C/Unix Programmer's Guide Lab/Discussion Exercises

May 19, 2023

Jason W. Bacon

Contents

List of Figures

May 19, 2023

0.1 Purpose of Lab

0.1.1 Overview

Labs/discussions are primarily a place to practice problem solving and the programming *process*, and discuss code quality and other topics. Part of each lab will be scripted, and part will be open for instructors and students to ask additional questions and do additional examples. Instructors are available for the duration of the scheduled lab time, but how that time is used will vary as needs dictate.

Lab sessions are primarily a place to do two things:

- Interact with the instructor and other students. Everyone should be both a student and a teacher during these sessions.
- Get hands-on practice with the basics, to prepare you for doing homework, programming projects, quizzes and exams individually.

Note These sessions are your opportunity to share knowledge and ideas with other students and the instructor. All work outside of class is to be done individually.

Labs/discussions are not meant to be a repeat of lecture. They are meant to augment lecture by providing students a place where they can take a more active role in learning, ask questions, and share their own ideas. Everyone in a lab/discussion session is both a student and a teacher. This is the place where students and instructors alike should learn from each other.

Each lab/discussion session will cover material up to and including the most recent lecture. They will not introduce new material that has not yet been covered in lecture.

Come to lab prepared. Students MUST be caught up on the lectures and reading before attending labs. Part of each lab will be like an oral quiz, where students will either volunteer or be called on randomly to answer questions in the lab manual. Your participation grade will depend on your ability to answer questions intelligently (though not necessarily perfectly) in lab. Attend all lectures, read ahead in the lab manual to see what questions are upcoming, and read the book to find the answers.

Students may come to the front of the room to work out certain problems on the board, such as number conversions.

Instructors and students should work interactively to answer basic questions and develop and code/test example programs. Instructors do not *lead* the session, they only *guide* it. Think of them as coaches and referees. They act as secretaries when doing programming examples, taking dictation from the students and typing it in for all to see on the projector screen or video conference.

The discussion should not be dominated by a few students. All students should contribute. Those who do not contribute voluntarily will be called on by name at some point. Participation in lab/discussion sessions is required, and part of your grade. Come prepared so that you can contribute something valuable to the discussion, but don't be afraid to be wrong. We learn as much from our mistakes as we do from our successes, often more.

Instructors should help students figure things out on their own. Rather than spoon feed answers, they will guide the students by asking leading questions. An example exchange is shown below. Don't worry if you don't understand the details of this example. Just focus on how the instructor and students interact for now.

```
Student:    What compiler flags should we use?

Teacher:    Where do we find information on compiler flags?

Students:   (After some deep thought and possibly browsing materials)
            man cc

Teacher:    Everyone run "man cc" and see what comes up.
```

```
    Teacher runs "man cc" on the projector / video conference and discusses
    the man page with the students.

Teacher:    What are we trying to accomplish with the compiler flags?

Students:   Faster programs

Teacher:    What flag(s) will help produce faster code?

Students:   (After browsing the man page)
            -O or -O2
            (Student feels somewhat Godlike, knowing how to optimize code)

Teacher:    What else are we trying to accomplish with the compiler flags?

Students:   (Think to selves: There's another purpose?)
            (Ponder and browse materials more)
            Find potential problems in the code.

Teacher:    What flags will help the compiler warn us about problems?

Students:   -Wall

Etcetera...
```

0.1.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

0.1.3 Code Reviews

Once we get into C programming, each lab/discussion will include a code review session. Examples of code are displayed on the projector screen, and both students and instructors will suggest ways to improve it. NO PROGRAM IS FINISHED AS LONG

AS IT HAS ROOM FOR IMPROVEMENT. First, the class will review the lead instructor's code, then the instructor's code, then code from a few students. Every student should have their code reviewed at least once during the semester.

Code reviews are probably *the* most valuable way to learn how to improve your code. Having many eyes on it to spot potential issues dramatically accelerates the debugging process. Learning to accept criticism about your code without affect on your ego is one of the most important steps in becoming a great programmer. Be completely objective. Accept good suggestions without hesitation and reject those that would reduce the quality of your code based on rational analysis. Know that none of the code you will ever write will be perfect. Don't be afraid to put unfinished code out there and invite critique. The perfect is the enemy of the good. What matters is that you have created something potentially valuable, and let improving it be a community effort.

