binary file.
3. Do Unix operating systems distinguish between text files and binary files? Explain.

4. Does the Unix standard include a convention on the format of text files?
5. Are Unix text files the same as Windows (and DOS) text files? Explain.
6. How can text files be converted between Windows and Unix conventional formats?
7. What is a directory? Does it go by any other names?
8. How are files and directories organized in a Unix file system?
9. What are some of the conventional directories in the Unix file system organization?
10. What is the root directory?
11. What is a parent directory?
12. What is a sibling directory?
13. What is a child directory?
14. What is a subdirectory?
15. What is a descendant directory?
16. What is an ancestor directory?
17. What is a home directory?
18. What is a subtree?
19. What is a full or absolute path name?
20. What is the current working directory? What is a current working directory a property of?
21. What is a relative path name? How can you convert a relative path name to an absolute path name?
22. Why should absolute path names be avoided?
23. How can you determine the current working directory of your shell process?
24. How can you change the current working directory of your shell process to each of the following?
    (a) Your home directory.
    (b) /etc
    (c) The directory `Program1`, which is a subdirectory of `Programs`, which is a subdirectory of the current working directory.
25. Show the simplest Unix command to view each of the following files?
    (a) `/etc/hosts`
    (b) A file called .cshrc in the current working directory.
    (c) A file called .cshrc in your home directory, regardless of what the current working directory is.
    (d) A file called .cshrc in the home directory of a user with user name "bacon".
    (e) A file called `readme.txt` in the parent directory of the current working directory.
26. How can you change the group ownership of the file `.cshrc` in your home directory to the group "smithlab"?
27. How can you change the individual ownership of the file `.cshrc` in your home directory to your friend with user name Bob? Explain.
28. How can you change the permissions on the file `.cshrc` in your home directory so that only you can modify it, members of the group can read and execute it but not modify it, and anyone else can read it but not modify or execute it?
29. How can you see the ownership and permissions on all the files in /etc? In the current working directory?

1.11 Unix Commands and the Shell

Before You Begin You should have a basic understanding of Unix processes, files, and directories. These topics are covered in Section 1.9 and Section 1.10.

Unix commands fall into one of two categories:

- Internal commands are part of the shell.

  No new process is created when you execute an internal command. The shell simply carries out the execution of internal commands by itself.

- External commands are programs separate from the shell. The command name of an external command is actually the name of an *executable file*, i.e. a file containing the program or script. For example, when you run the **ls** command, you are executing the program contained in the file `/bin/ls`.

  When you run an external command, the shell locates the program file, loads the program into memory, and creates a new (child) process to execute the program. The shell then normally waits for the child process to end before prompting you for the next command.

1.11.1 Internal Commands

Commands are implemented internally only when it is necessary, or when there is a substantial benefit. If all commands were part of the shell, the shell would be enormous and would require too much memory.

An example is the **cd** command, which changes the CWD of the shell process. The **cd** command cannot be implemented as an external command, since the CWD is a property of the process.

We can prove this using *Proof by Contradiction*. Assuming the **cd** command is external, it would run as a child process of the shell. Hence, running **cd** would create a child process, which would alter its CWD, and then terminate. Altering the CWD of a child process does not affect the CWD of the parent process. Remember that every process in a Unix system has it's own independent CWD.

Expecting an external command to change your CWD for you would be akin to asking one of your children to go to take a shower for you. Neither is capable of affecting the desired change.

Likewise, any command that alters the state of the shell process must be implemented as an internal command.

A command might also be implemented internally simply because it's trivial to do so, and it saves the overhead of loading and running an external command. When the work done by a command is very simple, it might take more resources to load an external program than it does to actually run it. In these cases, it makes more sense to implement it as part of the shell.

1.11.2 External Commands

The executable files containing external commands are kept in certain directories, most of which are called `bin` (short for binary, since most executable files are binary files). The most essential commands that are common to most Unix systems are kept in `/bin` and `/usr/bin`. The location of optional add-on commands varies, but a typical location is `/usr/local/bin`.

The list of directories that are searched when looking for external commands is kept in an *environment variable* called `PATH`. The environment is discussed in more detail in Section 1.15.

Practice Break

1. Use **which** under C shell family shells (csh and tcsh) to find out whether the following commands are internal or external. Use **type** under Bourne family shells (bash, ksh, dash, zsh). You can use either command under either shell, but will get better results if you follow the advice above. (Try both and see what happens.)

```
shell-prompt: which cd
shell-prompt: which cp
shell-prompt: which exit
shell-prompt: which ls
shell-prompt: which pwd
```

2. Find out where your external commands are stored by running **echo $PATH**.
3. Use **ls** to find out what commands are located in `/bin` and `/usr/bin`.

1.11.3 Getting Help

In the olden days before Unix, when programmers wanted to look up a command or function, they would have to get out of their chair and walk somewhere to pick up a typically ring-bound manual to flip through.

The Unix designers saw this as a waste of time. They thought, wouldn't it be nice if we could sit in the same chair for ten hours straight without ever taking our eyes off the monitor or our hands off the keyboard?

And so, online documentation was born. On Unix systems, all common Unix commands are documented in detail on the Unix system itself, and the documentation is accessible via the command line (you do not need a GUI to view it). Whenever you want to know more about a particular Unix command, you can find out by typing **man command-name**. For example, to learn all about the **ls** command, type:

```
shell-prompt: man ls
```

The **man** covers virtually every common command, as well as other topics. It even covers itself:

```
shell-prompt: man man
```

The **man** command displays a nicely formatted document known as a *man page*. It uses a file viewing program called **more**, which can be used to browse through text files very quickly. Table 1.6 shows the most common keystrokes used to navigate a man page. For complete information on navigation, run:

```
shell-prompt: man more
```

Key	Action
h	Show key commands
Space bar	Forward one page
Enter/Return	Forward one line
b	Back one page
/	Search

Table 1.6: Common hot keys in **more**

Man pages include a number of standard sections, such as SYNOPSIS, DESCRIPTION, and SEE ALSO, which helps you identify other commands that might be of use.

Man pages do not always make good tutorials. Sometimes they contain too much detail, and they are often not well-written for novice users. If you're learning a new command for the first time, you might want to consult a Unix book or the WEB. The man pages will provide the most detailed and complete reference information on most commands, however.

The **apropos** command is used to search the man page headings for a given topic. It is equivalent to **man -k**. For example, to find out what man pages exist regarding Fortran, we might try the following:

```
shell-prompt: apropos fortran
```

or

```
shell-prompt: man -k fortran
```

The **whatis** is similar to **apropos** in that it lists short descriptions of commands. However, **whatis** only lists those commands with the search string in their name or short description, whereas **apropos** attempts to list everything related to the string.

The **info** command is an alternative to man that uses a non-graphical hypertext system instead of flat files. This allows the user to navigate extensive documentation more efficiently. The **info** command has a fairly high learning curve, but it is very powerful, and is often the best option for documentation on a given topic. Some open source software ships documentation in info format and provides a man page (converted from the info files) that actually has less information in it.

```
shell-prompt: info gcc
```

Practice Break

1. Find out how to display a '/' after each directory name and a '*' after each executable file when running **ls**.
2. Use **apropos** to find out what Unix commands to use with bzip files.

1.11.4 A Basic Set of Unix Commands

Most Unix commands have short names which are abbreviations or acronyms for what they do. (pwd = print working directory, cd = change directory, ls = list, ...) Unix was originally designed for people with good memories and poor typing skills.

Some of the most commonly used Unix commands are described below.

Note This section is meant to serve as a quick reference, and to inform new readers about which commands they should learn. There is much more to know about these commands than we can cover here. For full details about any of the commands described here, consult the **man** pages, **info** pages, or the WEB.

This section uses the same notation conventions as the Unix man pages:

- Optional arguments are shown inside [].
- The pipe symbol (|) between two items means one or the other.
- An ellipses (...) means optionally more of the same.
- "file" means a filename is required and a directory name is not allowed. "directory" means a directory name is required, and a filename is not allowed. "path" means either a filename or directory name is acceptable.

File and Directory Management

cp copies one or more files.

```
shell-prompt: cp source-file destination-file
shell-prompt: cp source-file [source-file ...] destination-directory
```

If there is only one source filename, then destination can be either a filename or a directory. If there are multiple source files, then destination must be a directory. If destination is a filename, and the file exists, it will be overwritten.

```
shell-prompt: cp file file.bak     # Make a backup copy
shell-prompt: cp file file.bak ~   # Copy files to home directory
```

ls lists files in CWD or a specified file or directory.

```
shell-prompt: ls [path ...]
```

```
shell-prompt: ls           # List CWD
shell-prompt: ls /etc      # List /etc directory
```

mv moves or renames files or directories.

```
shell-prompt: mv source destination
shell-prompt: mv source [source ...] destination-directory
```

If multiple sources are given, destination must be a directory.

```
shell-prompt: mv prog1.c Programs
```

ln link files or directories.

```
shell-prompt: ln source-file destination-file
shell-prompt: ln -s source destination
```

The **ln** command creates another path name for the same file. Both names refer to the same file, and changes made through one appear in the other. Without `-s`, a standard directory entry, known as a *hard link* is created. In this case, source and destination must be on the same partition. (The **df** will list partitions and their location within the directory tree.) With `-s`, a *symbolic link* is created. A symbolic link is not a standard directory entry, but a pointer to the source path name. Only symbolic links can be used for directories, and symbolic links to not have to be on the same partition as the source.

```
shell-prompt: ln -s /etc/motd ~    # Make a convenient link to motd
```

rm removes one or more files.

```
shell-prompt: rm file [file ...]
```

```
shell-prompt: rm temp.txt core a.out
```

Caution Removing files with **rm** is not like dragging them to the trash. Once files are removed by **rm**, they cannot be recovered.

srm (secure rm) removes files securely, erasing the file content and directory entry so that the file cannot be recovered. Use this to remove files that contain sensitive data. This is not a standard Unix command, but a free program that can be easily installed on most systems.

mkdir creates one or more directories.

```
shell-prompt: mkdir [-p] path name [path name ...]
```

The `-p` flag indicates that mkdir should attempt to create any parent directories in the path that don't already exist. If not used, **mkdir** will fail unless all but the last component of the path exist.

```
shell-prompt: mkdir Programs
shell-prompt: mkdir -p Programs/C/MPI
```

rmdir removes one or more empty directories.

```
shell-prompt: rmdir directory [directory ...]
```

rmdir will fail if a directory is not completely empty. You may also need to check for hidden files using **ls -a directory**.

```
shell-prompt: rmdir Programs/C/MPI
```

find locates files within a subtree using a wide variety of possible criteria.

```
shell-prompt: find start-directory criteria [action]
```

find is a very powerful and complex command that can be used to not only find files, but run commands on the files matching the search criteria.

```
shell-prompt: find . -name core                    # List cores
shell-prompt: find . -name core -exec rm '{}' \;   # Remove cores
```

df shows the free disk space on all currently mounted partitions.

```
shell-prompt: df
```

du reports the disk usage of a directory and everything under it.

```
shell-prompt: du [-s] [-h] path
```

The `-s` (summary) flag suppresses output about each file in the subtree, so that only the total disk usage of the directory is shown.

```
shell-prompt: du -sh Programs
```

Shell Internal Commands

As mentioned previously, internal commands are part of the shell, and serve to control the shell itself. Below are some of the most common internal commands.

cd changes the current working directory of the shell process. It is described in more detail in Section 1.10.2.

```
shell-prompt: cd [directory]
```

The **pwd** command prints the CWD of the shell process. It is described in detail in Section 1.10.2[1]. You can use **pwd** like a Unix file system GPS, to get your bearing when you're lost.

pushd changes CWD and saves the old CWD on a stack so that we can easily return.

```
shell-prompt: pushd directory
```

Users often encounter the need to temporarily go to another directory, run a few commands, and then come back to the current directory.

The **pushd** command is a very useful alternative to **cd** that helps in this situation. It performs the same operation as **cd**, but it records the starting CWD by adding it to the top of a stack of CWDs. You can then return to where the last **pushd** command was invoked using **popd**. This saves you from having to retype the path name of the directory you want to return to. Not all shells support pushd and popd, but the ones you are likely to use for a login session do.

[1] **pwd** is actually an external command, but we cover it here since it relates to the CWD of the shell process. (recall that new processes inherit the CWD of the shell process, so **pwd** need not be internal.)

Practice Break
Try the following sequence of commands:

```
shell-prompt: pwd          # Check starting point
shell-prompt: pushd /etc
shell-prompt: more motd
shell-prompt: pushd /home
shell-prompt: ls
shell-prompt: popd         # Back to /etc
shell-prompt: pwd
shell-prompt: more motd
shell-prompt: popd         # Back to starting point
shell-prompt: pwd
```

exit terminates the shell process.

```
shell-prompt: exit
```

This is the most reliable way to exit a shell. In some situations you could also type **logout** or simply press Ctrl+d, but these alternatives will not work for every shell process.

Text File Processing

cat echoes the contents of one or more text files.

```
shell-prompt: cat file [file ...]
```

```
shell-prompt: cat /etc/motd
```

more shows the contents of one or more text files interactively.

```
shell-prompt: more file [file ...]
```

```
shell-prompt: more prog1.c
```

head shows the top N lines of one or more text files.

```
shell-prompt: head -n # file [file ...]
```

If a flag consisting of a - followed by an integer number N is given, the top N lines are shown instead of the default of 10.

```
shell-prompt: head -n 5 prog1.c
```

tail shows the bottom N lines of one or more text files.

```
shell-prompt: tail -n # file [file ...]
```

Tail is especially useful for viewing the end of a large file that would be cumbersome to view with **more**.

If a flag consisting of a - followed by an integer number N is given, the bottom N lines are shown instead of the default of 10.

```
shell-prompt: tail -n 5 output.txt
```

grep shows lines in one or more text files that match a given *regular expression*.

```
shell-prompt: grep regular-expression file [file ...]
```

The regular expression is most often a simple string, but can represent patterns as described by **man re_format**.

Show all lines containing the string "printf" in prog1.c.

```
shell-prompt: grep printf prog1.c
```

Show all lines containing the variable names in prog1.c. (Variable names begin with a letter or underscore and may contain letters, underscores, or digits after that.)

```
shell-prompt: grep '[a-zA-Z_][a-zA-Z0-9_]*' prog1.c
```

The **diff** command shows the differences between two text files. This is most useful for comparing two versions of the same file to see what has changed. Also see **cdiff**, a specialized version of **diff**, for comparing C source code.

```
shell-prompt: diff -u input1.txt input2.txt
```

Text Editors

There are more text editors available for Unix systems than any one person is aware of. Some are terminal-based, some are graphical, and some have both types of interfaces.

All Unix systems support running graphical programs from remote locations, but most graphical programs require a fast connection (10 megabits/sec) or more to function tolerably.

Knowing how to use a terminal-based text editor is therefore a very good idea, so that you're prepared to work over a slow connection if necessary. Some of the more common terminal-based editors are described below.

vi (visual editor) is the standard text editor for all Unix systems. Most users either love or hate the vi interface, but it's a good editor to know since it is standard on every Unix system.

nano is an extremely simplistic text editor that is ideal for beginners. It is a rewrite of the **pico** editor, which is known to have many bugs and security issues. Neither editor is standard on Unix systems, but both are free and easy to install. These editors entail little or no learning curve, but are not sophisticated enough for extensive programming or scripting.

emacs (Edit MACroS) is a more sophisticated editor used by many programmers. It is known for being hard to learn, but very powerful. It is not standard on most Unix systems, but is free and easy to install.

ape is a menu-driven, user-friendly IDE (integrated development environment), i.e. programmer's editor. It has an interface similar to PC and Mac programs, but works on a standard Unix terminal. It is not standard on most Unix systems, but is free and easy to install. **ape** has a small learning curve, and advanced features to make programming much faster.

Networking

hostname prints the network name of the machine.

```
shell-prompt: hostname
```

This is often useful when you are working on multiple Unix machines at the same time (e.g. via **ssh**), and forgot which window applies to each machine.

ssh is used to remotely log into another machine on the network and start a shell.

```
    ssh [name@]hostname
```

```
shell-prompt: ssh joe@login.peregrine.hpc.uwm.edu
```

sftp is used to remotely log into another machine on the network and transfer files to or from it.

```
shell-prompt: sftp [name@]host
```

```
shell-prompt: sftp joe@data.peregrine.hpc.uwm.edu
```

rsync is used to synchronize two directories either on the same machine or on different machines.

```
shell-prompt: rsync [flags] [[name@]host:]path [[name@]host:]path
```

rsync compares the contents of the two source and destination directories and transfers only the differences. Hence, it can save an enormous amount of time when you make small changes to a large project and need to synchronize another copy of the project.

```
shell-prompt: rsync -av Project joe@data.peregrine.hpc.uwm.edu:
```

Identity and Access Management

passwd changes your password. It asks for your old password once, and the new one twice (to ensure that you don't accidentally set your password to something you don't know because your finger slipped). Unlike many graphical password programs, **passwd** does not echo anything for each character typed. (Even showing the length of your password is a bad idea from a security standpoint.)

```
shell-prompt: passwd
```

Terminal Control

clear clears your terminal screen (assuming the TERM variable is properly set).

```
shell-prompt: clear
```

reset resets your terminal to an initial state. This is useful when your terminal has been corrupted by bad output, such as when attempting to view a binary file.

Terminals are controlled by *magic sequences*, sequences of invisible control characters sent from the host computer to the terminal amid the normal output. Magic sequences move the cursor, change the color, change the international character set, etc. Binary files contain random data that sometimes by chance contain magic sequences that could alter the mode of your terminal. If this happens, running **reset** will usually correct the problem. If not, you will need to log out and log back in.

```
shell-prompt: reset
```

1.11.5 Self-test

1. What is an internal command?
2. What is an external command?
3. What kinds of commands must be implemented as internal commands?
4. How can you quickly view detailed information on a Unix command?
5. How can you identify Unix commands related to a topic? (Describe two methods.)
6. Show the simplest Unix command to accomplish each of the following in order:
   (a) List the files in `/usr/local/share`.
   (b) Make your home directory the CWD.
   (c) Copy the file `/etc/hosts` to your home directory.
   (d) Rename the file `~/hosts` to `~/hosts.bak`.
   (e) Create a subdirectory called `~/Temp`.
   (f) Make `~/Temp` the CWD.
   (g) Copy the file `~/hosts.bak` to `~/Temp`.

(h) Create a hard link to the file `~/Temp/hosts.bak` called `~/hosts.bak.temp`.

(i) Create a link to the directory `/usr/local/share` in your home directory.

(j) Make your home directory the CWD.

(k) Remove the entire subtree in `~/Temp` and the files `~/hosts.bak` and `~/hosts.bak.temp`.

(l) Show how much space is used by the directory `/etc`.

7. Show the simplest Unix command to accomplish each of the following:

   (a) Change the current working directory of your shell process to /etc, remembering the previous current working directory on the directory stack.

   (b) Return to the previous current working directory on the directory stack.

   (c) Terminate the shell.

8. Show the simplest Unix command to accomplish each of the following:

   (a) Show the contents of the text file `/etc/motd` a page at a time.

   (b) Show the first 5 lines of `/etc/motd`.

   (c) Show the last 8 lines of `/etc/motd`.

   (d) Show lines in `/etc/group` and `/etc/passwd` containing your username.

   (e) Edit the text file `./prog1.c`.

9. Show the simplest Unix command to accomplish each of the following:

   (a) Show the network name (host name) of the computer running the shell.

   (b) Remotely log into "login.peregrine.hpc.uwm.edu" and start a shell.

   (c) Remotely log into "data.peregrine.hpc.uwm.edu" for the purpose of transferring files.

   (d) Synchronize the folder `~/Programs/Prog1` on login.peregrine.hpc.uwm.edu to `./Prog1`, transferring only the differences.

   (e) Clear the terminal screen.

   (f) Restore functionality to a terminal window that's in a funk.

1.12 Unix Command Quick Reference

Table 1.7 provides a quick reference for looking up common Unix commands. For details on any of these commands, run **man command** (or **info command** on some systems).

1.13 POSIX and Extensions

Unix-compatible systems generally conform to standards published by the International Organization for Standardization (ISO), the Open Group, and the IEEE Computer Society.

The primary standard used for this purpose is *POSIX*, the portable operating system standard based on Unix.

Programs and commands that conform to the POSIX standard should work on any Unix system. Therefore, developing your programs and scripts according to POSIX will prevent problems and wasted time.

Nevertheless, many common Unix programs have been enhanced beyond the POSIX standard to provide useful features. Fortunately, most such programs are open source and can therefore be easily installed on most Unix systems.

Features that do not conform to the POSIX standard are known as *extensions*. Extensions are often described according to their source, e.g. BSD extensions or GNU extensions.

Synopsis	Description
ls [fileldirectory]	List file(s)
cp source-file destination-file	Copy a file
cp source-file [source-file ...] directory	Copy multiple files to a directory
mv source-file destination-file	Rename a file
mv source-file [source-file ...] directory	Move multiple files to a directory
ln source-file destination-file	Create another name for the same file. (source and destination must be in the same file system)
ln -s source destination	Create a symbolic link to a file or directory
rm file [file ...]	Remove one or more files
rm -r directory	Recursively remove a directory and all of its contents
srm file [file ...]	Securely erase and remove one or more files
mkdir directory	Create a directory
rmdir directory	Remove a directory (the directory must be empty)
find start-directory criteria	Find files/directories based on flexible criteria
make	Rebuild a file based on one or more other files
od/hexdump	Show the contents of a file in octal/hexadecimal
awk	Process tabular data from a text file
sed	Stream editor. Echo files, making changes to contents.
sort	Sort text files based on flexible criteria
uniq	Echo files, eliminating adjacent duplicate lines.
diff	Show differences between text files.
cmp	Detect differences between binary files.
cdiff	Show differences between C programs.
cut	Extract substrings from text.
m4	Process text files containing m4 mark-up.
chfn	Change finger info (personal identity).
chsh	Change login shell.
su	Substitute user.
cc/gcc/icc	Compile C programs.
f77/f90/gfortran/ifort	Compile Fortran programs.
ar	Create static object libraries.
indent	Beautify C programs.
astyle	Beautify C, C++, C#, and Java programs.
tar	Pack a directory tree into a single file.
gzip	Compress files.
gunzip	Uncompress gzipped files.
bzip2	Compress files better (and slower).
bunzip2	Uncompress bzipped files.
zcat/zmore/zgrep/bzcat/bzmore/bzgrep	Process compressed files.
exec command	Replace shell process with command.
date	Show the current date and time.
cal	Print a calendar for any month of any year.
bc	Unlimited precision calculator.
printenv	Print environment variables.

Table 1.7: Unix Commands

Many base commands such as awk, make, and sed, may contain extensions that depend on the specific operating system. For example, BSD systems use the BSD versions of awk, make, and sed, which contain BSD extensions, while GNU/Linux systems use the GNU versions of awk, make, and sed, which contain GNU extensions.

When installing GNU software on BSD systems, the GNU version of the command is often prefixed with a 'g', to distinguish it from the native BSD command. For example, on FreeBSD, "make" and "awk" are the BSD implementations and "gmake" and "gawk" would be the GNU implementations. Likewise, on GNU/Linux systems, BSD commands would generally be prefixed with a 'b' or 'bsd'. The "make" and "tar" commands would refer to GNU versions and BSD make would be run as "bmake" and BSD tar as "bsdtar".

All of them will support POSIX features, so if you use only POSIX features, they will all behave the same. If you want to use GNU or other extensions, it's generally best to use the extended command name, e.g. gawk instead of awk.

Program	**Sample of extensions**
BSD Tar	Support for extracting ISO and Apple DMG files
GNU Make	Various "shortcut" rules for compiling multiple source files
GNU Awk	Additional built-in functions

Table 1.8: Common Extensions

1.14 File Transfer

Many users will need to transfer data between other computers and a remote Unix system. For example, users of a shared research computer running Unix will need to transfer input data from their computer to the Unix machine, run the research programs, and finally transfer results back to their computer. There are many software tools available to accomplish this. Some of the more convenient tools are described below.

1.14.1 File Transfers from Unix

For Unix (including Mac and Cygwin) users, the recommended method for transferring files is the **rsync** command. The rsync command is a simple but intelligent tool that makes it easy to synchronize two directories on the same machine or on different machines across a network. Rsync is free software and part of the base installation of many Unix systems including Mac OS X. On Cygwin, you can easily add the rsync package using the Cygwin Setup utility.

Rsync has two major advantages over other file transfer programs:

- If you have transferred the directory before, and only want to update it, rsync will automatically determine the differences between the two copies and only transfer what is necessary. When conducting research that generates large amounts of data, this can save an enormous amount of time.
- If a transfer fails for any reason (which is more likely for large transfers), rsync's inherent ability to determine the differences between two copies allows it to resume from where it left off. Simply run the exact same rsync command again, and the transfer will resume.

The rsync command can either push (send) files from the local machine to a remote machine, or pull (retrieve) files from a remote machine to the local machine. The command syntax is basically the same in both cases. It's just a matter of how you specify the source and destination for the transfer.

The rsync command has many options, but the most typical usage is to create an exact copy of a directory on a remote system. The general rsync command to push files to another host would be:

```
shell-prompt: rsync -av --delete source-path [username@]hostname:[destination-path]
```

Example 1.2 Pushing data with rsync

The following command synchronizes the directory `Project` from the local machine to ~joeuser/Data/Project on Peregrine:

```
shell-prompt: rsync -av --delete Project joeuser@data.peregrine.hpc.uwm.edu:Data
```

The general syntax for pulling files from another host is:

```
shell-prompt: rsync -av --delete [username@]hostname:[source-path] destination-path
```

Example 1.3 Pulling data with rsync

The following command synchronizes the directory ~joeuser/Data/Project on Peregrine to ./Project on the local machine:

```
shell-prompt: rsync -av --delete joeuser@data.peregrine.hpc.uwm.edu:Data/project .
```

If you omit "username@" from the source or destination, rsync will try to log into the remote system with your username on the local system.

If you omit destination-path from a push command or source-path from a pull command, rsync will use your home directory on the remote host.

The command-line flags used above have the following meanings:

-a Use *archive* mode. Archive mode copies all subdirectories recursively and preserves as many file attributes as possible, such as ownership, permissions, etc.

-v Verbose copy: Display names of files and directories as they are copied.

--delete Delete files and directories from the destination if they do not exists in the source. Without --delete, rsync will add and replace files in the destination, but never remove anything.

Caution

Note that a trailing "/" on source-path affects where rsync stores the files on the destination system. Without a trailing "/", rsync will create a directory called "source-path" under "destination-path" on the destination host.

With a trailing "/" on source-path, destination-path is assumed to be the directory that will replace source-path on the destination host. This feature is a somewhat cryptic method of allowing you to change the name of the directory during the transfer. However, it is compatible with the basic Unix **cp** command.

Note also that the trailing "/" only affects the command when applied to source-path. A trailing "/" on destination-path has no effect.

The command below creates an identical copy of the directory Data/Model in Model (/home/bacon/Data/Model to be precise) on data.peregrine.hpc.uwm.edu. The resulting directory is the same regardless of whether the destination directory existed before the command or not.

```
shell-prompt: rsync -av --delete Model bacon@data.peregrine.hpc.uwm.edu:Data
```

The command below dumps the *contents* of Model directly into Data, and deletes everything else in the Data directory! In other words, it makes the destination directory Data identical to the source directory Model.

```
shell-prompt: rsync -av --delete Model/ bacon@data.peregrine.hpc.uwm.edu:Data
```

To achieve the same effect as the command with no "/", you would need to fully specify the destination path:

```
shell-prompt: rsync -av --delete Model/ bacon@data.peregrine.hpc.uwm.edu:Data/Model
```

Note that if using globbing on the remote system, any globbing patterns must be protected from expansion by the local shell by escaping them or enclosing them in quotes. We want the pattern expanded on the remote system, not the local system:

```
shell-prompt: rsync -av --delete bacon@unixdev1.hpc.uwm.edu:Data/Study\* .
shell-prompt: rsync -av --delete 'bacon@unixdev1.hpc.uwm.edu:Data/Study*' .
```

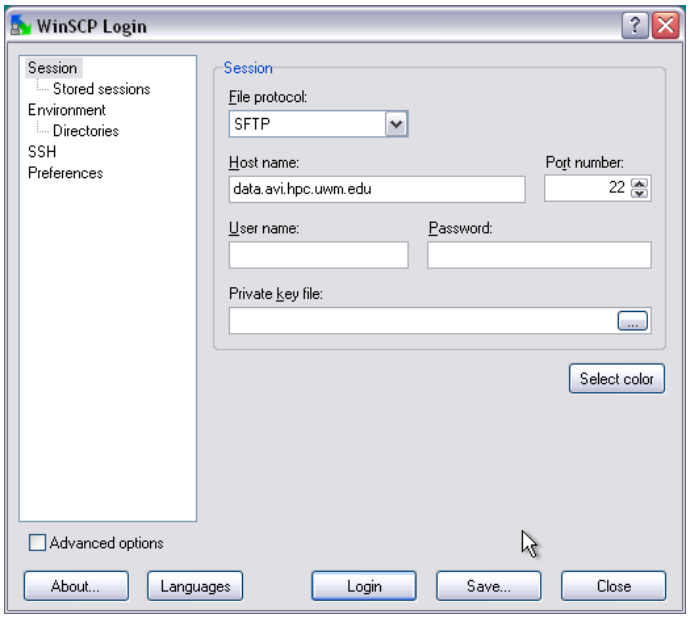
For full details on the rsync command, type

```
shell-prompt: man rsync
```

1.14.2 File Transfer from Windows without Cygwin

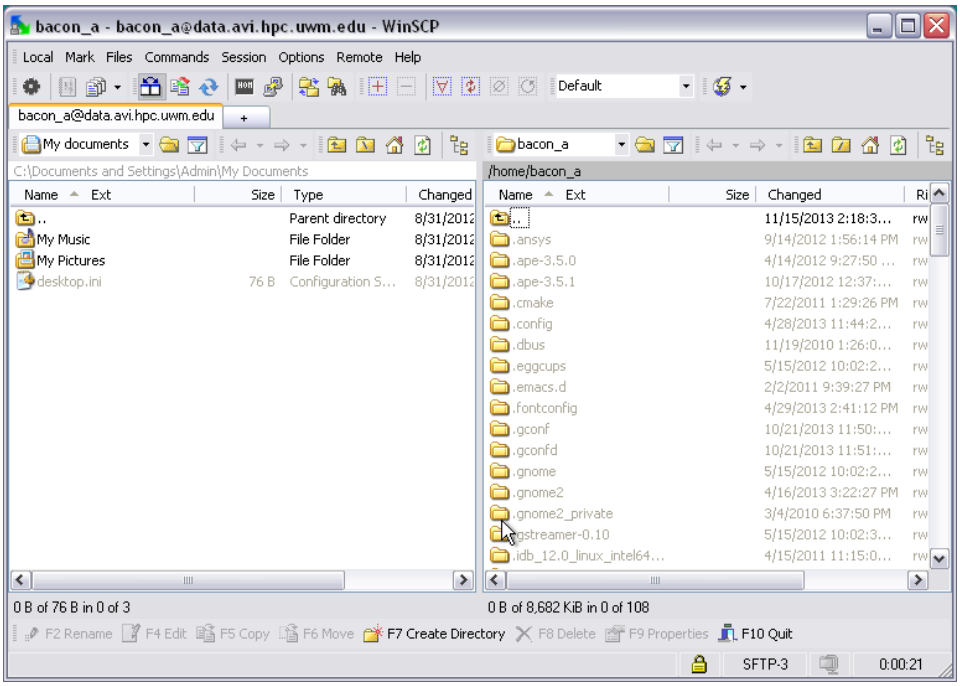
If you're using Cygwin from Windows, you can utilize the rsync command as discussed in Section 1.14.1, provided you've installed the Cygwin rsync package. Otherwise, WinSCP provides a simple way to transfer files to and from your Windows PC. WinSCP is a free program that can be downloaded and installed in a few minutes from http://winscp.net.

After installing WinSCP, simply launch the program, and the following dialog appears:



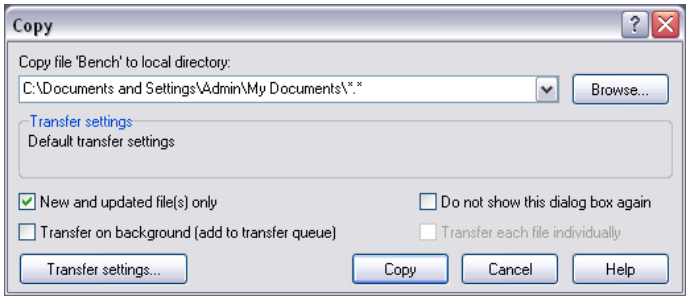
The WinSCP login dialog

WinSCP uses the secure shell protocol to connect to a remote system. Like the Unix **ssh** command, if this is the first time connecting from this computer, you will be asked if you want to add the host key and continue:



A WinSCP session

Once you've successfully logged in, you can simply drag files or directories from one system to the other. If you're updating a large directory that already exists on the destination machine, you may want to check the "New and updated file(s) only" box. This will cause WinSCP to transfer only the files that are different on each end. This feature is a crude approximation to the functionality of **rsync**.



A WinSCP copy operation

1.14.3 Self-test

1. Show the simplest Unix command to accomplish each of the following:

   (a) Copy or synchronize the directory `./PCB-Study` to `~/PCB-Study` under the user "joeuser" on the host data.peregrine.hp

   (b) Copy or synchronize the directory `~/PCB-Study` under the user "joeuser" on the host data.peregrine.hpc.uwm.edu to `./PCB-Study`.

1.15 Environment Variables

Every Unix process maintains a list of character string variables called the *environment*. When a new process is created, it inherits the environment from the process that created it (its parent process).

Since the shell creates a new process whenever you run an external command, the shell's environment can be used to pass information down to any command that you run. For example, text editors and other programs that manipulate the full terminal

screen need to know what type of terminal you are using. Different types of terminals use different magic sequences to move the cursor, clear the screen, scroll, etc. To provide this information, we set the shell's environment variable TERM to the terminal type (usually "xterm"). When you run a command from the shell, it inherits the shell's TERM variable, and therefore knows the correct magic sequences for your terminal.

The **printenv** shows all of the environment variables currently set in your shell process.

```
shell-prompt: printenv
```

Setting environment variables requires a different syntax depending on which shell you are using. Most modern Unix shells are extensions of either Bourne shell (sh) or C shell (csh), so there are only two variations of most shell commands that we need to know for most purposes.

For Bourne shell derivatives, we use the **export** command:

```
shell-prompt: TERM=xterm
shell-prompt: export TERM
```

For C shell derivatives, we use **setenv**:

```
shell-prompt: setenv TERM xterm
```

The `PATH` variable specifies a list of directories containing external Unix commands. When you type a command at the shell prompt, the shell checks the directories listed in `PATH` in order to find the command you typed. For example, when you type the **ls** command, the shell utilizes `PATH` to locate the program in `/bin/ls`.

The directory names within in PATH are separated by colons. A simple value for `PATH` might be `/bin:/usr/bin:/usr/local/bin`. When you type **ls**, the shell first checks for the existence of `/bin/ls`. If it does not exist, the shell then checks for `/usr/bin/ls`, and so on, until it either finds the program or has checked all directories in `PATH`. If the program is not found, the shell issues an error message such as "ls: Command not found".

Environment variables can be set from the shell prompt using the **export** command in Bourne shell and its derivatives (sh, bash, ksh):

```
shell-prompt: export PATH='/bin:/usr/bin:/usr/local/bin'
```

or using **setenv** in C shell and its derivatives (csh, tcsh):

```
shell-prompt: setenv PATH '/bin:/usr/bin:/usr/local/bin'
```

The **env** can be used to alter the environment just for the invocation of one child process, rather than setting it for the current shell process.

Suppose Bob has a script called **rna-trans** that we would like to run in his `~/bin` directory. This script also invokes other scripts in the same directory, so we'll need it in our path while his script runs.

```
shell-prompt: env PATH='/bin:/usr/bin:/usr/local/bin:~bob/bin' rna-trans
```

1.15.1 Self-test

1. What are environment variables?
2. Does a Unix process have any environment variables when it starts? If so, where do they come from?
3. How can environment variables be used to communicate information to child processes?
4. Describe one common environment variable that it typically set by the shell and used by processes running under the shell.
5. Show how to set the environment variable TERM to the value "xterm" in
   (a) Bourne shell (sh)

(b) Korn shell (ksh)

(c) Bourne again shell (bash)

(d) C shell (csh)

(e) T-shell (tcsh)

6. Show a Unix command that runs **ls** with the LSCOLORS environment variable set to "CxFxCxDxBxegedaBaGaCaD". You may not change the LSCOLORS variable for the current shell process.

1.16 Shell Variables

In addition to the environment, shells maintain a similar set of variables for their own use. These variables are not passed down to child processes, and are only used by the shell.

Shell variables can be arbitrary, but each shell also treats certain variable names specially. One common example is the shell variable that stores the shell prompt.

In Bourne-shell derivatives, this variable is called PS1. To set a shell variable in Bourne-shell derivatives, we use a simple assignment. (The export command above actually sets a shell variable called TERM and then exports it to the environment.)

```
shell-prompt: PS1="peregrine: "
```

In C shell derivatives, the variable is called `prompt`, and is set using the **set** command:

```
shell-prompt: set prompt="peregrine: "
```

Note The syntax for **set** is slightly different than for **setenv**. **Set** uses an '=' while **setenv** uses a space.