0.2 Course Logistics

Note

Students must bring a laptop to this lab session. It must have a working terminal emulator, an SSH client, and the **rsync** command, which will be used to practice file transfers. All of these tools are preinstalled on macOS, and can be easily installed by the package manager for most BSD and GNU/Linux systems. Windows users can obtain them by installing Cygwin or a Unix virtual machine.

Demonstrating the ability to log into the remote Unix server from your laptop during the first lab session will be the first homework grade.

0.2.1 Connecting to VPN

Access to computers from off campus may require use of a *VPN* (*Virtual Private Network*), which allows a device that is physically off campus to obtain an IP address on the campus network. This allows the device to access resources that are hidden from the Internet. Information on the VPN can be found on your campus IT website. Questions about the proprietary client software provided here should be directed to the IT department. Course instructors are not the best source of information on this, and the IT department should be made aware any time users are having problems with their products.

BSD and Linux users can also use open source VPN clients such as OpenConnect (https://github.com/openconnect) and the graphical GlobalProtect-OpenConnect (https://github.com/yuezk/GlobalProtect-openconnect). The GlobalProtect-OpenConnect client supports 2-factor authentication for GlobalProtect VPN servers.

Figure 1: GlobalProtect-OpenConnect VPN Client

0.2.2 Connecting to the Remote Unix Server

Requirements for Unix access

We will use a Unix server to test all homework assignments for this class. The host name will be provided by your instructor. Students will need the following tools in order to connect to the remote Unix server:

- A terminal emulator. This is a program that pretends to be a terminal. A terminal is basically a keyboard and screen with some sort of communication interface, used to access a remote computer. Since a modern PC has everything a terminal has an more, most people run terminal emulator software on their PC rather than use a hardware terminal. There are many free terminal emulator programs available for all popular operating systems.
- An SSH (Secure SHell) client. SSH is a secure communication protocol used to log into a remote computer using 100% encrypted communication.

These tools are included with most Unix-compatible systems, including BSD, Linux, and macOS. For Windows users, the easiest way to install them is Cygwin, a free Unix-compatibility layer for Windows that installs in about 15 minutes. It can be downloaded from https://www.cygwin.com/install.html.

Cygwin Overview

Cygwin is a very simple and non-invasive way to add a Unix environment to your Windows PC. Cygwin does not install any drivers or other system hacks. All Cygwin files are installed under one directory, separate from Windows system files, so there is no risk of Cygwin interfering with other things your computer. Cygwin provides most of the same basic tools used by real Unix users on your Windows machine. This allows BSD, Linux, macOS, and Windows users to all use the same commands.

Note You will need to install these tools on your laptop BEFORE the first lab session. It will not be possible for all students to install Cygwin during the first lab session. This would overwhelm the WiFi connection with Cygwin downloads. Students using Windows must install Cygwin BEFORE the first lab.

Note All assignments for CS 337 are to be done individually, except this one. The assignment in this case is simply to have a terminal emulator and **ssh** command on your laptop before the first lab session. If someone helped you do this, that's fine as far as credit for the assignment, though it may be a sign that you need to up your game. This should be easy for any computer science student, following the instructions in the lab manual provided on Canvas (c-unix-lab.pdf). Better to ask an instructor or a friend for help than to fail to complete the assignment, though.

Abbreviated Cygwin install instructions for the computer-savvy are below. Detailed instructions for installing Cygwin can be found in Section 30.1. If you need help with this step, schedule a Zoom meeting with your TA. Ideally, multiple students should attend the same Zoom session to save time.

1. Download `setup-x86_64.exe` from https://www.cygwin.com/install.html and SAVE IT TO YOUR DESKTOP (Right-click and select "Save link as").
2. Run `setup-x86_64.exe` by double clicking on the file you saved to your Windows desktop.
3. Accept default answers for most questions, except the following:
   - Select a geographically nearby mirror. The menu doesn't provide many clues about the location of each mirror, but https://mirrors.sonic.net seems fast from Milwaukee.
   - In the "Select Packages" screen, select the following, by clicking on the word "Skip" in the menu. This will change "Skip" to a package version.
     - openssh (in the networking category)
     - rsync (in the networking category)
     - xhost (in the x11 category)
     - xinit (in the x11 category)

After completing the Cygwin setup, you should see the Cygwin Terminal icon on your desktop. Double click on this to start the Cygwin terminal emulator.