Shell prompt variables may contain certain special symbols that represent dynamic information they you might want to include in your shell prompt, such as the host name, command counter, current working directory, etc. Consult the documentation for your shell for details.

In all shells, you can view the current shell variables by typing **set** with no arguments:

```
shell-prompt: set
```

1.16.1 Self-test

1. Show how to set the shell prompt to "Peregrine: " in:

   (a) Bourne shell

   (b) C shell

2. How can you view a list of all current shell variables and their values?

1.17 More Shell Tools

1.17.1 Redirection and Pipes

Device Independence

Many operating systems that came before Unix treated each input or output device differently. Each time a new device became available, programs would have to be modified in order to access it. This is intuitive, since the devices all look different and perform different functions.

The Unix designers realized that this is actually unnecessary and a waste of programming effort, so they employed the concept of *device independence*. Unix device independence works by treating virtually every input and output device exactly like an ordinary file. All input and output, whether to/from a file on a disk, a keyboard, a mouse, a scanner, or a printer, is simply a stream of bytes to be input or output by a program.

Most I/O devices are actually accessible as a *device file* in `/dev`. For example, the primary CD-ROM might be `/dev/cd0`, and the main disk might be `/dev/ad0`.

Data are often recovered from corrupted file systems or accidentally deleted files by reading the raw disk partition as a file using standard Unix commands such as grep!

```
shell-prompt: grep string /dev/ad0s1f
```

To see the raw input from a mouse as it is being moved, one could use the following command:

```
shell-prompt: hexdump /dev/mouse
```

cat /dev/mouse would also work, but the binary data stream would appear as garbage on the terminal screen.

Some years ago while mentoring my son's robotics team, as part of a side project, I reverse-engineered a USB game pad so I could control a Lego robot via Bluetooth from a laptop. Thanks for device-independence, no special software was needed to figure out the game pad's communication protocol.

After plugging the game pad into my FreeBSD laptop, the **dmesg** command shows the name of the new device file created under `/dev`.

```
ugen1.2: <vendor 0x046d product 0xc216> at usbus1
uhid0 on uhub3
uhid0: <vendor 0x046d product 0xc216, class 0/0, rev 1.10/3.00, addr 2> on usbus1
```

One can then view the input from the game pad using **hexdump**. It was easy to see that moving the right joystick up resulted in lower numbers in the 3rd and 7th columns, while moving down increased the values. Center position sends a value around 8000 (hexadecimal), fully up is around 0, fully down is ffff. Analogous results were seen for the other joystick and left or right motion, as well as the various buttons. It was then relatively easy to write a small program to read the joystick position from the game pad and send commands over Bluetooth to the robot, adjusting motor speeds accordingly. Sending commands over Bluetooth is also done with the same functions as writing to a file.

```
FreeBSD manatee.acadix  bacon ~ 410: hexdump /dev/uhid0
0000000 807f 7d80 0008 fc04 807f 7b80 0008 fc04
0000010 807f 7780 0008 fc04 807f 6780 0008 fc04
0000020 807f 5080 0008 fc04 807f 3080 0008 fc04
0000030 807f 0d80 0008 fc04 807f 0080 0008 fc04
0000060 807f 005e 0008 fc04 807f 005d 0008 fc04
0000070 807f 0060 0008 fc04 807f 0063 0008 fc04
0000080 807f 006c 0008 fc04 807f 0075 0008 fc04
0000090 807f 0476 0008 fc04 807f 1978 0008 fc04
00000a0 807f 4078 0008 fc04 807f 8c7f 0008 fc04
00000b0 807f 807f 0008 fc04 807f 7f7f 0008 fc04
00000c0 807f 827f 0008 fc04 807f 847f 0008 fc04
```

```
00000d0 807f 897f 0008 fc04 807f 967f 0008 fc04
00000e0 807f a77f 0008 fc04 807f be80 0008 fc04
00000f0 807f d980 0008 fc04 807f f780 0008 fc04
0000100 807f ff80 0008 fc04 807f ff83 0008 fc04
0000110 807f ff8f 0008 fc04 807f ff93 0008 fc04
```

It's interesting to note that the **hexdump** command first appeared in 4.3 BSD years before USB debuted and more than a decade before USB game pads existed. I could have just as easily used the **od** (octal dump) command, which was part of the original AT& Unix 1 in the early 1970s. The developers could not possibly have imagined that this program would one day be used this way. It was intended for looking at binary files and possibly input from devices of the time, but because of device independence, these commands would never need to be altered in order to work with new devices connected to a Unix system. The ability to use software without modification on devices invented decades later is the mark of intelligent software engineering.

Redirection

Since I/O devices and files are so interchangeable, Unix shells provide a facility called *redirection* to easily interchange them for any command without the command even knowing it.

Redirection depends on the notion of a *file stream*. You can think of a file stream as a hose connecting a program to a particular file or device. Redirection simply disconnects the hose from the default file or device and connects it to another one chosen by the shell user.

Every Unix process has three standard streams that are open from the moment the process is born. The standard streams are normally connected to the terminal, as shown in Table 1.9.

Stream	Purpose	Default Connection
Standard Input	User input	Terminal keyboard
Standard Output	Normal output	Terminal screen
Standard Error	Errors and warnings	Terminal screen

Table 1.9: Standard Streams

Redirection in the shell allows any or all of the three standard streams to be disconnected from the terminal and connected to a file or other I/O device. It uses operators within the commands to indicate which stream(s) to redirect and where. The basic redirection operators shells are shown in Table 1.10.

Operator	Shells	Redirection type
<	All	Standard Input
>	All	Standard Output (overwrite)
>>	All	Standard Output (append)
2>	Bourne-based	Standard Error (overwrite)
2>>	Bourne-based	Standard Error (append)
>&	C shell-based	Standard Output and Standard Error (overwrite)
>>&	C shell-based	Standard Output and Standard Error (append)

Table 1.10: Redirection Operators

Note Memory trick: The arrow in a redirection operator points in the direction of data flow.

Caution
Using output redirection (>, 2>, or >&) in a command will normally overwrite (clobber) the file that you're redirecting to, even if the command itself fails.
Be very careful not to use output redirection accidentally. This most commonly occurs when a careless user meant to use input redirection, but pressed the wrong key.
The moment you press Enter after typing a command containing "> filename", filename will be erased! Remember that the shell performs redirection, not the command, so filename is clobbered before the command even begins running.

If `noclobber` is set for the shell, output redirection to a file that already exists will result in an error. The `noclobber` option can be overridden by appending a ! to the redirection operator in C shell derivatives or a | in Bourne shell derivatives. For example, >! can be used to force overwriting a file in csh or tcsh, and >| can be used in sh, ksh, or bash.

```
shell-prompt: ls > listing.txt         # Overwrite with listing of .
shell-prompt: ls /etc >> listing.txt   # Append listing of /etc
```

Note that redirection is performed by the shell, not the program. In the examples above, the **ls** command sends its output to the standard output. It is unaware that the standard output has been redirected to the file `listing.txt`.

Put another way, `listing.txt` is *not* an argument to the **ls** command. The redirection is handled by the shell, and **ls** runs as if it had been typed as simple:

```
shell-prompt: ls
```

More often than not, we want to redirect both normal output and error messages to the same place. This is why C shell and its derivatives use a combined operator that redirects both at once. The same effect can be achieved with Bourne-shell derivatives using another operator that redirects one stream to another stream. In particular, we redirect the standard output (stream 1) to a file (or device) and at the same time redirect the standard error (stream 2) to stream 1.

```
shell-prompt: find / -name '*.c' > list.txt 2>&1
```

If a program takes input from the standard input, we can redirect input from a file as follows:

```
shell-prompt: command < input-file
```

For example, consider the "bc" (binary calculator) command, an arbitrary-precision calculator which inputs numerical expressions from the standard input and writes the results to the standard output:

```
shell-prompt: bc
3.14159265359 * 4.2 ^ 2 + sqrt(30)
60.89491440932
quit
```

In the example above, the user entered "3.14159265359 * 4.2 ^ 2 + sqrt(30)" and "quit" and the bc program output "60.89491440932". We can place the input shown above in a file using any text editor, such as nano or vi, or by any other means:

```
shell-prompt: cat > bc-input.txt
3.14159265359 * 4.2 ^ 2 + sqrt(30)
quit
(Type Ctrl+d to signal the end of input to the cat command)
shell-prompt: more bc-input.txt
3.14159265359 * 4.2 ^ 2 + sqrt(30)
quit
```

Now that we have the input in a file, we can feed it to the bc command using input redirection instead of retyping it on the keyboard:

```
shell-prompt: bc < bc-input.txt
60.29203070318
```

Special Files in /dev

Although it may seem a little confusing and circular, the standard streams themselves are represented as device files on Unix systems. This allows us to redirect one stream to another without modifying a program, by appending the stream to one of the device files /dev/stdout or /dev/stderr. For example, if a program sends output to the standard output and we want to send it instead to the standard error, we could do the following:

```
    printf "Oops!" >> /dev/stderr
```

If we would like to discard output sent to the standard output or standard error, we can redirect it to /dev/null. For example, to see only error messages (standard error) from myprog, we could do the following:

```
    ./myprog > /dev/null
```

To see only normal output and not error messages, assuming Bourne shell:

```
    ./myprog 2> /dev/null
```

The device /dev/zero is a readable file that produces a stream of zero bytes.

The device /dev/random is a readable file that produces a stream of random integers in binary format.

Pipes

Quite often, we may want to use the output of one program as input to another. Such a thing could be done using redirection, as shown below:

```
shell-prompt: sort names.txt > sorted-names.txt
shell-prompt: uniq < sorted-names.txt > unique-names.txt
```

The same task can be accomplished in one command using a *pipe*. A pipe redirects one of the standard streams, just as redirection does, but to another process instead of to a file or device. In other words, we can use a pipe to send the standard output and/or standard error of one process directly to the standard input of another process.

Example 1.4 Simple Pipe

The command below uses a pipe to redirect the standard output of the **sort** command directly to the standard input of the **uniq**.

```
shell-prompt: sort names.txt | uniq > uniq-names.txt
```

Since a pipe runs multiple commands in the same shell, it is necessary to understand the concept of *foreground* and *background* processes, which are covered in detail in Section 1.18.

Multiple processes can output to a terminal at the same time, although the results would obviously be chaos in most cases.

Only one process can receiving input from the keyboard, however.

The *foreground process* running under a given shell process is defined as the process that receives the input from the standard input device (usually the keyboard). This is the only difference between a foreground process and a background process.

When running a pipeline command, the last process in the pipeline is the foreground process. All others run in the background, i.e. do not use the standard input device inherited from the shell process. Hence, when we run:

```
shell-prompt: find /etc | more
```

It is the **more** command that receives input from the keyboard. The more command has its standard input redirected from the standard output of find, and the standard input of the find command is effectively disabled.

The more command is somewhat special: Since its standard input is redirected from the pipe, it opens another stream to connect to the keyboard so that the user can interact with it, pressing the space bar for another screen, etc.

For piping stderr, the notation is similar to that used for redirection:

Operator	Shells	Pipe stream(s)
|	All	Standard Output to Standard Input
|&	C shell family	Standard Output and Standard Error to Standard Input
2|	Bourne shell family	Standard Error to Standard Input

Table 1.11: Pipe Operators

The entire chain of commands connected by pipes is known as a *pipeline*.

This is such a common practice that Unix has defined the term *filter* to apply to programs that can be used in this way. A filter is any command that can receive input from the standard input and send output to the standard output. Many Unix commands are designed to accept a file names as an arguments, but also to use the standard input and/or standard output if no filename arguments are provided.

Example 1.5 Filters

The **more** command is commonly used as a filter. It can read a file whose name is provided as an argument, but will use the standard input if no argument is provided. Hence, the following two commands have the same effect:

```
shell-prompt: more names.txt
shell-prompt: more < names.txt
```

The only difference between these two commands is that in the first, the **more** receives `names.txt` as a command line argument, opens the file itself (creating a new file stream), and reads from the new stream (not the standard input stream). In the second instance, the shell opens the file and connects the standard input stream of the **more** command to it.
Using the filtering capability of more, we can paginate the output of any command:

```
shell-prompt: ls | more
shell-prompt: find . -name '*.c' | more
shell-prompt: sort names.txt | more
```

We can string any number of commands together using pipes:

```
shell-prompt: cat names.txt | sort | uniq | more
```

One more useful tool worth mentioning is the **tee** command. The **tee** is a simple program that reads from its standard input and writes to both the standard output and to one or more files whose names are provided on the command line. This allows you to view the output of a program on the screen and redirect it to a file at the same time.

```
shell-prompt: ls | tee listing.txt
```

Recall that Bourne-shell derivatives do not have combined operators for redirecting standard output and standard error at the same time. Instead, we redirect the standard output to a file or device, and redirect the standard error to the standard output using `2>&1`.

We can use the same technique with a pipe, but there is one more condition: For technical reasons, the `2>&1` must come before the pipe.

```
shell-prompt: ls | tee listing.txt 2>&1    # Won't work
shell-prompt: ls 2>&1 | tee listing.txt    # Will work
```

The **yes** command produces a stream of y's followed by newlines. It is meant to be piped into a program that prompts for y's or n's in response to yes/no questions, so that the program will receive a yes answer to all of its prompts and run without user input.

```
yes | ./myprog
```

In cases where the response isn't always "yes" we can feed a program any sequence of responses using redirection or pipes. Be sure to add a newline (\n) after each response to simulate pressing the Enter key:

```
./myprog < responses.txt
printf "y\nn\ny\n" | ./myprog
```

1.17.2 Subshells

Commands placed between parentheses are executed in a new child shell process rather than the shell process that received the commands as input.

This can be useful if you want a command to run in a different directory or with altered environment variables, without affecting the current shell process.

```
shell-prompt: (cd /etc; ls)
```

Since the commands above are executed in a new shell process, the shell process that printed "shell prompt: " will not have its current working directory changed. This command has the same net effect as the following:

```
shell-prompt: pushd /etc
shell-prompt: ls
shell-prompt: popd
```

1.17.3 Self-test

1. What does device independence mean?
2. Show a Unix command that could be used to view the data stream sent by a mouse represented as `/dev/mse0`.
3. Name and describe the three standard streams available to all Unix processes.
4. Show the simplest Unix command to accomplish each of the following:
   (a) Save a list of all files in /etc to the file `list.txt`.
   (b) Compile `prog.c` under **bash** using **gcc**, saving error messages to `errors.txt` and normal screen output to `output.txt`.
   (c) Compile `prog.c` under **tcsh** using **gcc**, saving both error messages and normal screen output to `output.txt`.
   (d) Compile `prog.c` under **tcsh** using **gcc**, saving both error messages and normal screen output to `output.txt`. Overwrite `output.txt` even if `noclobber` is set.
   (e) Run the program **./prog1**, causing it to use the file `input.txt` as the standard input instead of the keyboard.
   (f) Compile `prog.c` under **tcsh** using **gcc**, saving both error messages and normal screen output to `output.txt` and sending them to the screen at the same time.
5. Which program in a pipeline runs as the foreground process?
6. How many programs can be included in a single pipeline?
7. What is a filter program?
8. Show a Unix command that will edit all the C program files in the subdirectory Programs, using the **vi** editor.
9. Show a Unix command that runs the command "make" in the directory "./src" without changing the current working directory of the current shell process.

1.18 Process Control

Unix systems provide many tools for managing and monitoring processes that are already running.

Note that these tools apply to local Unix processes only. On distributed systems such as clusters and grids, job management is done using networked schedulers such as HTCondor, Grid Engine, or PBS.

It is possible to have multiple processes running under the same shell session. Such processes are considered either *foreground processes* or *background processes*. The foreground process is simply the process that receives the keyboard input. There can be no more than one foreground process under a given shell session, for obvious reasons.

Note that all processes, both foreground and background, can send output to the terminal at the same time, however. It is up to the user to ensure that output is managed properly and not intermixed.

There are three types of tools for process management, described in the following subsections.

1.18.1 External Commands

Unix systems provide a variety of external commands that monitor or manipulate processes based on their process ID (PID). A few of the most common commands are described below.

ps lists the currently running processes.

```
shell-prompt: ps [-a]     # BSD
shell-prompt: ps [-e]     # SYSV
```

ps is one of the rare commands whose options vary across different Unix systems. There are only two standards to which it may conform, however. The BSD version uses `-a` to indicate that all processes (not just your own) should be shown. System 5 (SYSV) **ps** uses `-e` for the same purpose. Run **man ps** on your system to determine which flags should be used.

kill sends a signal to a process (which may kill the process, but could serve other purposes).

```
shell-prompt: kill [-#] pid
```

The pid (process ID) is determined from the output of **ps**.

The signal number is an integer value following a -, such as `-9`. If not provided, the default signal sent is the TERM (terminate) signal.

Some processes ignore the TERM signal. Such processes can be force killed using the KILL (9) signal.

```
shell-prompt: kill -9 2342
```

Run **man signal** to learn about all the signals that can be issued with **kill**.

```
shell-prompt: ps
  PID  TT  STAT      TIME COMMAND
 41167   0  Is     0:00.25 tcsh
 78555   0  S+     0:01.98 ape unix.dbk
shell-prompt: kill 78555
```

The **killall** command will kill all processes running the program named as the argument. This eliminates the need to find the PID first, and is more convenient for killing multiple processes running the same program.

```
shell-prompt: killall fdtd
```

1.18.2 Special Key Combinations

Ctrl+c sends a terminate signal to the current foreground process. This usually kills the process immediately, although it is possible that some processes will ignore the signal.

Ctrl+z sends a suspend signal to the current foreground process. The process remains in memory, but does not execute further until it receives a resume signal (usually sent by running **fg**).

Ctrl+s suspends output to the terminal. This does not technically control the process directly, but has the effect of blocking any processes that are sending output, since they will stop running until the terminal begins accepting output again.

Ctrl+q resumes output to the terminal if it has been suspended.

1.18.3 Internal Shell Commands and Symbols

jobs lists the processes running under the current shell, but using the shell's job IDs instead of the system's process IDs.

Caution Shell jobs are ordinary processes running on the local system and should not be confused with cluster and grid jobs, which are managed by networked schedulers.

```
shell-prompt: jobs
```

fg brings a background job into the foreground.

```
shell-prompt: fg [%job-id]
```

There cannot be another job already running in the foreground. If no job ID is provided, and multiple background jobs are running, the shell will choose which background job to bring to the foreground. A job ID should always be provided if more than one background job is running.

bg resumes a job suspended by Ctrl+z in the background.

```
shell-prompt: prog
Ctrl+z
shell-prompt: bg
shell-prompt:
```

An & at the end of any command causes the command to be immediately placed in the background. It can be brought to the foreground using **fg** at any time.

```
shell-prompt: command &
```

nice runs a process at a lower than normal priority.

```
shell-prompt: nice command
```

If (and only if) other processes in the system are competing for CPU time, they will get a bigger share than processes run under **nice**.

time runs a command under the scrutiny of the time command, which keeps track of the process's resource usage.

```
shell-prompt: time command
```

There are both internal and external implementations of the time command. Run **which time** to determine which one your shell is configured to use.

nohup allows you to run a command that will continue after you log out. Naturally, all input and output must be redirected away from the terminal in order for this to work.

Bourne shell and compatible:

```
shell-prompt: nohup ./myprogram < inputfile > outputfile 2>&1
```

C shell and compatible:

```
shell-prompt: nohup ./myprogram < inputfile >& outputfile
```

This is often useful for long-running commands and where network connections are not reliable.

There are also free add-on programs such as GNU screen that allow a session to be resumed if it's disrupted for any reason.

1.18.4 Self-test

1. What is a process?
2. What is the difference between a foreground process and a background process?
3. How many foreground processes can be running at once under a single shell process? Why?
4. How many background processes can be running at once under a single shell process? Why?
5. Show the simplest Unix command that will accomplish each of the following:

(a) List all processes currently running.

(b) List processes owned by you.

(c) Kill the process with ID 7243.

(d) Kill all processes running the program **netsim**.

(e) Kill the process with ID 7243 after the first attempt failed.

6. How do you perform each of the following tasks?

(a) Kill the current foreground process.

(b) Suspend the current foreground process.

(c) Resume a suspended process in the foreground.

(d) Resume a suspended process in the background.

(e) Start a new process, placing it in the background immediately.

(f) Suspend terminal output for a process without suspending the process itself.

(g) Resume suspended terminal output.

(h) List the currently running jobs as seen by the shell.

(i) Return job #2 to the foreground.

(j) Run the program **netsim** at a reduced priority so that other processes will respond faster.

(k) Run the program **netsim** and report the CPU time used when it finishes.

1.19 Remote Graphics

Most users will not need to run graphical applications on a remote Unix system.. If you know that you will need to use a graphical user interface with your research software, or if you want to use a graphical editor such as gedit or emacs on over the network, read on. Otherwise, you can skip this section for now.

Unix uses a networked graphics interface called the X Window system. It is also sometimes called simply X11 for short. (X11 is the latest major version of the system.) X11 allows programs running on a Unix system to display graphics on the local screen or the screen of another Unix system on the network. The programs are called *clients*, and they display graphical output by sending commands to the *X11 server* on the machine where the output is to be displayed. Hence, your local computer must be running an X11 server in order to display Unix graphics, regardless of whether the client programs are running on your machine or another.

Some versions of OS X had the Unix X11 API included, while others need it installed separately. At the time of this writing, X11 on the latest OS X is provided by the XQuartz project, described at https://support.apple.com/en-us/HT201341. You will need to download and install this free package to enable X11 on your Mac.

1.19.1 Configuration Steps Common to all Operating Systems

Modern Unix systems such as BSD, Linux, and Mac OS X have most of the necessary tools and configuration in place for running remote graphical applications.

However, some additional steps may be necessary on your computer to allow remote systems to access your display. This applies to *all* computers running an X11 server, regardless of operating system. Some additional steps that may be necessary for Cygwin systems are discussed in Section 1.19.2.

If you want to run graphical applications on a remote computer over an ssh connection, you will need for forward your local display to the remote system. This can be done for a single ssh session by providing the `-X` flag:

```
shell-prompt: ssh -X joe@login.peregrine.hpc.uwm.edu
```

This causes the **ssh** command to inform the remote system that X11 graphical output should be sent to your local display through the **ssh** connection. (This is called SSH tunneling.)

Caution Allowing remote systems to display graphics on your computer can pose a security risk. For example, a remote user may be able to display a false login window on your computer in order to collect login and password information.

If you want to forward X11 connections to all remote hosts for all users on the local system, you can enable X11 forwarding in your `ssh_config` file (usually found in /etc or /etc/ssh) by adding the following line:

```
    ForwardX11 yes
```

Caution Do this only if you are prepared to trust all users of your local system as well as all remote systems to which they might connect.

Some X11 programs require additional protocol features that can pose more security risks to the client system. If you get an error message containing "Invalid MIT-MAGIC-COOKIE" when trying to run a graphical application over an **ssh** connection, try using the `-Y` flag with **ssh** to open a *trusted* connection.

```
shell-prompt: ssh -Y joe@login.peregrine.hpc.uwm.edu
```

You can establish trusted connections to *all* hosts by adding the following to your `ssh_config` file:

```
    ForwardX11Trusted yes
```

Caution This is generally considered a bad idea, since it states that every host connected to from this computer to should be trusted completely. Since you don't know in advance what hosts people will connect to in the future, this is a huge leap of faith.

If you are using ssh over a slow connection, such as home DSL/cable, and plan to use X11 programs, it can be very helpful to enable compression, which is enabled by the `-C` flag. Packets are then compressed before being sent over the wire and decompressed on the receiving end. This adds more CPU load on both ends, but reduces the amount of data flowing over the network and may significantly improve the responsiveness of a graphical user interface. Run **man ssh** for details.

```
shell-prompt: ssh -YC joe@login.peregrine.hpc.uwm.edu
```

1.19.2 Graphical Programs on Windows with Cygwin

It is possible for Unix graphical applications on the remote Unix machine to display on a Windows machine, but this will require installing additional Cygwin packages and performing a few configuration steps on your computer in addition to those discussed in Section 1.19.1.

Installation

You will need to install the x11/xinit and x11/xhost packages using the Cygwin setup utility. This will install an X11 server on your Windows machine.

Configuration

After installing the Cygwin X packages, there are additional configuration steps:

1. Create a working `ssh_config` file by running the following command from a Cygwin shell window:

   ```
   shell-prompt: cp /etc/defaults/etc/ssh_config /etc
   ```

2. Then, using your favorite text editor, update the new `/etc/ssh_config` as described in Section 1.19.1.

3. Add the following line to .bashrc or .bash_profile (in your home directory):

   ```
       export DISPLAY=":0.0"
   ```

   Cygwin uses bash for all users by default. If you are using a different shell, then edit the appropriate start up script instead of .bashrc or .bash_profile.

   This is not necessary when running commands from an xterm window (which is launched from Cygwin-X), but *is* necessary if you want to launch X11 applications from a Cygwin bash terminal which is part of the base Cygwin installation, and not X11-aware.

Start-up

To enable X11 applications to display on your Windows machine, you need to to start the X11 server on Windows by clicking Start → All Programs → Cygwin-X → XWin Server. The X server icon will appear in your Windows system tray to indicate that X11 is running. You can launch an xterm terminal emulator from the system tray icon, or use the Cygwin bash terminal, assuming that you have set your DISPLAY variable.

1.20 Where to Learn More

There is a great deal of information available on the web. There are also many length books dedicated to Unix, which can provide more detail than this tutorial.

If you simply want to know what commands are available on your system, list the bin directories!

```
shell-prompt: ls /bin /usr/bin /usr/local/bin | more
```

Chapter 2

Unix Shell Scripting

Before You Begin
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 1) and the Unix shell (Section 1.4.3).

2.1 What is a Shell Script?

A shell script is essentially a file containing a sequence of Unix commands. A script is a type of program, but is distinguished from other programs in that it represents programming at a higher level.

While a typical program is largely made of calls to subprograms, a script contains invocations of whole programs.

In other words, a script is a way of automating the execution of multiple separate programs in sequence.

The Unix command-line structure was designed to be convenient for both interactive use and for programming in scripts. Running a Unix command is much like calling a subprogram. The difference is just syntax. A subprogram call in C encloses the arguments in parenthesis and separates them with commas:

```
function_name(arg1,arg2,arg3);
```

A Unix command is basically the same, except that it uses spaces instead of parenthesis and commas:

```
command_name arg1 arg2 arg3
```

2.1.1 Self-test

1. What is a shell script?
2. How are Unix commands similar to and different from subprogram calls in a language like C?

2.2 Scripts vs Programs

It is important to understand the difference between a "script" and a "real program", and which languages are appropriate for each.

Scripts tend to be small (no more than a few hundred or a few thousand lines of code) and do not do any significant computation of their own.

Instead, scripts run "real programs" to do most of the computational work. The job of the script is simply to automate and document the process of running programs.

As a result, scripting languages do not need to be efficient and are generally interpreted rather than compiled. (Interpreted language programs run an order of magnitude or more slower than equivalent compiled programs, unless most of their computation is done by built-in, compiled subprograms.)

Real programs may be quite large and may implement complex computational algorithms. Hence, they need to be fast and as a result are usually written in compiled languages.

If you plan to use exclusively pre-existing programs such as Unix commands and/or add-on application software, and need only automate the execution of these programs, then you need to write a script and should choose a good scripting language.

If you plan to implement your own algorithm(s) that may require a lot of computation, then you need to write a program and should select an appropriate compiled programming language.

2.2.1 Self-test

1. How do scripts differ from programs written in languages like C or Fortran?
2. Why would it not be a good idea to write a matrix multiplication program as a Bourne shell script?

2.3 Why Write Shell Scripts?

2.3.1 Efficiency and Accuracy

Any experienced computer user knows that we often end up running basically the same sequence of commands many times over. Typing the same sequence of commands over and over is a waste of time and highly prone to errors.

All Unix shells share a feature that can help us avoid this repetitive work: They don't care where their input comes from.

It is often said that the Unix shell reads commands from the keyboard and executes them. This is not really true. The shell reads commands from *any input source* and executes them. The keyboard is just one common input source that can be used by the shell. Ordinary files are also very commonly used as shell input.

Recall from Chapter 1 that Unix systems employ device independence, which means that any Unix program that reads from a keyboard can also read the same input from a file or any other input device.

Hence, if we're going to run the same sequence of commands more than once, we don't need to retype the sequence each time. The shell can read the commands from anywhere, not just from the keyboard. We can put those commands into a text file *once* and tell the shell to read the commands from the file, which is much easier than typing them all again.

Rule of Thumb If you might have to do it again, script it.

In theory, Unix commands could also be piped in from another program or read from any other device attached to a Unix system, although in practice, they usually come from the keyboard or a script file.

Self-test

1. Describe two reasons for writing shell scripts.
2. Are there Unix commands that you can run interactively, but not from a shell script? Explain.
3. What feature of Unix makes shell scripts so convenient to implement?
4. What is a good rule of thumb for deciding whether to write a shell script?

2.3.2 Documentation

There is another very good reason for writing shell scripts in addition to saving us a lot of redundant typing:

A shell script is the ultimate documentation of the work we have done on a computer.

By writing a shell script, we record the exact sequence of commands needed to reproduce results, in perfect detail. Hence, the script serves a dual purpose of automating and documenting our processes.

Developing a script has a ratchet effect on your knowledge. Once you add a command to a script, you will never forget how to use it for that task.

Rule of Thumb A Unix user should never find themselves trying to remember how they did something. Script it the first time...

Clear documentation of our work flow is important in order to justify research funding and to be able to reproduce results months or years later.

Imagine that we instead decided to run our sequence of commands manually and document what we did in a word processor. First, we'd be typing everything twice: Once at the shell prompt and again into the document.

The process of typing the same commands each time would be painful enough, but to document it in detail while we do it would be distracting. We'd also have to remember to update the document every time we type a command differently. This is hard to do when we're trying to focus on getting results.

Writing a shell script allows us to stay focused on perfecting the process. Once the script is finished and working perfectly, we have the process perfectly documented. We can and should add comments to the script to make it more readable, but even without comments, the script itself preserves the process in detail.

Many experienced users will *never* run a processing command from the keyboard. Instead, they *only* put commands into a script and run and re-run the script until it's finished.

Self-test

1. Describe another good reason for writing shell scripts.
2. Why is it so important to document the sequence of commands used?

2.3.3 Why Unix Shell Scripts?

There are many scripting languages to choose from, including those used on Unix systems, like Bourne shell, C shell, Perl, Python, etc., as well as some languages confined to other platforms like Visual Basic (Microsoft Windows only) and AppleScript (Apple only).

Note that the Unix-based scripting languages can be used on *any* platform, *including* Microsoft Windows (with Cygwin, for example) and Apple's Mac OS X, which is Unix-compatible out of the box.

Once you learn to write Unix shell scripts, you're prepared to do scripting on any computer, without having to learn another language.

Self-test

1. What are two advantages of writing Unix shell scripts instead of using a scripting language such as Visual Basic or AppleScript?

2.3.4 Self-test

1. Describe three reasons for writing shell scripts instead of running commands from the keyboard.

2.4 Which Shell?

2.4.1 Common Shells

When writing a shell script, there are essentially two scripting languages to choose from: Bourne shell and C shell. These were the first two popular shells for Unix, and all common shells that have come since are compatible with one or the other.

The most popular new shells are Bourne Again shell (bash), which is an extension of Bourne shell, Korn shell (ksh), which is another extension of Bourne shell, Z-shell, a very sophisticated extension of Bourne shell, and T-shell (TENEX C shell, tcsh), which is an extended C shell.

- Bourne shell family
  - Bourne shell (sh)
  - Bourne-again shell (bash)
  - Korn shell (ksh)
  - Z-shell (zsh)
- C shell family
  - C shell (csh)
  - T-shell (tcsh)

Both Bourne shell and C shell have their own pros and cons. C shell syntax is cleaner, more intuitive, and more similar to the C programming language (hence the name C shell). However, C shell lacks some features such as subprograms (although C shell scripts can run other C shell scripts, which is arguably a better approach in many situations).

Bourne shell is used almost universally for Unix system scripts, while C shell is fairly popular in scientific research.

Note Every Unix system has a Bourne shell in `/bin/sh`. Hence, using vanilla Bourne shell (not bash, ksh, or zsh) for scripts maximizes their portability by ensuring that they will run on any Unix system without the need to install any additional shells.

If your script contains only external commands, then it actually won't matter which shell runs it. However, most scripts utilize the shell's internal commands, control structures, and features like redirection and pipes, which differ among shells.

More modern shells such as bash, ksh, and tcsh, are backward-compatible with Bourne shell or C shell, but add additional scripting constructs in addition to convenient interactive features. The details are beyond the scope of this text. For full details, see the documentation for each shell.

2.4.2 Self-test

1. What is one advantage of Bourne shell over C shell?
2. What is one advantage of C shell over Bourne shell?

2.5 Writing and Running Shell Scripts

A shell script is a simple text file and can be written using any Unix text editor. Some discussion of Unix text editors can be found in Section 1.11.4.

Caution Recall from Section 1.10.1 that Windows uses a slightly different text file format than Unix. Hence, editing Unix shell scripts in a Windows editor can be problematic. Users are advised to do all of their editing on a Unix machine rather than write programs and scripts on Windows and transfer them to Unix.

Shell scripts often contain very complex commands that are wider than a typical terminal window. A command can be continued on the next line by typing a backslash (\) immediately before pressing **Enter**. This feature is present in all Unix shells. Of course, it can be used on an interactive CLI as well, but is far more commonly used in scripts to improve readability.

```
printf "%s %s\n" "This command is too long to fit in a single 80-column" \
    "terminal window, so we break it up with a backslash."
```

It's not a bad idea to name the script with a file name extension that matches the shell it uses. This just makes it easier to see which shell each of your script files use. Table 2.1 shows conventional file name extensions for the most common shells. However, if a script is to be installed into the PATH so that it can be used as a regular command, it is usually given a name with no extension. Most users would rather type "cleanup" than "cleanup.bash".

Like all programs, shell scripts should contain comments to explain what the commands in it are doing. In all Unix shells, anything from a '#' character to the end of a line is considered a comment and ignored by the shell.

```
# Print the name of the host running this script
hostname
```

Shell	Extension
Bourne Shell	.sh
C shell	.csh
Bourne Again Shell	.bash
T-shell	.tcsh
Korn Shell	.ksh
Z-shell	.zsh

Table 2.1: Conventional script file name extensions

Practice Break

Using your favorite text editor, enter the following text into a file called `hello.sh`.

1. The first step is to create the file containing your script, using any text editor, such as nano:

   ```
   shell-prompt: nano hello.sh
   ```

   Once in the text editor, add the following text to the file:

   ```
   printf "Hello!\n"
   printf "I am a script running on a computer called `hostname`\n"
   ```

   After typing the above text into the script, save the file and exit the editor. If you are using nano, the menu at the bottom of the screen tells you how to save (write out, Ctrl+o) and exit (Ctrl+x).

2. Once we've written a script, we need a way to run it. A shell script is simply input to a shell program. Like many Unix programs, shells take their input from the standard input by default. We could, therefore, use redirection to make it read the file via standard input:

   ```
   shell-prompt: sh < hello.sh
   ```

Sync-point: Instructor: Make sure everyone in class succeeds at this exercise before moving on.

Since the shell normally reads commands from the standard input, the above command will "trick" **sh** into reading its commands from the file `hello.sh`.

However, Unix shells and other scripting languages provide a more convenient method of indicating what program should interpret them. If we add a special comment, called a *shebang line* to the top of the script file and make the file executable using

chmod, the script can be executed like a Unix command. We can then simply type its name at the shell prompt, and another shell process will start up and run the commands in the script. If the directory containing such a script is included in $PATH, then the script can be run from any current working directory, just like **ls**, **cp**, etc.

The shebang line consists of the string "#!" followed by the full path name of the command that should be used to execute the script, or the path **/usr/bin/env** followed by the name of the command. For example, both of the following are valid ways to indicate a Bourne shell (sh) script, since `/bin/sh` is the Bourne shell command.

```
#!/bin/sh
```

```
#!/usr/bin/env sh
```

When you run a script as a command, by simply typing its file name at the Unix command-line, a new shell process is created to interpret the commands in the script. The shebang line specifies which program is invoked for the new shell process that runs the script.

Note The shebang line must begin at the very first character of the script file. There cannot even be blank lines above it or white space to the left of it. The "#!" is an example of a *magic number*. Many files begin with a 16-bit (2-character) code to indicate the type of the file. The "#!" indicates that the file contains some sort of interpreted language program, and the characters that follow will indicate where to find the interpreter.

The **/usr/bin/env** method is used for add-on shells and other interpreters, such as Bourne-again shell (bash), Korn shell (ksh), and Perl (perl). These interpreters may be installed in different directories on different Unix systems. For example, bash is typically found in **/bin/bash** on Linux systems, **/usr/local/bin/bash** on FreeBSD systems, and **/usr/pkg/bin/bash** on NetBSD. The T-shell is found in **/bin/tcsh** on FreeBSD and CentOS Linux and in **/usr/bin/tcsh** on Ubuntu Linux.