Windows OpenSSH

As a fallback option, users with Windows 10 build 1809 or later should have a basic SSH client as part of the Windows installation. You can run the **ssh** command from PowerShell. This will be enough to participate in the first lab. If you find that **ssh** does not work for you under PowerShell, you may need to enable the SSH feature manually: (https://learn.microsoft.com/en-us/windows-server/administration/openssh/openssh_install_firstuse?tabs=gui).

However, this *only* provides a basic **ssh** command and nothing else. To install **rsync** (which you will need for file transfers), X11 graphics capabilities, and other Unix tools, you will need Cygwin (or a Unix virtual machine).

Logging into the server

Once you have a terminal emulator and an SSH client installed, you're ready to remotely log into the remote Unix server. Remember that you need to be connected to the VPN if you are not on campus. Figure 2 shows the **ssh** command being entered into a terminal emulator called **coreterminal** on a FreeBSD system. It will look basically the same on other Unix systems such as other BSDs, GNU/Linux systems, SunOS, etc.

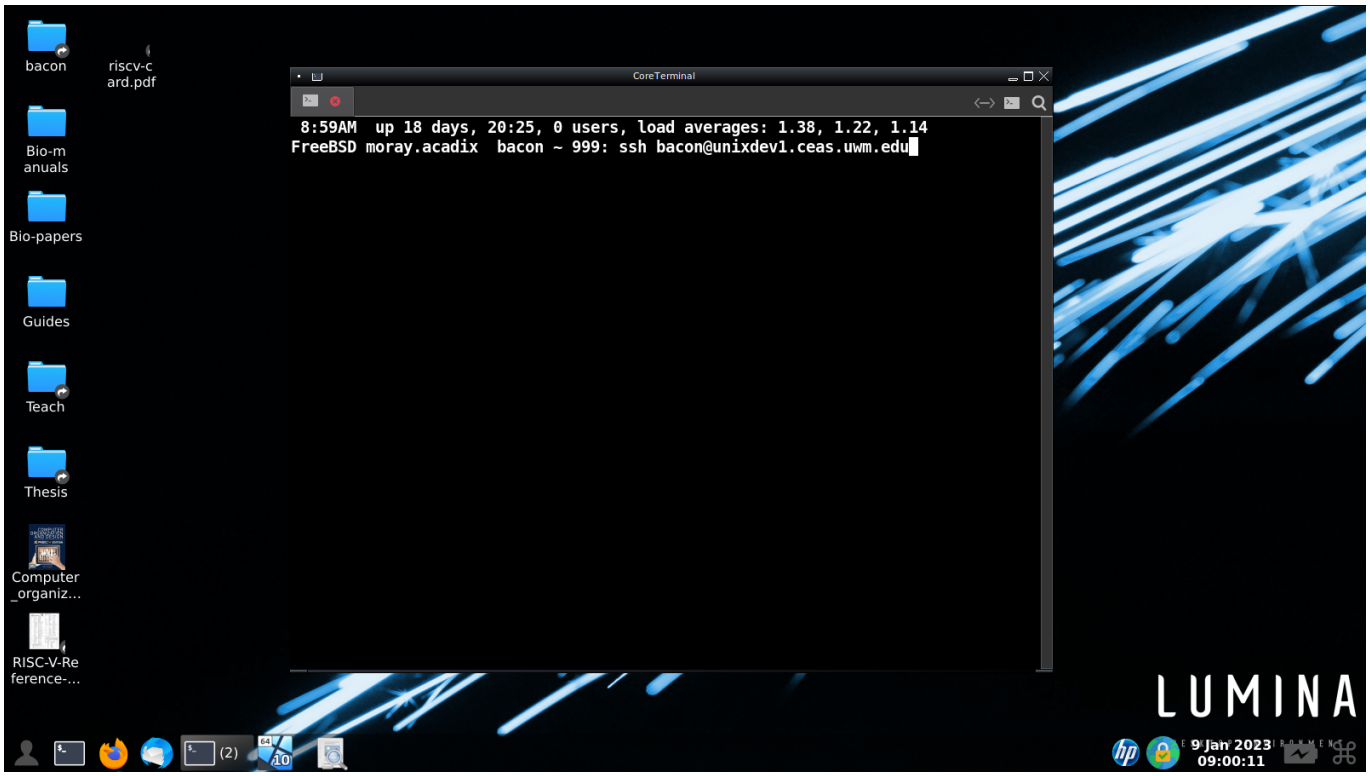


Figure 2: Running SSH in a Unix terminal emulator

Mac users can use the **Terminal** application found in `/Applications/Utilities`. macOS comes with the Figure 3 shows the macOS Terminal application. **ssh** command already installed.