The **env** command is found in `/usr/bin/env` on virtually all Unix systems. Hence, this provides a method for writing shell scripts that are portable across Unix systems (i.e. they don't need to be modified to run on different Unix systems).

Note
Every script or program should be tested on more than one platform (e.g. BSD, Cygwin, Linux, Mac OS X, etc.) immediately, in order to shake out bugs before they cause problems.
The fact that a program works fine on one operating system and CPU does not mean that it's free of bugs.
By testing it on other operating systems, other hardware types, and with other compilers or interpreters, you will usually expose bugs that will seem obvious in hindsight.
As a result, the software will be more likely to work properly when time is critical, such as when there is an imminent deadline approaching and no time to start over from the beginning after fixing bugs. Encountering software bugs at times like these is very stressful and usually easily avoided by testing the code on multiple platforms in advance.

Bourne shell (sh) is present and installed in `/bin` on all Unix-compatible systems, so it's safe to hard-code #!/bin/sh is the shebang line.

C shell (csh) is not included with all systems, but is virtually always in /bin if present, so it is generally safe to use #!/bin/csh as well.

For all other interpreters it's best to use #!/usr/bin/env.

```
#!/bin/sh           (OK and preferred)
```

```
#!/bin/csh          (Generally OK)
```

```
#!/bin/bash         (Bad idea: Not portable)
```

```
#!/usr/bin/perl     (Bad idea: Not portable)
```

```
#!/usr/bin/python    (Bad idea: Not portable)
```

```
#!/bin/tcsh          (Bad idea: Not portable)
```

```
#!/usr/bin/env bash      (This is portable)
```

```
#!/usr/bin/env tcsh      (This is portable)
```

```
#!/usr/bin/env perl      (This is portable)
```

```
#!/usr/bin/env python    (This is portable)
```

Even if your system comes with `/bin/bash` and you don't intend to run the script on any other system, using `/usr/bin/env` is still a good idea, because you or someone else may want to use a newer version of bash that's installed in a different location. The same applies to other scripting languages such as C-shell, Perl, Python, etc.

Example 2.1 A Simple Bash Script

Suppose we want to write a script that is always executed by bash, the Bourne Again Shell. We simply need to add a shebang line indicating the path name of the **bash** executable file.

```
shell-prompt: nano hello.sh
```

Enter the following text in the editor. Then save the file and exit back to the shell prompt.

```
#!/usr/bin/env bash

# A simple command in a shell script
printf "Hello, world!\n"
```

Now, make the file executable and run it:

```
shell-prompt: chmod a+rx hello.sh    # Make the script executable
shell-prompt: ./hello.sh             # Run the script as a command
```

Example 2.2 A Simple T-shell Script

Similarly, we might want to write a script that is always executed by tcsh, the TENEX C Shell. We simply need to add a shebang line indicating the path name of the **tcsh** executable file.

```
shell-prompt: nano hello.tcsh
```

```
#!/usr/bin/env tcsh

# A simple command in a shell script
printf "Hello, world!\n"
```

```
shell-prompt: chmod a+rx hello.tcsh    # Make the script executable
shell-prompt: ./hello.tcsh             # Run the script as a command
```

Note Many of the Unix commands you use regularly may actually be scripts rather than binary programs.

Note
There may be cases where you cannot make a script executable. For example, you may not own it, or the file system may not allow executables, to prevent users from running programs where they shouldn't.
In these cases, we can simply run the script as an argument to an appropriate shell. For example:

```
shell-prompt: sh hello.sh
shell-prompt: bash hello.bash
shell-prompt: csh hello.csh
shell-prompt: tcsh hello.tcsh
shell-prompt: ksh hello.ksh
```

Note also that the shebang line in a script is ignored when you explicitly run a shell this way. The content of the script will be interpreted by the shell that you have manually invoked, regardless of what the shebang line says.

Scripts that you create and intend to use regularly can be placed in your PATH, so that you can run them from anywhere. A common practice among Unix users is to create a directory called `~/bin`, and configure the login environment to that this directory is always in the PATH. Programs and scripts placed in this directory can then be used like any other Unix command, without typing the full path name.

2.5.1 Self-test

1. What tools can be used to write shell scripts?
2. Is it a good idea to write Unix shell scripts under Windows? Why or why not?
3. After creating a new shell script, what must be done in order to make it executable like a Unix command?
4. What is a shebang line?
5. What does the shebang line look like for a Bourne shell script? A Bourne again shell script? Explain the differences.

2.6 Shell Start-up Scripts

Each time you log into a Unix machine or start a new shell (e.g. when you open a new terminal), the shell process runs one or more special scripts called *start up scripts*. Some common start up scripts:

Script	Shells that use it	Executed by
/etc/profile, ~/.profile	Bourne shell family	Login shells only
File named by $ENV (typically .shrc or .shinit)	Bourne shell family	All interactive shells (login and non-login)
~/.bashrc	Bourne-again shell only	All interactive shells (login and non-login)
~/.bash_profile	Bourne-again shell only	Login shells only
~/.kshrc	Korn shell	All interactive shells (login and non-login)
/etc/csh.login, ~/.login	C shell family	Login shells only
/etc/csh.cshrc, ~/.cshrc	C shell family	All shell processes
~/.tcshrc	T-shell	All shell processes

Table 2.2: Shell Start Up Scripts

Note
Non-interactive Bourne-shell family shell processes, such as those used to execute shell scripts, do not execute any start up scripts. Hence, Bourne shell family scripts are not affected by start up scripts.
In contrast, all C shell script processes execute ~/.cshrc if it exists. Hence, C shell family scripts are affected by ~/.cshrc. You can override this in C-shell scripts by invoking the shell with `-f` as follows:

```
#!/bin/csh -f
```

The man page for your shell has all the details about which start up scripts are invoked and when.

Start up scripts are used to configure your PATH and other environment variables, set your shell prompt and other shell features, create aliases, and anything else you want done when you start a new shell.

One of the most common alterations users make to their start up script is editing their PATH to include a directory containing their own programs and scripts. Typically, this directory is named `~/bin`, but you can name it anything you want.

To set up your own ~/bin to store your own scripts and programs, do the following:

1. shell-prompt: mkdir ~/bin
2. Edit your start up script and add ~/bin to the PATH.

   If you're using Bourne-again shell, you can add `~/bin` to your PATH for login shells only by adding the following to your `.bashrc`:

   ```
   PATH=${PATH}:${HOME}/bin
   export PATH
   ```

   If you're using T-shell, add the following to your `.cshrc` or `.tcshrc`:

   ```
   setenv PATH ${PATH}:~/bin
   ```

   If you are using a different shell, see the documentation for your shell to determine the correct start up script and command syntax.

   Caution
   Adding ~/bin before (left of) ${PATH} will cause your shell to look in ~/bin before looking in the standard directories such as /bin and /usr/bin. Hence, if a binary or script in ~/bin has the same name as another command, the one in ~/bin will override it. This is considered a security risk, since users could be tricked into running a Trojan-horse **ls** or other common command if care is not taken to protect `~/bin` from modification.
   Hence, adding to the tail (right side) of PATH is usually recommended, especially for inexperienced users.

3. Update the PATH in your current shell process by sourcing the start up script, or by logging out and logging back in.

There is no limit to what your start up scripts can do, so you can use your imagination freely and find ways to make your Unix shell environment easier and more powerful.

2.6.1 Self-test

1. What is the purpose of a shell start-up script?
2. What are the limitations on what a start-up script can do compared to a normal script?

2.7 Sourcing Scripts

In some circumstances, we might not want a script to be executed by a separate shell process.

For example, suppose we just made some changes to our .cshrc or .bashrc file that would affect PATH or some other important environment variable.

If we run the start up script by typing **~/.cshrc** or **~/.bashrc**, a new shell process will be started which will execute the commands in the script and then terminate. The shell you are using, which is the parent process, will be unaffected, since parent processes do not inherit environment from their children.

In order to make the "current" shell process run the commands in a script, we must *source* it. This is done using the internal shell command **source** in all shells except Bourne shell, which uses ".". Most Bourne-derived shells support both "." and "source".

Hence, to source `.cshrc`, we would run

```
shell-prompt: source ~/.cshrc
```

To source `.bashrc`, we would run

```
shell-prompt: source ~/.bashrc
```

or

```
. ~/.bashrc
```

To source `.shrc` from a basic Bourne shell, we would have to run

```
. ~/.shrc
```

2.7.1 Self-test

1. What is sourcing?
2. When would we want to source a script?

2.8 Scripting Constructs

Although Unix shells make no distinction between commands entered from the keyboard and those input from a script, there are certain shell features that are meant for scripting and not convenient or useful to use interactively.

Many of these features will be familiar to anyone who has done any computer programming. They include constructs such as comments, conditionals and loops.

The following sections provide a very brief introduction to shell constructs that are used in scripting, but generally not used on the command line.

2.9 Strings

A string constant in a shell script is anything enclosed in single quotes ('this is a string') or double quotes ("this is also a string").

Unlike most programming languages, text in a shell scripts that is not enclosed in quotes and does not begin with a '$' or other special character is also interpreted as a string constant. Hence, all of the following are the same:

```
shell-prompt: ls /etc
shell-prompt: ls "/etc"
shell-prompt: ls '/etc'
```

However, something contains white space (spaces or tabs), then it will be seen as multiple separate strings. The last example below will not work properly, since 'Program' and 'Files' are seen as separate arguments:

```
shell-prompt: cd 'Program Files'
shell-prompt: cd "Program Files"
shell-prompt: cd Program Files
```

Note
Special sequences such as '\n' must be enclosed in quotes or escaped, otherwise the '\' is seen as escaping the 'n'.

```
Hello\n != "Hello\n"
"Hello\n" = 'Hello\n' = Hello\\n
```

2.10 Output

Output commands are only occasionally useful at the interactive command line. We may sometimes use them to take a quick look at a variable such as $PATH.

```
shell-prompt: echo $PATH
```

Output commands are far more useful in scripts, and are used in the same ways as output statements in any programming language.

The **echo** command is commonly used to output something to the terminal screen:

```
shell-prompt: echo 'Hello!'
Hello!
shell-prompt: echo $PATH
/usr/local/bin:/home/bacon/scripts:/home/bacon/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local ↩
    /sbin
```

However, **echo** should be avoided, since it is not portable across different shells and even the same shell on different Unix systems. There are many different implementations of **echo** commands, some internal to the shell and some external programs. Different implementations of **echo** use different command-line flags and special characters to control output formatting.

In addition, the output formatting capabilities of **echo** commands are extremely limited.

The **printf** command supersedes **echo**. It has a rich set of capabilities and is specified in the POSIX.2 standard, so its behavior is the same on all Unix systems.

The **printf** command is an external command, so it is independent of which shell you are using.

The functionality of **printf** closely matches that of the `printf()` function in the standard C library. It recognizes special characters such as '\n' (line feed), '\t' (tab), '\r' (carriage return), etc. and can print numbers in different bases.

```
shell-prompt: printf 'Hello!\n'
Hello!
```

The basic syntax of a printf command is as follows:

```
printf format-string argument [argument ...]
```

The format-string contains literal text and a *format specifier* to match each of the arguments that follows.

Each format specifier begins with a '%' and is followed by a symbol indicating the format in which to print the argument.

The printf command also recognized most of the same special character sequences as the C printf() function:

```
printf "%s %d %o\n" 10 10 10
```

Specifier	Output
%s	String
%d	Decimal number
%o	Octal number

Table 2.3: Printf Format Specifiers

Sequence	Meaning
\n	Newline (move down to next line)
\r	Carriage Return (go to beginning of current line)
\t	Tab (go to next tab stop)

Table 2.4: Special Character Sequences

Output:

```
10 10 12
```

There are many other format specifiers and special character sequences. For complete information, run **man printf**.

To direct printf output to the standard error instead of the standard output, we simply take advantage of device independence and use redirection:

```
printf 'Hello!\n' >> /dev/stderr
```

Practice Break

Sync-point: Instructor: Make sure everyone in class succeeds at this exercise before moving on.

Write a shell script containing the printf statement above and run it. Write the same script using two different shells, such as Bourne shell and C shell. What is the difference between the two scripts?

2.10.1 Self-test

1. What are the advantages of **printf** over the **echo** command?
2. Does printf work under all shells? Why or why not?

2.11 Shell and Environment Variables

Variables are essential to any programming language, and scripting languages are no exception. Variables are useful for user input, control structures, and for giving short names to commonly used values such as long path names.

Most programming languages distinguish between variables and constants, but in shell scripting, we use variables for both.

Shell processes have access to two separate sets of string variables.

Recall from Section 1.15 that every Unix process has a set of string variables called the *environment*, which are handed down from the parent process in order to communicate important information.

For example, the TERM variable, which identifies the type of terminal a user us using, is used by programs such as top, vi, nano, more, and other programs that need to manipulate the terminal screen (move the cursor, highlight certain characters, etc.) The TERM environment variable is usually set by the shell process so that all of the shell's child processes (those running vi, nano, etc.) will inherit the variable.

Unix shells also keep another set of variables that are not part of the environment. These variables are used only for the shell's purpose and are not handed down to child processes.

There are some special shell variables such as "prompt" and "PS1" (which control the appearance of the shell prompt in C shell and Bourne shell, respectively).

Most shell variables, however, are defined by the user for use in scripts, just like variables in any other programming language.

2.11.1 Assignment Statements

In all Bourne Shell derivatives, a shell variable is created or modified using the same simple syntax:

```
varname=value
```

Caution In Bourne shell and its derivatives, there can be no space around the '='. If there were, the shell would think that 'varname' is a command, and '=' and 'value' are arguments.

```
bash-4.2$ name = Fred
bash: name: command not found
bash-4.2$ name=Fred
bash-4.2$ printf "$name\n"
Fred
```

When assigning a string that contains white space, it must be enclosed in quotes or the white space characters must be escaped:

```
#!/usr/bin/env bash

name=Joe Sixpack    # Error
name="Joe Sixpack"  # OK
name=Joe\ Sixpack   # OK
```

C shell and T-shell use the **set** command for assigning variables.

```
#!/bin/csh

set name="Joe Sixpack"
```

Caution Note that Bourne family shells also have a **set** command, but it has a completely different meaning, so take care not to get confused. The Bourne **set** command is used to set shell command-line options, not variables.

Unlike some languages, shell variables need not be declared before they are assigned a value. Declaring variables is unnecessary, since there is only one data type in shell scripts.

All variables in a shell script are character strings. There are no integers, Booleans, enumerated types, or floating point variables, although there are some facilities for interpreting shell variables as integers, assuming they contain only digits.

If you *must* manipulate real numbers in a shell script, you could accomplish it by piping an expression through **bc**, the Unix arbitrary-precision calculator:

```
printf "scale=5\n243.9 * $variable\n" | bc
```

Such facilities are very inefficient compared to other languages, however, partly because shell languages are interpreted, not compiled, and partly because they must convert each string to a number, perform arithmetic, and convert the results back to a string. Shell scripts are meant to automate sequences of Unix commands and other programs, not perform numerical computations.

In Bourne shell family shells, environment variables are set by first setting a shell variable of the same name and then *exporting* it.

```
TERM=xterm
export TERM
```

Modern Bourne shell derivatives such as bash (Bourne Again Shell) can do it in one line:

```
export TERM=xterm
```

Note Exporting a shell variable permanently tags it as exported. Any future changes to the variable's value will automatically be copied to the environment. This type of linkage between two objects is very rare in programming languages: Usually, modifying one object has no effect on any other.

C shell derivatives use the setenv command to set environment variables:

```
setenv TERM xterm
```

Caution Note that unlike the 'set' command, setenv requires white space, not an '=', between the variable name and the value.

2.11.2 Variable References

To reference a shell variable or an environment variable in a shell script, we must precede its name with a '$'. The '$' tells the shell that the following text is to be interpreted as a variable name rather than a string constant. The variable reference is then *expanded*, i.e. replaced by the value of the variable. This occurs anywhere in a command except inside a string bounded by single quotes or following an escape character (\), as explained in Section 2.9.

These rules are basically the same for all Unix shells.

```
#!/usr/bin/env bash

name="Joe Sixpack"
printf "Hello, name!\n"      # Not a variable reference!
printf "Hello, $name!\n"     # References variable "name"
```

Output:

```
Hello, name!
Hello, Joe Sixpack!
```

Practice Break
Type in and run the following scripts:

```
#!/bin/sh

first_name="Bob"
last_name="Newhart"
printf "%s %s is the man.\n" $first_name $last_name
```

CSH version:

```
#!/bin/csh

set first_name="Bob"
set last_name="Newhart"
printf "%s %s is the man.\n" $first_name $last_name
```

Note
If both a shell variable and an environment variable with the same name exist, a normal variable reference will expand the shell variable.
In Bourne shell derivatives, a shell variable and environment variable of the same name always have the same value, since exporting is the only way to set an environment variable. Hence, it doesn't really matter which one we reference.
In C shell derivatives, a shell variable and environment variable of the same name can have different values. If you want to reference the environment variable rather than the shell variable, you can use the printenv command:

```
Darwin heron bacon ~ 319: set name=Sue
Darwin heron bacon ~ 320: setenv name Bob
Darwin heron bacon ~ 321: echo $name
Sue
Darwin heron bacon ~ 322: printenv name
Bob
```

There are some special C shell variables that are automatically linked to environment counterparts. For example, the shell variable path is always the same as the environment variable PATH. The C shell man page is the ultimate source for a list of these variables.

If a variable reference is immediately followed by a character that could be part of a variable name, we could have a problem:

```
#!/usr/bin/env bash

name="Joe Sixpack"
printf "Hello to all the $names of the world!\n"
```

Instead of printing "Hello to all the Joe Sixpacks of the world", the printf will fail because there is no variable called "names". In Bourne Shell derivatives, non-existent variables are treated as empty strings, so this script will print "Hello to all the of the world!". C shell will complain that the variable "names" does not exist.

We can correct this by delimiting the variable name in curly braces:

```
#!/usr/bin/env bash

name="Joe Sixpack"
printf "Hello to all the ${name}s of the world!\n"
```

This syntax works for all shells.

2.11.3 Using Variables for Code Quality

Another very good use for shell variables is in eliminating redundant string constants from a script:

```
#!/usr/bin/env bash

output_value=`myprog`
printf "$output_value\n" >> Run2/Output/results.txt
more Run2/Output/results.txt
cp Run2/Output/results.txt latest-results.txt
```

If for any reason the relative path `Run2/Output/results.txt` should change, then you'll have to search through the script and make sure that all instances are updated. This is a tedious and error-prone process, which can be avoided by using a variable:

```
#!/usr/bin/env bash

output_value=`myprog`
output_file="Run2/Output/results.txt"
printf "$output_value\n" >> $output_file
more $output_file
cp $output_file latest-results.txt
```

In the second version of the script, if the path name of `results.txt` changes, then only one change must be made to the script.

Avoiding redundancy is one of the primary goals of any good programmer.

In a more general programming language such as C or Fortran, this role would be served by a constant, not a variable. However, shells do not support constants, so we use a variable for this.

In most shells, a variable can be marked read-only in an assignment to prevent accidental subsequent changes. Bourne family shells use the **readonly** command for this, while C shell family shells use **set -r**.

```
#!/bin/sh

readonly output_value=`myprog`
printf "$output_value\n" >> Run2/Output/results.txt
more Run2/Output/results.txt
cp Run2/Output/results.txt latest-results.txt
```

```
#!/bin/csh

set -r output_value=`myprog`
printf "$output_value\n" >> Run2/Output/results.txt
more Run2/Output/results.txt
cp Run2/Output/results.txt latest-results.txt
```

2.11.4 Output Capture

Output from a command can be captured and used as a string in the shell environment by enclosing the command in back-quotes (``). In Bourne-compatible shells, we can use $() in place of back-quotes.

```
#!/bin/sh -e

# Using output capture in a command
printf "Today is %s.\n" `date`
printf "Today is %s.\n" $(date)

# Using a variable.  If using the output more than once, this will
# avoid running the command multiple times.
today=`date`
printf "Today is %s\n" $today
```

2.11.5 Self-test

1. Describe two purposes for shell variables.
2. Are any shell variable names reserved? If so, describe two examples.
3. Show how to assign the value "Roger" to the variable "first_name" in both Bourne shell and C shell.
4. Why can there be no spaces around the '=' in a Bourne shell variable assignment?
5. How can you avoid problems when assigning values that contain white space?
6. Do shell variables need to be declared before they are used? Why or why not?
7. Show how to assign the value "xterm" to the *environment* variable TERM in both Bourne shell and C shell.
8. Why do we need to precede variables names with a '$' when referencing them?
9. How can we output a letter immediately after a variable reference (no spaces between them). For example, show a printf statement that prints the contents of the variable `fruit` immediately followed by the letter 's'.

```
fruit=apple
# Show a printf that will produce the output "I have 10 apples", using
# the variable fruit.
```

10. How can variables be used to enhance code quality? From what kinds of errors does this protect you?
11. How can a variable be made read-only in Bourne shell? In C shell?

2.12 Hard and Soft Quotes

Double quotes are known as *soft quotes*, since shell variable references, history events (!), and command output capture ($() or ``) are all expanded when used inside double quotes.

```
shell-prompt: history
  1003  18:11   ps
  1004  18:11   history

shell-prompt: echo "!hi"
echo "history"
history

shell-prompt: echo "Today is `date`"
Today is Tue Jun 12 18:12:33 CDT 2018

shell-prompt: echo "$TERM"
xterm
```

Single quotes are known as *hard quotes*, since every character inside single quotes is taken literally as part of the string, except for history events. Nothing else inside hard quotes is processed by the shell. If you need a literal ! in a string, it must be escaped.

```
shell-prompt: history
  1003  18:11   ps
  1004  18:11   history
shell-prompt: echo '!hi'
echo 'history'
history
shell-prompt: echo '\!hi'
!hi
shell-prompt: echo 'Today is `date`'
Today is `date`
shell-prompt: echo '$TERM'
$TERM
```

What will each of the following print? (If you're not sure, try it!)

```
#!/usr/bin/env bash

name='Joe Sixpack'
printf "Hi, my name is $name.\n"
```

```
#!/usr/bin/env bash

name='Joe Sixpack'
printf 'Hi, my name is $name.\n'
```

```
#!/usr/bin/env bash

first_name='Joe'
```

```
last_name='Sixpack'
name='$first_name $last_name'
printf "Hi, my name is $name.\n"
```

If you need to include a quote character as part of a string, you have two choices:

1. "Escape" it (precede it with a backslash character):

   ```
   printf 'Hi, I\'m Joe Sixpack.\n'
   ```

2. Use the other kind of quotes to enclose the string. A string terminated by double quotes can contain a single quote and vice-versa:

   ```
   printf "Hi, I'm Joe Sixpack.\n"
   printf 'You can use a " in here.\n'
   ```

No special operators are needed to concatenate strings in a shell script. We can simply place multiple strings in any form (variable references, literal text, etc.) next to each other.

```
printf 'Hello ,'$var'.'     # Variable between two hard-quotes strings
printf "Hello, $var."       # Variable between text in a soft-quoted string
```

2.12.1 Self-test

1. What is the difference between soft and hard quotes?
2. Show the output of the following script:

   ```
   #!/bin/sh

   name="Bill Murray"
   printf "$name\n"
   printf '$name\n'
   printf $name\n
   ```

2.13 User Input

In Bourne Shell derivatives, data can be input from the standard input using the **read** command:

```
#!/usr/bin/env bash

printf "Please enter your name: "
read name
printf "Hello, $name!\n"
```

C shell and T-shell use a symbol rather than a command to read input:

```
#!/bin/csh

printf "Please enter your name: "
set name="$<"
printf "Hello, $name!\n"
```

The $< symbol behaves like a variable, which makes it more flexible than the **read** command used by Bourne family shells. It can be used anywhere a regular variable can appear.

```
#!/bin/csh

printf "Enter your name: "
printf "Hi, $<!\n"
```

Note The $< symbol should always be enclosed in soft quotes in case the user enters text containing white space.

Practice Break

Write a shell script that asks the user to enter their first name and last name, stores each in a separate shell variable, and outputs "Hello, first-name last-name".

```
Please enter your first name: Barney
Please enter your last name: Miller
Hello, Barney Miller!
```

2.13.1 Self-test

1. Do the practice break in this section if you haven't already.

2.14 Conditional Execution

Sometimes we need to run a particular command or sequence of commands only if a certain condition is true.

For example, if program B processes the output of program A, we probably won't want to run B at all unless A finished successfully.

2.14.1 Command Exit Status

Conditional execution in Unix shell scripts often utilizes the *exit status* of the most recent command.

All Unix programs return an exit status. It is not possible to write a Unix program that does not return an exit status. Even if the programmer neglects to explicitly return a value, the program will return a default value.

By convention, programs return an exit status of 0 if they determine that they completed their task successfully and a variety of non-zero error codes if they failed. There are some standard error codes defined in the C header file sysexits.h. You can learn about them by running **man sysexits**.

We can check the exit status of the most recent command by examining the shell variable $? in Bourne shell family shells or $status in C shell family shells.

```
bash> ls
myprog.c
bash> echo $?
0
bash> ls -z
ls: illegal option -- z
usage: ls [-ABCFGHILPRSTUWZabcdfghiklmnopqrstuwx1] [-D format] [file ...]
bash> echo $?
1
bash>
```

```
tcsh> ls
myprog.c
tcsh> echo $status
0
tcsh> ls -z
ls: illegal option -- z
usage: ls [-ABCFGHILPRSTUWZabcdfghiklmnopqrstuwx1] [-D format] [file ...]
tcsh> echo $status
1
tcsh>
```

Practice Break
Run several commands correctly and incorrectly and check the $? or $status variable after each one.

2.14.2 If-then-else Statements

All Unix shells have an if-then-else construct implemented as internal commands. The Bourne shell family of shells all use the same basic syntax. The C shell family of shells also use a common syntax, which is somewhat different from the Bourne shell family.

Bourne Shell Family

The general syntax of a Bourne shell family if statement is shown below. Note that there can be an unlimited number of elifs, but we will use only one for this example.

```
#!/bin/sh

if command1
then
    command
    command
    ...
elif command2
then
    command
    command
    ...
else
    command
    command
    ...
fi
```

Note
The 'if' and the 'then' are actually two separate commands, so they must either be on separate lines as shown above, or separated by an operator such as ';', which can be used instead of a newline to separate Unix commands.

```
cd; ls
if command; then
```

The if command executes command1 and checks the exit status when it completes.

If the exit status of command1 is 0 (indicating success), then all the commands before the elif are executed, and everything after the elif is skipped.

If the exit status is non-zero, then nothing above the elif is executed. The elif command then executes command2 and checks its exit status.

If the exit status of command2 is 0, then the commands between the elif and the else are executed and everything after the else is skipped.

If the exit status of command2 is non-zero, then everything above the else is skipped and everything between the else and the fi is executed.

Note In Bourne shell if statements, an exit status of zero effectively means 'true' and non-zero means 'false', which is the opposite of C and similar languages.

In most programming languages, we use some sort of Boolean expression (usually a comparison, also known as a relation), not a command, as the condition for an if statement.

This is generally true in Bourne shell scripts as well, but the capability is provided in an interesting way. We'll illustrate by showing an example and then explaining how is works.

Suppose we have a shell variable and we want to check whether it contains the string "blue". We could use the following if statement to test:

```
#!/bin/sh

printf "Enter the name of a color: "
read color

if [ "$color" = "blue" ]; then
    printf "You entered blue.\n"
elif [ "$color" = "red" ]; then
    printf "You entered red.\n"
else
    printf "You did not enter blue or red.\n"
fi
```

The interesting thing about this code is that the square brackets are *not* Bourne shell syntax. As stated above, the Bourne shell if statement simply executes a command and checks the exit status. This is *always* the case.

The '[' in the condition above is actually an external command! In fact, is simply another name for the **test** command. The files /bin/test and /bin/[are actually the same program file:

```
FreeBSD tocino bacon ~ 401: ls -l /bin/test /bin/[
-r-xr-xr-x  2 root  wheel  8516 Apr  9  2012 /bin/[*
-r-xr-xr-x  2 root  wheel  8516 Apr  9  2012 /bin/test*
```

We could have also written the following:

```
if test "$color" = "blue"; then
```

Hence, "$color", =, "blue", and] are all separate arguments to the [command, and must be separated by white space. If the command is invoked as '[', then the last argument must be ']'. If invoked as 'test', then the ']' is not allowed.

The test command can be used to perform comparisons (relational operations) on variables and constants, as well as a wide variety of tests on files. For comparisons, test takes three arguments: the first and third are string values and the second is a relational operator.

```
# Compare a variable to a string constant
test "$name" = 'Bob'
```

```
# Compare the output of a program directly to a string constant
test `myprog` = 42
```

For file tests, test takes two arguments: The first is a flag indicating which test to perform and the second is the path name of the file or directory.

```
# See if output file exists and is readable to the user
# running test
test -r output.txt
```

The exit status of test is 0 (success) if the test is deemed to be true and a non-zero value if it is false.

```
shell-prompt: test 1 = 1
shell-prompt: echo $?
0
shell-prompt: test 1 = 2
shell-prompt: echo $?
1
```

The relational operators supported by **test** are shown in Table 2.5.

Operator	Relation
=	Lexical equality (string comparison)
-eq	Integer equality
!=	Lexical inequality (string comparison)
-ne	Integer inequality
<	Lexical less-than (10 < 9)
-lt	Integer less-than (9 -lt 10)
-le	Integer less-than or equal
>	Lexical greater-than
-gt	Integer greater-than
-ge	Integer greater-than or equal

Table 2.5: Test Relational Operators

Caution

Note that some operators, such as < and >, have special meaning to the shell, so they must be escaped or quoted.

```
test 10 > 9     # Redirects output to a file called '9'.
    # The only argument sent to the test command is '10'.
    # The test command issues a usage message since it requires
    # more arguments.
test 10 \> 9    # Compares 10 to 9.
test 10 '>' 9   # Compares 10 to 9.
```

Caution It is a common error to use '==' with the test command, but the correct comparison operator is '='.

Common file tests are shown in Table 2.6. To learn about additional file tests, run "man test".

Flag	Test
-e	Exists
-r	Is readable
-w	Is writable
-x	Is executable
-d	Is a directory
-f	Is a regular file
-L	Is a symbolic link
-s	Exists and is not empty
-z	Exists and is empty

Table 2.6:

Caution

Variable references in a [or test command should usually be enclosed in soft quotes. If the value of the variable contains white space, such as "navy blue", and the variable is not enclosed in quotes, then "navy" and "blue" will be considered two separate arguments to the [command, and the [command will fail. When [sees "navy" as the first argument, it expects to see a relational operator as the second argument, but instead finds "blue", which is invalid.
Furthermore, if there is a chance that a variable used in a comparison is empty, then we must attach a common string to the arguments on both sides of the operator. It can be almost any character, but '0' is popular and easy to read.

```
name=""
if [ "$name" = "Bob" ]; then    # Error, expands to: if [ = Bob; then
if [ 0"$name" = 0"Bob" ]; then  # OK, expands to: if [ 0 = 0Bob ]; then
```

Relational operators are provided by the test command, not by the shell. Hence, to find out the details, we would run "man test" or "man [", not "man sh" or "man bash".

See Section 2.14.1 for information about using the test command.

Practice Break
Run the following commands in sequence and run 'echo $?' after every test or [command under bash and 'echo $status' after every test or [command under tcsh.

```
bash
test 1 = 1
test 1=2
test 1 = 2
[ 1 = 1
[ 1 = 1 ]
[ 1 = '1' ]
[1=1]
[ 2 < 10 ]
[ 2 = 3 ]
[ 2 \< 10 ]
[ 2 '<' 10 ]
[ 2 -lt 10 ]
[ $name = 'Bill' ]
[ 0$name = 0'Bill' ]
name='Bob'
[ $name = 'Bill' ]
[ $name = Bill ]
[ $name = Bob ]
exit
tcsh
[ $name = 'Bill' ]
[ 0$name = 0'Bill' ]
set name='Bob'
[ $name = 'Bill' ]
which [
exit
```

C shell Family

Unlike the Bourne shell family of shells, the C shell family implements its own operators, so there is generally no need for the test or [command (although you can use it in C shell scripts if you really want to).

The C shell if statement requires () around the condition, and the condition is always a Boolean expression, just like in C and similar languages. As in C, and unlike Bourne shell, a value of zero is considered false and non-zero is true.

```
#!/bin/csh -ef

printf "Enter the name of a color: "
set color = "$<"

if ( "$color" == "blue" ) then
    printf "You entered blue.\n"
else if ( "$color" == "red" ) then
    printf "You entered red.\n"
else
    printf "You did not enter blue or red.\n"
endif
```

The C shell relational operators are shown in Table 2.7.

Note C shell does not directly support string comparisons except for equality and inequality. To see if a string is lexically less-than or greater than another, use the test or [command with <, <=, >, or >=.

Operator	Relation
<	Integer less-than
>	Integer greater-than
<=	Integer less-then or equal
>=	Integer greater-than or equal
==	String equality
!=	String inequality
=~	String matches glob pattern
!~	String does not match glob pattern

Table 2.7: C Shell Relational Operators

Conditions in a C shell if statement do not have to be relations, however. We can check the exit status of a command in a C shell if statement using the { } operator:

```
#!/bin/csh

if ( { command } ) then
    command
    ...
endif
```

The { } essentially inverts the exit status of the command. If command returns 0, the the value of { command } is 1, which means "true" to the if statement. If command returns a non-zero status, then the value of { command } is zero.

C shell if statements also need soft quotes around strings that contain white space. However, unlike the test command, it can handle empty strings, so we don't need to add an arbitrary prefix like '0' if the string may be empty.

```
if [ 0"$name" = 0"Bob" ]; then
```

```
if ( "$name" == "Bob" ) then
```

Practice Break

Type in the following commands in sequence:

```
tcsh
if ( $first_name == Bob ) then
    printf 'Hi, Bob!\n'
endif
set first_name=Bob
if ( $first_name == Bob ) then
    printf 'Hi, Bob!\n'
endif
exit
```

2.14.3 Conditional Operators

The shell's conditional operators allow us to alter the exit status of a command or utilize the exit status of each in a sequence of commands. They include the Boolean operators AND (&&), OR (||), and NOT (!).

```
shell-prompt: command1 && command2
shell-prompt: command1 || command2
shell-prompt: ! command
```

Operator	Meaning	Exit status
! command	NOT	0 if command failed, 1 if it succeeded
command1 && command2	AND	0 if both commands succeeded
command1 || command2	OR	0 if either command succeeded

Table 2.8: Shell Conditional Operators

Note that in the case of the && operator, command2 will not be executed if command 1 fails (exits with non-zero status), since it could not change the exit status. Once any command in an && sequence fails, the exit status of the whole sequence will be 1, so no more commands will be executed.

Likewise in the case of a || operator, once any command succeeds (exits with zero status), the remaining commands will not be executed.

This fact is often used to conditionally execute a command only if another command is successful:

```
pre-processing && main-processing && post-processing
```

When using the test or [commands, multiple tests can be performed using either the shell's conditional operators or the test command's Boolean operators:

```
if [ 0"$first_name" = 0"Bob" ] && [ 0"$last_name" = 0"Newhart" ]; then
if test 0"$first_name" = 0"Bob" && test 0"$last_name" = 0"Newhart"; then
```

```
if [ 0"$first_name" = 0"Bob" -a 0"$last_name" = 0"Newhart" ]; then
if test 0"$first_name" = 0"Bob" -a 0"$last_name" = 0"Newhart"; then
```

The latter is probably more efficient, since it only executes a single [command, but efficiency in shell scripts is basically a lost cause, so it's best to aim for readability instead. If you want speed, use a compiled language.

Conditional operators can also be used in a C shell if statement. Parenthesis are recommended around each relation for readability.

```
if ( ("$first_name" == "Bob") && ("$last_name" == "Newhart") ) then
```

Practice Break
Run the following commands in sequence and run 'echo $?' after every command under bash and 'echo $status' after every command under tcsh.

```
bash
ls -z
ls -z && echo Done
ls -a && echo Done
ls -z || echo Done
ls -a || echo Done
first_name=Bob
last_name=Newhart
if [ 0"$first_name" = 0"Bob" ] && [ 0"$last_name" = 0"Newhart" ]
then
    printf 'Hi, Bob!\n'
fi
if [ 0"$first_name" = 0"Bob" -a 0"$last_name" = 0"Newhart" ]
then
    printf 'Hi, Bob!\n'
fi
exit
tcsh
ls -z
ls -z && echo Done
ls -a && echo Done
ls -z || echo Done
ls -a || echo Done
if ( $first_name == Bob && $last_name == Newhart ) then
    printf 'Hi, Bob!\n'
endif
set first_name=Bob
set last_name=Nelson
if ( $first_name == Bob && $last_name == Newhart ) then
    printf 'Hi, Bob!\n'
endif
set last_name=Newhart
if ( $first_name == Bob && $last_name == Newhart ) then
    printf 'Hi, Bob!\n'
endif
exit
```

2.14.4 Case and Switch Statements

If you need to compare a single variable to many different values, you could use a long string of elifs or else ifs:

```
#!/bin/sh

printf "Enter a color name: "
read color

if [ "$color" = "red" ] || \
   [ "$color" = "orange" ]; then
    printf "Long wavelength\n"
elif [ "$color" = "yello" ] || \
     [ "$color" = "green" ] || \
     [ "$color" = "blue" ]; then
    printf "Medium wavelength\n"
elif [ "$color" = "indigo" ] || \
     [ "$color" = "violet" ]; then
```

```
    printf "Short wavelength\n"
else
    printf "Invalid color name: $color\n"
fi
```

Like most languages, however, Unix shells offer a cleaner solution.