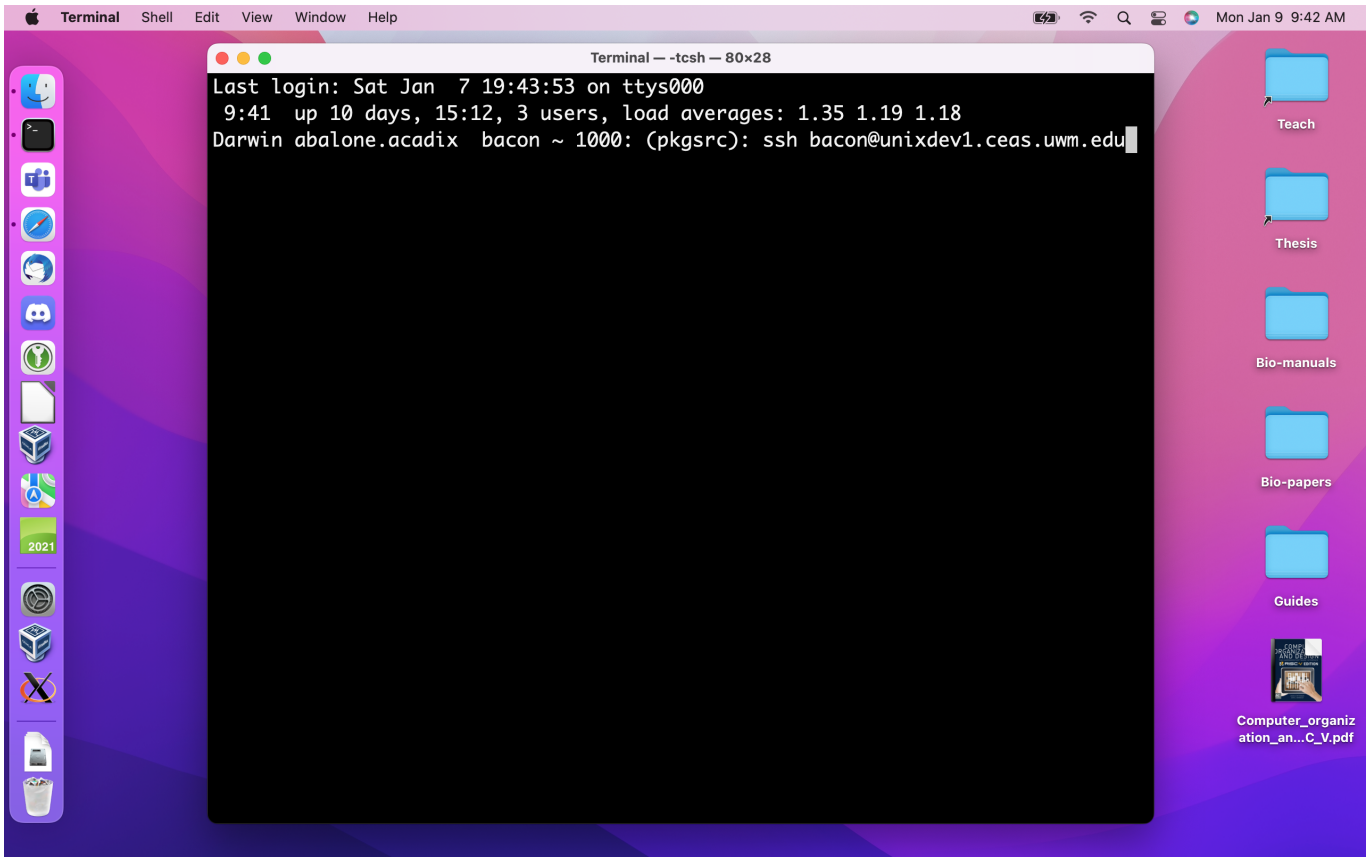


Figure 3: Running SSH in a macOS terminal emulator

Figure 4 shows a Cygwin terminal under Windows 10. As you can see, Cygwin makes the experience on Windows basically the same as a Unix system.