Bourne shell has the case statement:

```
#!/bin/sh

printf "Enter a color name: "
read color

case $color in
    red|orange)
        printf "Long wavelength\n"
        ;;
    yellow|green|blue)
        printf "Medium wavelength\n"
        ;;
    indigo|violet)
        printf "Short wavelength\n"
        ;;
    *)
        printf "Invalid color name: $color\n"
        ;;
esac
```

C shell has a switch statement that looks almost exactly like the switch statement in C, C++, and Java:

```
#!/bin/csh -ef

printf "Enter a color name: "
set color = "$<"

switch($color)
case    red:
case    orange:
    printf "Long wavelength\n"
    breaksw
case    yellow:
case    green:
case    blue:
    printf "Medium wavelength\n"
    breaksw
case    indigo:
case    violet:
    printf "Short wavelength\n"
    breaksw
default:
    printf "Invalid color name: $color\n"
endsw
```

Note The ;; and breaksw statements cause a jump to the first statement after the entire case or switch. The ;; is required after every value in the case statement. The breaksw is optional in the switch statement. If omitted, the script will simply continue on and execute the statements for the next case value.

2.14.5 Self-test

1. What is an "exit status"? What conventions to Unix programs follow regarding the exit status?
2. How can we find out the exit status of the previous command in Bourne shell? In C shell?
3. Write a Bourne shell script that uses an if statement to run 'ls -l > output.txt' and view the output using 'more' only of the ls command succeeded.
4. Repeat the previous problem using C shell.
5. Write a Bourne shell script that inputs a first name and outputs a different message depending on whether the name is 'Bob'.

```
shell-prompt: ./script
What is your name? Bob
Hey, Bob!
shell-prompt: ./script
What is your name? Bill
Hey, you're not Bob!
```

6. Repeat the previous problem using C shell.
7. Write a Bourne and/or C shell script that inputs a person's age and indicates whether they get free peanuts. Peanuts are provided to senior citizens.

```
shell-prompt: ./script
How old are you? 42
Sorry, no peanuts for you.
shell-prompt: ./script
How old are you? 72
Have some free peanuts, wise sir!
```

8. Why is it necessary to separate the tokens between [and] with white space? What will happen if we don't?
9. What will happen if a value being compared using test or [contains white space? How to we remedy this?
10. What will happen if a value being compared using test or [is an empty string? How to we remedy this?
11. Why do the < and > operators need to be escaped (\<, \>) or quoted when used with the test command?
12. What is the == operator used for with the test command?
13. How do we check the exit status of a command in a C shell if statement?
14. Write a Bourne and/or C shell script that inputs a person's age and indicates whether they get free peanuts. Peanuts are provided to senior citizens and children between the ages of 3 and 12.

```
shell-prompt: ./script
How old are you? 42
Sorry, no peanuts for you.
shell-prompt: ./script
How old are you? 72
Have some free peanuts, wise sir!
```

15. Write a Bourne shell script that uses conditional operators to run 'ls -l > output.txt' and view the output using 'more' only of the ls command succeeded.
16. Write a shell script that checks the output of **uname** using a case or switch statement and reports whether the operating system is supported. Assume supported operating systems include Cygwin, Darwin, FreeBSD, and Linux.

```
shell-prompt: ./script
FreeBSD is supported.
```

```
shell-prompt: ./script
AIX is not supported.
```

2.15 Loops

We often need to run the same command or commands on a group of files or other data.

2.15.1 For and Foreach

Unix shells offer a type of loop that takes an enumerated list of string values, rather than counting through a sequence of numbers. This makes it more flexible for working with sets of files or or arbitrary sets of values.

This type of loop is well suited for use with globbing (file name patterns using wild cards, as discussed in Section 1.8.5):

```
#!/usr/bin/env bash

# Process input-1.txt, input-2.txt, etc.
for file in input-*.txt
do
    ./myprog $file
done
```

```
#!/bin/csh -ef

# Process input-1.txt, input-2.txt, etc.
foreach file (input-*.txt)
    ./myprog $file
end
```

These loops are not limited to using file names, however. We can use them to iterate through any list of string values:

```
#!/bin/sh

for fish in flounder gobie hammerhead manta moray sculpin
do
    printf "%s\n" $fish
done
```

```
#!/usr/bin/env bash

for c in 1 2 3 4 5 6 7 8 9 10
do
    printf "%d\n" $c
done
```

Practice Break
Type in and run the fish example above.

Note Note again that the Unix commands, including the shell, don't generally care whether their input comes from a file or a device such as the keyboard. Try running the fish example by typing it directly at the shell prompt as well as by writing a script file. When running it directly, be sure to use the correct shell syntax for the interactive shell you are running.

Example 2.3 Multiple File Downloads

Often we need to download many large files from another site. This process would be tedious to do manually: Start a download, wait for it to finish, start another... There may be special tools provided, but often they are unreliable or difficult to install. In many cases, we may be able to automate the download using a simple script and a file transfer tool such as **curl**, **fetch**, **rsync** or **wget**.

The model scripts below demonstrate how to download a set of files using curl. The local file names will be the same as those on the remote site and if the transfer is interrupted for any reason, we can simply run the script again to resume the download where it left off.

Depending on the tools available on your local machine and the remote server, you may need to substitute another file transfer program for curl.

```
#!/bin/sh -e

# Download genome data from the ACME genome project
site=http://server.with.my.files/directory/with/my/files
for file in frog1 frog2 frog3 toad1 toad2 toad3; do
    printf "Fetching $site/$file.fasta.gz...\n"

    # Use filename from remote site and try to resume interrupted
    # transfers if a partial download already exists
    curl --continue-at - --remote-name $site/$file.fasta.gz
fi
```

```
#!/bin/csh -ef

# Download genome data from the ACME genome project
set site=http://server.with.my.files/directory/with/my/files
foreach file (frog1 frog2 frog3 toad1 toad2 toad3)
    printf "Fetching $site/$file.fasta.gz...\n"

    # Use filename from remote site and try to resume interrupted
    # transfers if a partial download already exists
    curl --continue-at - --remote-name $site/$file.fasta.gz
end
```

2.15.2 While Loops

A for or foreach loop is not convenient for iterating through a long number sequence.

The **while** loop is a more general loop that iterates as long as some condition is true. It uses the same types of expressions as an **if** statement.

The while loop is often used to iterate through long integer sequences:

```
#!/usr/bin/env bash

c=1
while [ $c -le 100 ]
do
    printf "%d\n" $c
    c=$(($c + 1))        # (( )) encloses an integer expression
done
```

Note again that the [above is an external command, as discussed in Section 2.14.1.

```
#!/bin/csh -ef

set c = 1
```

```
while ( $c <= 100 )
    printf "%d\n" $c
    @ c = $c + 1        # @ is like set, but indicates an integer expression
end
```

Practice Break
Type in and run the script above.

While loops can also be used to iterate until an input condition is met:

```
#!/bin/sh

continue=''
while [ 0"$continue" != 0'y' ] && [ 0"$continue" != 0'n' ]; do
    printf "Would you like to continue? (y/n) "
    read continue
done
```

```
#!/bin/csh -ef

set continue=''
while ( ("$continue" != 'y') && ("$continue" != 'n') )
    printf "Continue? (y/n) "
    set continue="$<"
end
```

Practice Break
Type in and run the script above.

We may even want a loop to iterate forever. This is often useful when using a computer to collect data at regular intervals:

```
#!/bin/sh

# 'true' is an external command that always returns an exit status of 0
while true; do
    sample-data     # Read instrument
    sleep 10        # Pause for 10 seconds without using any CPU time
done
```

```
#!/bin/csh -ef

while ( 1 )
    sample-data     # Read instrument
    sleep 10        # Pause for 10 seconds without using any CPU time
end
```

2.15.3 Self-test

1. Write a shell script that prints the square of every number from 1 to 100.

2. Write a shell script that sorts all files with names ending in ".txt" one at a time, removes duplicate entries, and saves the output to `filename.txt.sorted`. The script then merges all the sorted text into a single file called `combined.txt.sorted`. The **sort** can also merge presorted files when used with the `-m` flag.

   The standard Unix **sort** can be used to sort an individual file. The **uniq** command will remove duplicate lines that are adjacent to each other. (Hence, the data should be sorted already.)

3. Do the examples for shell loops above give you any ideas about using multiple computers to speed up processing?

2.16 Generalizing Your Code

All programs and scripts require input in order to be useful.

Inputs commonly include things like scalar parameters to use in equations and the names of files containing more extensive data such as a matrix or a database.

2.16.1 Hard-coding: Failure to Generalize

All too often, inexperienced programmers provide what should be input to a program by hard-coding values and file names into their programs and scripts:

```
#!/bin/csh

# Hard-coded values 1000 and output.txt
calcpi 1000 > output.txt
```

Many programmers will then make another copy of the program or script with different constants or file names in order to do a different run. The problem with this approach should be obvious. It creates a mess of many nearly identical programs or scripts, all of which have to be maintained together. If a bug is found in one of them, then all of them have to be checked and corrected for the same error.

2.16.2 Generalizing with User Input

A better approach is to take these values as input:

```
#!/bin/csh

printf "How many iterations? "
set iterations = "$<"
printf "Output file? "
set output_file = "$<"

calcpi $iterations > $output_file
```

If you don't want to type in the values every time you run the script, you can put them in a separate input file, such as "input-1000.txt" and use redirection:

```
shell-prompt: cat input-1000.txt
1000
output-1000.txt
shell-prompt: calcpi-script < input-1000.txt
```

This way, if you have 50 different inputs to try, you have 50 input files and only one script to maintain instead of 50 scripts.

2.16.3 Generalizing with Command-line Arguments

Another approach is to design the script so that it can take command-line arguments, like most Unix commands. Using command-line arguments is quite simple in most scripting and programming languages.

In all Unix shell scripts, the first argument is denoted by the special variable $1, the second by $2, and so on.

$0 refers to the name of the command as it was invoked.

Bourne Shell Family

In Bourne Shell family shells, we can find out how many command-line arguments were given by examining the special shell variable "$#". This is most often used to verify that the script was invoked with the correct number of arguments.

```
#!/bin/sh

# If invoked incorrectly, tell the user the correct way
if [ $# != 2 ]; then
    printf "Usage: $0 iterations output-file\n"
    exit 1
fi

# Assign to named variables for readability
iterations=$1
output_file="$2"    # File name may contain white space!

calcpi $iterations > "$output_file"
```

```
shell-prompt: calcpi-script
Usage: calcpi-script iterations output-file
shell-prompt: calcpi-script 1000 output-1000.txt
shell-prompt: cat output-1000.txt
3.141723494
```

C shell Family

In C shell family shells, we can find out how many command-line arguments were given by examining the special shell variable "$#argv".

```
#!/bin/csh

# If invoked incorrectly, tell the user the correct way
if ( $#argv != 2 ) then
    printf "Usage: $0 iterations output-file\n"
    exit 1
endif

# Assign to named variables for readability
set iterations=$1
set output_file="$2"    # File name may contain white space!

calcpi $iterations > "$output_file"
```

```
shell-prompt: calcpi-script
Usage: calcpi-script iterations output-file
shell-prompt: calcpi-script 1000 output-1000.txt
shell-prompt: cat output-1000.txt
3.141723494
```

2.16.4 Self-test

1. Modify the following shell script so that it takes the file name of the dictionary and the sample word as user input instead of hard-coding it. You may use any shell you choose.

```
#!/bin/sh

if fgrep 'abacus' /usr/share/dict/words; then
    printf 'abacus is a real word.\n'
else
    printf 'abacus is not a real word.\n'
fi
```

2. Repeat the above exercise, but use command-line arguments instead of user input.

2.17 Scripting an Analysis Pipeline

2.17.1 What's an Analysis Pipeline?

An analysis pipeline is simply a sequence of processing steps.

Since the steps are basically the same for a given type of analysis, we can automate the pipeline using a scripting language for the reasons we discussed at the beginning of this chapter: To save time and avoid mistakes.

A large percentage of scientific research analyses require multiple steps, so pipelines are very common in practice.

2.17.2 Where do Pipelines Come From?

It has been said that for every PhD thesis, there is a pipeline.

There are many preexisting pipelines available for a wide variety of tasks.

Many such pipelines were developed by researchers for a specific project, and then generalized in order to be useful for other projects or other researchers.

Unfortunately, most such pipelines are not well designed or rigorously tested, so they don't work well for analyses that differ significantly from the one for which they were originally designed.

Another problem is that most of them are not well maintained over the long term. Developers set out with good intentions to help other researchers, but once their project is done and they move onto new things, they find that they don't have time to maintain old pipelines anymore. Also, new tools are constantly evolving and old pipelines therefore quickly become obsolete unless they are aggressively updated.

Finally, many pipelines are integrated into a specific system with a graphical user interface (GUI) or web interface, and therefore cannot be used on a generic computer or HPC cluster (unless the entire system is installed and configured, which is often difficult or impossible).

For these reasons, every researcher should know how to develop their own pipelines. Relying on the charity of your competitors for publishing space and grant money will not lead to long-term success.

This is especially true for long-term studies. If you become dependent on a preexisting pipeline early on, and it is not maintained by its developers for the duration of *your* study, then the completion of your study will prove very difficult.

2.17.3 Implementing Your Own Pipeline

A pipeline can be implemented in any programming language.

Since most pipelines involve simply running a series of programs with the appropriate command-line arguments, a Unix shell script is a very suitable choice in most cases.

In some cases, it may be possible to use Unix shell pipes to perform multiple steps at the same time. This will depend on a number of things:

- Do the processing programs use standard input and standard output? If not, then redirecting to and from them with pipes will not be possible.
- What are the resource requirements of each step? Do you have enough memory to run multiple steps at the same time?
- Do you need to save the intermediate files generated by some of the steps? If so, then either don't use a Unix shell pipe, or use the **tee** command to dump output to a file and pipe it to the next command at the same time.

```
shell-prompt: step1 < input1 | tee output1 | step2 > output2
```

2.17.4 An Example Genomics Pipeline

Below is a simple shell script implementation of the AmrPlusPlus pipeline, which, according to their website, is used to "characterize the content and relative abundance of sequences of interest from the DNA of a given sample or set of samples".

People can use this pipeline by uploading their data to the developer's website, or by installing the pipeline to their own Docker container or Galaxy server.

In reality, the analysis is performed by the following command-line tools, which are developed by other parties and freely available:

- Trimmomatic
- BWA
- Samtools
- SNPFinder
- ResistomeAnalyzer
- RarefactionAnalyzer

The role of AmrPlusPlus is to coordinate the operation of these tools. AmrPlusPlus is itself a script.

If you don't want to be dependent on their web service, a Galaxy server, or their Docker containers, or if you would like greater control over and understanding of the analysis pipeline, or if you want to use the newer versios of tools such as samtools, you can easily write your own script to run the above commands.

Also, when developing our own pipeline, we can substitute other tools that perform the same function, such as Cutadapt in place of Trimmomatic, or Bowtie (1 or 2) in place of BWA for alignment.

All of these tools are designed for "short-read" DNA sequences (on the order of 100 base pair per fragment). When we take control of the process rather than rely on someone else's pipeline, we open the possibility of developing an analogous pipeline using a different set of tools for "long-read" sequences (on the order of 1000 base pair per fragment).

For our purposes, we install the above commands via FreeBSD ports and/or pkgsrc (on CentOS and Mac OS X). To facilitate this, I created a metaport that automatically installs all the tools needed by the pipeline (trimmimatic, bwa, ...) as dependencies. The following will install everything in a few minutes:

```
shell-prompt: cd /usr/ports/wip/amr-cli
shell-prompt: make install
```

Then we just write a Unix shell script to implement the pipeline for our data.

Note that this is a real pipeline used for research at the UWM School of Freshwater Science.

It is not important whether you understand genomics analysis for this example. Simply look at how the script uses loops and other scripting constructs to see how the material you just learned can be used in actual research. I.e., don't worry about what **cutadapt** and **bwa** are doing with the data. Just see how they are run within the pipeline script, using redirection, command line arguments, etc. Also read the comments within the script for a deeper understanding of what the conditionals and loops are doing.

```
#!/bin/sh -e

# Get gene fraction threshold from user
printf "Resistome threshold? "
read threshold

################################################################################
# 1. Enumerate input files

raw_files="SRR*.fastq"

################################################################################
# 2.  Quality control: Remove adapter sequences from raw data

for file in $raw_files; do
    output_file=trimmed-$file
    # If the output file already exists, assume cutadapt was already run
    # successfully.  Remove trimmed-* before running this script to force
    # cutadapt to run again.
    if [ ! -e $output_file ]; then
        cutadapt $file > $output_file
    else
        printf "$raw already processed by cutadapt.\n"
    fi
done

################################################################################
# 3. If sequences are from a host organism, remove host dna

# Index resistance gene database
if [ ! -e megares_database_v1.01.fasta.ann ]; then
    bwa index megares_database_v1.01.fasta
fi

################################################################################
# 4. Align to target database with bwa mem.

for file in $raw_files; do
    # Output is an aligned sam file.  Replace trimmed- prefix with aligned-
    # and replace .fastq suffix with .sam
    output_file=aligned-${file%.fastq}.sam
    if [ ! -e $output_file ]; then
        printf "\nRunning bwa-mem on $file...\n"
        bwa mem megares_database_v1.01.fasta trimmed-$file > $output_file
    else
        printf "$file already processed by bwa mem\n"
    fi
done

################################################################################
# 5. Resistome analysis.

aligned_files=aligned-*.sam
for file in $aligned_files; do
    if [ ! -e ${file%.sam}group.tsv ]; then
        printf "\nRunning resistome analysis on $file...\n"
        resistome -ref_fp megares_database_v1.01.fasta -sam_fp $file \
            -annot_fp megares_annotations_v1.01.csv \
            -gene_fp ${file%.sam}gene.tsv \
            -group_fp ${file%.sam}group.tsv \
            -class_fp ${file%.sam}class.tsv \
```

```
            -mech_fp ${file%.sam}mech.tsv \
            -t $threshold
    else
        printf "$file already processed by resistome.\n"
    fi
done

################################################################################
# 6. Rarefaction analysis?
```

I generally write a companion to every analysis script to remove output files and allow a fresh start for the next attempt:

```
#!/bin/sh -e

rm -f trimmed-* aligned-* aligned-*.tsv megares*.fasta.*
```

2.18 Functions and Calling other Scripts

Most scripts tend to be short, but even a program of 100 lines long can benefit from being broken down and organized into modules.

The Bourne family shells support simple functions for this purpose.