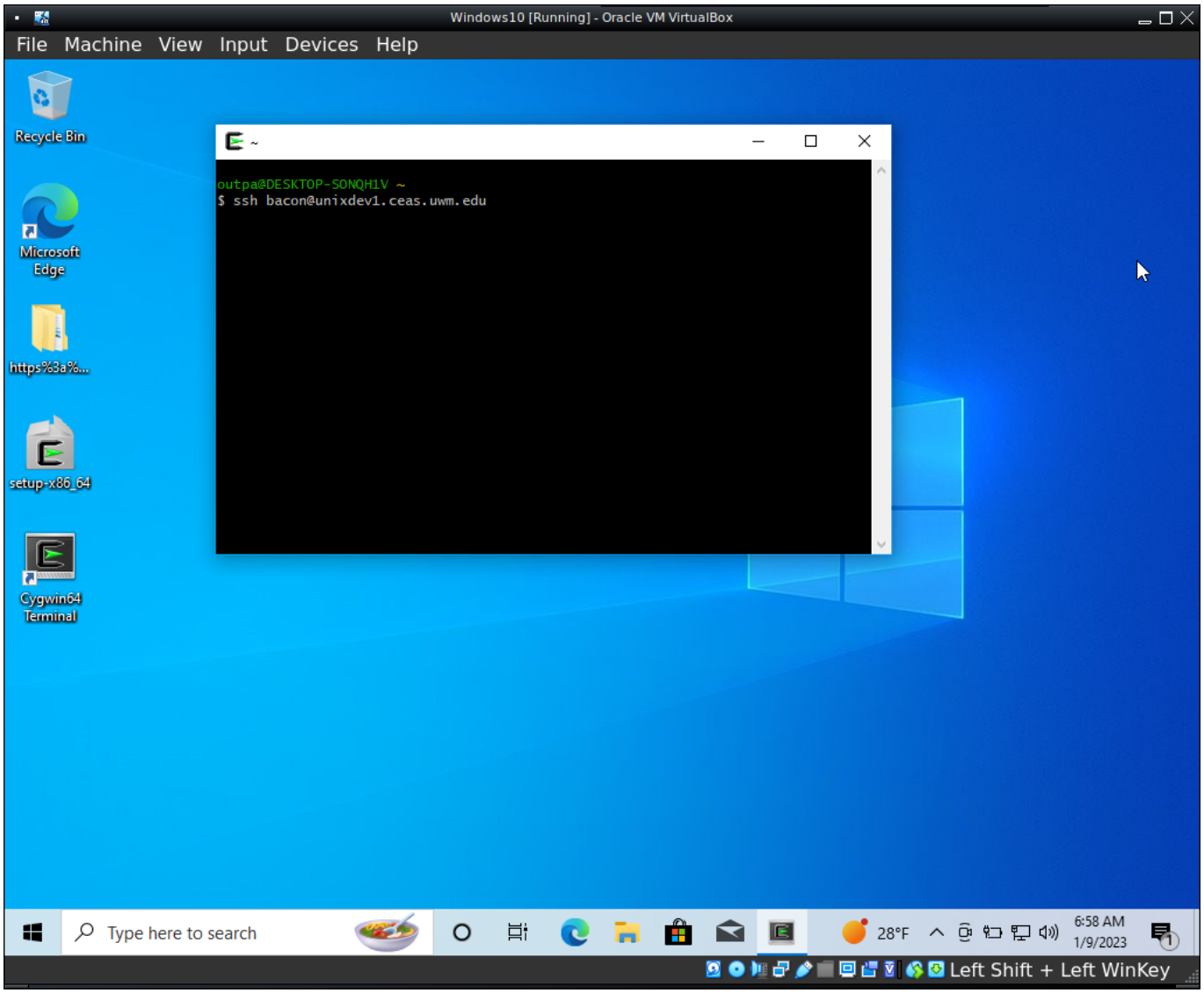


Figure 4: Running SSH in a Cygwin terminal emulator

The TA will demonstrate opening a terminal window and connecting to the remote Unix server using the **ssh** command from a real Unix session or from Cygwin. Students should do the same on their own computers if possible.