C shell family shells do not support separate functions within a script, but this does not mean that they cannot be modularized. A C shell script can, of course, run other scripts and these separate scripts can serve the purpose of subprograms.

Some would argue that separate scripts are more modular than functions, since a separate script is inherently available to any script that could use it, whereas a function is confined to the script that contains it.

Another advantage of using separate scripts is that they run as a separate process, so they have their own set of shell and environment variables. Hence, they do not have side-effects on the calling script. Bourne shell functions, on the other hand, can modify "global" variables and impact the subsequent behavior of other functions or the main program.

There are some functions that are unlikely to be useful in other scripts, however, and Bourne shell functions are convenient in these cases. Also, it is generally easy to convert a Bourne shell function into a separate script, so there isn't generally much to lose by using a function initially.

2.18.1 Bourne Shell Functions

A Bourne shell function is defined by simply writing a name followed by parenthesis, and a body between curly braces on the lines below:

```
name()
{
    commands
}
```

We call a function the same way we run any other command.

```
#!/bin/sh

line()
{
    printf '------------------------------------------------------------\n'
}

line
```

I we pass arguments to a function, then the variables $1, $2, etc. in the function will be set to the arguments passed to the function. Otherwise, $1, $2, etc. will be the arguments to the main script.

```
#!/bin/sh

print_square()
{
    printf $(($1 * $1))
}

c=1
while [ $c -le 10 ]; do
    printf "%d squared = %d\n" $c `print_square c`
    c=$((c + 1))
done
```

The return statement can be used to return a value to the caller. The return value is received by the caller in $?, just like the exit status of any other command. This is most often used to indicate success or failure of the function.

```
#!/bin/sh

myfunc()
{
    if ! command1; then
        return 1

    if ! command2; then
        return 1

    return 0
}

if ! myfunc; then
    exit 1
fi
```

We can define local variables in a function if we do not want the function to modify a variable outside itself.

```
#!/bin/sh

pause()
{
    local response

    printf "Press return to continue..."
    read response
}

pause
```

2.18.2 C Shell Separate Scripts

Since C shell does not support internal functions, we implement subprograms as separate scripts.

Each script is executed by a separate shell process, so all variables are essentially local to that script.

We can, of course, use the **source** to run another script using the parent shell process as described in Section 2.7. In this case, it will affect the shell and environment variables of the calling script. This is usually what we intend and the very reason for using the **source** command.

When using separate scripts as subprograms, it is especially helpful to place the scripts in a directory that is in your PATH. Most users use a directory such as ~/bin or ~/scripts for this purpose.

2.18.3 Self-test

2.19 Alias

An alternative to functions and separate scripts for very simple things is the **alias** command.

This command creates an alias, or alternate name for another command.

Aliases are supported by both Bourne and C shell families, albeit with a slightly different syntax.

They are most often used to create simple shortcuts for common commands.

In Bourne shell and derivatives, the new alias is followed by an '='. Any command containing white space must be enclosed in quotes, or the white space must be escaped with a \.

```
#!/bin/sh

alias dir='ls -als'

dir
```

C shell family shells use white space instead of an '=' and do not require quotes around commands containing white space.

```
#!/bin/csh

alias dir ls -als

dir
```

An alias can contain multiple commands, but in this case it must be enclosed in quotes, even in C shell.

```
#!/bin/csh

# This will not work:
# alias pause printf "Press return to continue..."; $<
#
# It is the same as:
#
# alias pause printf "Press return to continue..."
# $<

# This works
alias pause 'printf "Press return to continue..."; $<'

pause
```

2.20 Shell Flags and Variables

Unix shells have many command line flags to control their behavior.

One of the most popular shell flags is -e. The -e flag in both Bourne Shell and C shell cause the shell to exit if any command fails. This is almost always a good idea, to avoid wasting time and so that the last output of a script shows any error messages from the failed command.

Flags can be used in the shebang line if the path of the shell is fixed. When using #!/usr/bin/env, we must set the option using a separate command, because the shebang line on some systems treats everything after '#!/usr/bin/env' as a single argument to the env command.

```
#!/bin/sh -e

# The shebang line above is OK
```

```
#!/bin/csh -e

# The shebang line above is OK
```

```
#!/usr/bin/env bash -e

# The shebang line above is invalid on some systems and may cause
# an error such as "bash -e: command not found"
```

We can get around this in Bourne family shells using the set command, which can be used to turn on or off command-line flags within the script. For example, "set -e" in a script causes the shell running the script to terminate if any subsequent commands fail. A "set +e" turns off this behavior.

```
#!/usr/bin/env bash

# Enable exit-on-error
set -e
```

Unfortunately, C shell family shells do not have anything comparable to the Bourne shell set command. Recall that C shell has a set command, but it is use to set shell variables, not command-line flags.

Many features controlled by command-line flags can also be set within a C shell script using special shell variables, but -e is not one of them.

The -x flag is another flag common to both Bourne Shell and C shell. If causes the shell to **echo** commands to the standard output before executing them, which is often useful in debugging a script that it failing at an unknown location.

```
#!/bin/sh -x
```

```
#!/bin/csh -x
```

```
#!/bin/sh

set -x      # Enable command echo
command
command
set +x      # Disable command echo
```

```
#!/bin/csh

set echo    # Enable command echo
command
command
unset echo  # Disable command echo
```

As stated in Section 2.6, Bourne shell family scripts do not source any start up scripts by default. Bourne shells only source files like .shrc, .bashrc, etc. if the shell is interactive, i.e. the standard input is a terminal.

C shell and T shell scripts, on the other hand, will source .cshrc or .tcshrc by default. This behavior can be disabled using the `-f` flag. Disabling this is usually a good idea, since the script may behave differently for different people, depending on what's in their .cshrc.

```
#!/bin/csh -ef
```

There are many other command-line flags and corresponding C shell variables. For more information, run "man sh", "man csh", etc.

2.21 Arrays

Bourne shell does not support arrays, but some commands can process strings containing multiple words separated by white space.

```
#!/bin/sh

names="Barney Bert Billy Bob Brad Brett Brody"
for name in $names; do
    ...
done
```

C shell supports basic arrays. One advantage of this is that we can create lists of strings where some of the elements contain white space.

An array constant is indicated by a list enclosed in parenthesis.

Each array element is identified by an integer subscript.

We can also use a range of subscripts, separated by '-'.

```
#!/bin/csh -ef

set names=("Bob Newhart" "Bob Marley" "Bobcat Goldthwait")
set c=1
while ( $c <= $#names )
    printf "$names[$c]\n"
    @ c++
end

printf "$names[2-3]\n"
```

Caution The foreach command is not designed to work with arrays. It is designed to break a string into white space-separated tokens. Hence, given an array, foreach will view is as one large string and then break it wherever there is white space, which could break individual array elements into multiple pieces.

The $argv variable containing command-line arguments is an array. Hence, the $#argv variable is not special to $argv, but just another example of referencing the number of elements in an array.

2.22 Good and Bad Practices

A very common and very bad practice in shell scripting is checking the wrong information to make decisions. One of the most common ways this bad approach is used involves making assumptions based on the operating system in use.

Take the following code segment, for example:

```
if [ `uname` == 'Linux' ]; then
    compiler='gcc'
    endian='little'
fi
```

Both of the assumptions made about Linux in the code above were taken from real examples!

Setting the compiler to 'gcc' because we're running on Linux is wrong, because Linux can run other compilers such as clang or icc. Compiler selection should be based on the user's wishes or the needs of the program being compiled, not the operating system alone.

Assuming the machine is little-endian is wrong because Linux runs on a variety of CPU types, some of which are big-endian. The user who wrote this code probably assumed that if the computer is running Linux, it must be a PC with an x86 processor, which is not a valid assumption.

There are simple ways to find out the actual endianness of a system, so why would we instead try to infer it from an unrelated fact?? We should instead use something like the open source **endian** program, which runs on any Unix compatible system.

```
if [ `endian` == 'little' ]; then

fi
```

2.23 Here Documents

We often want to output multiple lines of text from a script, for instance to provide detailed instructions to the user. For instance, the output below is a real example from a script that generates random passphrases.

```
================================================================================
If no one can see your computer screen right now, you may use one of the
suggested passphrases about to be displayed.  Otherwise, make up one of
your own consisting of three words separated by random characters and
modified with a random capital letters or other characters inserted.
================================================================================
```

We could output this text using six printf statements. This would be messy, though, and would require quotes around each line of text.

We could also store it in a separate file and display it with the **cat** command:

```
#!/bin/sh -e

cat instructions.txt
```

This would mean keeping track of multiple files, however.

A "here document", or "heredoc", is another form of redirection that is typically only used in scripts. It essentially redirects the standard input to a portion of the script itself. The general for is as follows:

```
command << end-of-document-marker

end-of-document-marker
```

The end-of-document-marker can be any arbitrary text that you choose. This allows the text to contain literally anything. You simply have to choose a marker that it not in the text you want to display. Common markers are EOM (end of message) or EOF (end of file).

Heredocs can be used with any Unix command that reads from standard input, but are most often used with the **cat** command:

```
#!/bin/sh -e

cat << EOM
================================================================================
If no one can see your computer screen right now, you may use one of the
suggested passphrases about to be displayed.  Otherwise, make up one of
your own consisting of three words separated by random characters and
modified with a random capital letters or other characters inserted.
================================================================================
EOM
```

Heredocs can also be used to create files from a template that uses shell or environment variables. Any variable references that appear within the text of a heredoc will be expanded. The output of any command reading from a heredoc can, of course, be redirected to a file or other device.

```
#!/bin/csh -ef

# Generate a series of test input files with difference ending values
foreach end_value (10 100 1000 10000)
    foreach tolerance (0.0001 0.0005 0.001)
        cat << EOM > test-input-$end_value-$tolerance.txt
start_value=1
end_value=$end_value
tolerance=$tolerance
EOM
    end
end
```

2.24 Common Unix Tools Used in Scripts

It is often said that most Unix users don't need to write programs. The standard Unix commands contain all the functionality that a typical user needs, so they need only learn how to use the commands and write simple scripts to utilize them.

The sections below introduce some of the popular tools with the sole intention of raising awareness. The details of these tools would fill a separate book by themselves, so we will focus on simple, common examples here.

2.24.1 Grep

The **grep** command, short for General Regular exPressions, is a powerful tool for searching the content of text files.

Regular expressions are a standardized syntax for specifying patterns of text. They are similar to the globbing patterns discussed in Section 1.8.5, but the details are quite different. Also, while globbing patterns are meant to match file names, regular expressions are meant to match strings in any context.

Some of the more common regular expression features are shown in Table 2.9.

Token	Matches
.	Any character
[list]	Any single character in list
[first-last]	Any single character between first and last, in the order they appear in the character set in use. This may be affected by locale settings.
*	Zero or more of the preceding token
+	One or more of the preceding token

Table 2.9:

Note To match any special character, such as '.', or '[', precede it with a '\'.

On BSD systems, a POSIX regular expression reference is available via **man re_format**.

On Linux systems, a similar document is available via **man 7 regex**.

Regular expression pattern matching can be used in any language. At the shell level, patterns are typically matched using the **grep** command.

In short, **grep** searches a text file for patterns specified as arguments and prints matching lines.

```
grep pattern file-spec
```

Note Patterns passed to **grep** should usually be hard-quoted to prevent the shell from interpreting them as globbing patterns or other shell features.

```
# Show lines in Bourne shell scripts containing the string "printf"
grep printf *.sh

# Show lines in C programs containing strings that qualify as variable names
grep '[A-Za-z_][A-Za-z_0-9]*' *.c

# Show lines in C programs containing decimal integers
grep '[0-9]+' *.c

# Show lines in C programs containing real numbers
grep '[0-9]*\.[0-9]+' *.c
```

By default, the **grep** command follows an older standard for traditional regular expressions, in order to maintain backward compatibility in older scripts.

To enable the newer extended regular expressions, use **grep -E** or **egrep**.

To disable the use of regular expressions and treat each pattern as a fixed string, use **grep -F** or **fgrep**. This is sometimes useful for better performance or to eliminate the need for '\' before special characters.

2.24.2 Stream Editors

Stream editors are a class of programs that take input from one stream, often standard input, modify it in some way, and send the output to another stream, often standard output.

The **sed** (Stream EDitor) command is among the most commonly used stream editing programs. The **sed** has a variety of capabilities for performing almost any kind of changes you can imagine. Most often, though, it is used to replace text matching a regular expression with something else. Our introduction here will focus on this feature and we will leave the rest for tutorials dedicated to **sed**.

The basic syntax of a **sed** command for replacing text is as follows:

```
sed -e 's|pattern|replacement|g'
```

The `-e` flag specifies the use of traditional regular expressions. To use the more modern extended regular expressions, use `-E` as with **grep**.

The 's' is the 'substitute' command. Other commands, not discussed here, include 'd' (delete) and 'i' (insert).

The '|' is a separator. You can use any character as the separator as long as all three separators are the same. This allows any character to appear in the pattern or replacement text. Just use a separator that is not in either. The most popular separators are '|' and '/', since they usually stand out next to typical patterns.

The pattern is a regular expression, just as we would use with **grep**. Again, special characters that we want to match literally must be escaped (preceded by a '\').

The replacement text is not a regular expression, but may contain some special characters specific to **sed**. The most common is '&', which represents the current string matching the pattern. This feature makes it easy to add text to strings matching a pattern, even if they are not the same.

The 'g' means perform a global replacement. If omitted, only the first match on each line is replaced.

```
# Get snooty
sed -e 's|Bob|Robert|g' file.txt > modified-file.txt

# Convert integer constants to long constants in a C program
sed -e 's|[0-9]+|&L|g' prog.c > prog-long.c
```

The **tr** (translate) command is a simpler stream editing tool. It is typically used to replace or delete individual characters from a stream.

```
# Capitalize all occurrences of 'a', 'b', and 'c'
tr 'abc' 'ABC' file.txt > file-caps.txt

# Delete all digits from a file
tr -d '0123456789' file.txt > file-qless.txt
```

2.24.3 Tabular Data Tools

Unix systems provide standard tools for working with tabular data (text data organized in columns).

The **cut** command is a simple tool for removing a portion from each line of a text stream. The user can specify byte, character, or field positions to be removed.

```
# Remove the 3rd and 4th characters from every line
cut -c 3-4 file.txt > chopped-file.txt

# Remove the first column of numbers separated by white space
cut -w -f 1 results.txt > results-without-col1.txt
```

The **awk** command is an extremely sophisticated tool for manipulating tabular data. It is essentially a non-interactive spreadsheet, capable of doing modifications and computations of just about any kind.

Awk includes a scripting language that looks very much like C, with many extensions for easily processing textual data.

Entire books are available on **awk**, so we will focus on just a few basic examples.

Awk is generally invoked in one of two ways. For very simple awk operations (typically 1-line scripts), we can provide the awk script itself as a command-line argument, usually hard-quoted:

```
awk [-F field-separator] 'script' file-spec
```

For more complex, multi-line scripts, it may prove easier to place the awk script in a separate file and refer to it in the command:

```
awk [-F field-separator] -f script.awk file-spec
```

Input is separated into fields by white space by default, but we can specify any field-separator we like using the `-F`. The field separator can also be changed within the awk script by assigning the special variable FS.

Statements within the **awk** script consist of a pattern and an action.

Patterns may be relational expressions comparing a given field (column) to a pattern. In this case, the action will be invoked only on lines matching the pattern.

If pattern is omitted, the action will be performed on every line of input.

The special patterns BEGIN and END are used to perform actions before the first line is processed and after the last line is processed.

The action is essentially a C-like function. If omitted, the default action is to print the entire line matching pattern. (Hence, awk can behave much like grep.)

Example 1: A simple awk command

```
# Print password entries for users with uid >= 1000
shell-prompt: awk -F : '$3 >= 1000 { print $0 }' /etc/passwd
nobody:*:65534:65534:Unprivileged user:/nonexistent:/usr/sbin/nologin
joe:*:4000:4000:Joe User:/home/joe:/bin/tcsh
```

Example 2: A separate awk script

```
# Initialize variables
BEGIN {
    sum1 = sum2 = 0.0;
}

# Add column data to sum for each line
{
    print $1, $2
    sum1 += $1;
    sum2 += $2;
}

# Output sums after all lines are processed
END {
    printf("Sum of column 1 = %f\n", sum1);
    printf("Sum of column 2 = %f\n", sum2);
}
```

```
shell-prompt: cat twocol.txt
4.3     -2.1
5.5     9.0
-7.3    4.6

shell-prompt: awk -f ./sum.awk twocol.txt
4.3 -2.1
5.5 9.0
-7.3 4.6
Sum of column 1 = 2.500000
Sum of column 2 = 11.500000
```

2.24.4 Sort/Uniq

The **sort** command is a highly efficient, general-purpose stream sorting tool. It sorts the input stream line-by-line, optionally prioritizing the sort by one or more columns.

```
shell-prompt: cat names.txt
Kelso Bob
Cox Perry
Dorian John
Turk Christopher
Ried Elliot
Espinosa Carla

# Sort by entire line
shell-prompt: sort names.txt
Cox Perry
Dorian John
Espinosa Carla
Kelso Bob
Ried Elliot
Turk Christopher

# Sort by second column
shell-prompt: sort -k 2 names.txt
Kelso Bob
Espinosa Carla
Turk Christopher
Ried Elliot
```

```
Dorian John
Cox Perry

Shell-prompt: cat numbers.txt
45
-12
32
16
7
-12

# Sort sorts lexically by default
Shell-prompt: sort numbers.txt
-12
-12
16
32
45
7

# Sort numerically
Shell-prompt: sort -n numbers.txt
-12
-12
7
16
32
45
```

The **uniq** command eliminates adjacent duplicate lines from the input stream.

```
Shell-prompt: uniq numbers.txt
45
-12
32
16
7
-12

Shell-prompt: sort numbers.txt | uniq
-12
16
32
45
7
```

2.24.5 Perl, Python, and other Scripting Languages

All of the commands described above are described by the POSIX standard and included with every Unix compatible operating system.

A wide variety of tasks can be accomplished without writing anything more than a shell script utilizing commands like these.

Nevertheless, some Unix users have felt that there is a niche for tools more powerful than shells scripts and standard Unix commands, but more convenient than general-purpose languages like C, Java, etc.

As a result, a new class of scripting languages has evolved that are somewhat more like general-purpose languages. Among the most popular are TCL, Perl, PHP, Python, Ruby, and Lua.

These are interpreted languages, so performance is much slower than a compiled language such as C. However, they are self-contained, using built-in features or library functions instead of relying on external commands such as **awk** and **sed**. As a result,

many would argue the they are more suitable for writing sophisticated scripts that would lie somewhere between shell scripts and general programs.

Chapter 3

Index

R
rsync, 61

S
ssh_config, 76

U
UI, 10
User interface, 10

V
virtual desktop, 13

W
workspace, 13