Below is a generic **ssh** command used to connect to the remote Unix server. Replace "username" with your account name and "remote.server.edu" with the server URL provided by your instructor. The first time you connect to a new host with **ssh**, you will be warned that the host is unknown and asked if you trust it. You must answer "yes" (not just "y") to proceed.

```
ssh username@remote.server.edu
The authenticity of host 'remote.server.edu (129.89.25.223)' can't be established.
ED25519 key fingerprint is SHA256:hcBdLAYaJnObulMLpmueSlKP4RzepjZJK1AmjcDsML8.
No matching host key fingerprint found in DNS.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
```

0.2.3 The Unix Command Line Interface (CLI)

The CLI may be a new concept that takes a little getting used to. The key to this, like all learning, is time and repetition. Practice early and practice often, so your brain has time to rewire itself. Once you get a feel for the command line, you will see how much

faster it allows you to work.

Unix functionality is far too vast for a menu driven interface such as a GUI. There are thousands of different commands on a typical Unix system, each with anywhere from a few to dozens of options. A menu system encompassing all of this would have many levels and would be a nightmare to navigate. It would also be impossible for developers to maintain.

Example 0.1 Practice Break

Note For this, and all practice breaks that follow, students should do the exercises shown. If you are reading this for a class, then these exercises are meant to be done in class. Try them on your own, do your best to understand what is happening, and ask the instructor for clarification if necessary.

Remotely log into another Unix system using the **ssh** command or PuTTY, or open a shell on your Mac or other Unix system. Then try the commands shown below.
Unix commands are below preceded by the shell prompt "shell-prompt: ". Other text refers to input to the program (command) currently running. You must exit that program before running another Unix command.
Lines beginning with '#' are comments to help you understand the text below, and not to be typed.
Don't worry if you're not clear on what these commands do. You do not need to memorize them right now. This exercise is only meant to help you understand the Unix CLI. Specific commands will be covered later.

```
# Print the current working directory
shell-prompt: pwd

# List files in the current working directory (folder)
shell-prompt: ls
shell-prompt: ls -al

# Two commands on the same line.  A ';' is the same as a newline in Unix.
shell-prompt: ls; ls /etc

# List files in the root directory
shell-prompt: ls /

# List commands in the /bin directory
shell-prompt: ls /bin

# Search the directory tree under /etc
shell-prompt: find /etc -name '*.conf'

# Create a subdirectory
shell-prompt: mkdir -p Data/IRC

# Change the current working directory to the new subdirectory
shell-prompt: cd Data/IRC

# Print the current working directory
shell-prompt: pwd

# See if the nano editor is installed
# nano is a simple text editor (like Notepad on Windows)
shell-prompt: which nano

    If this does not report "command not found", then do the following:

# Try the nano editor.  Nano is an add-on tool, not a standard tool on
# Unix systems.  Some systems will not have it installed.
shell-prompt: nano sample.txt

# Type the following text into the nano editor:
```

```
This is a text file called sample.txt.
I created it using the nano text editor on Unix.

# Then save the file (press Ctrl+o), and exit nano (press Ctrl+x).
# You should now be back at the Unix shell prompt.

# Try the "vi" editor
# vi is standard editor on all Unix system.  It is more complex than nano.
# It is good to know vi, since all Unix systems have it.
shell-prompt: vi sample.txt

    Type 'i' to go into insert mode
    Type in some text
    Type Esc to exit insert mode and go back to command mode
    Type :w to save
    Type :q to quit

    # "ZZ" is a shortcut for ":w:q"

shell-prompt: ls

# Echo (concatenate) the contents of the new file to the terminal
shell-prompt: cat sample.txt

# Count lines, words, and characters in the file
shell-prompt: wc sample.txt

# Change the current working directory to your home directory
shell-prompt: cd
shell-prompt: pwd

# Show your login name
shell-prompt: id -un

# Show the name of the Unix system running your shell process
shell-prompt: hostname

# Show operating system and hardware info
shell-prompt: uname -a

# Today's date
shell-prompt: date

# Display a simple calendar
shell-prompt: cal
shell-prompt: cal 2023
shell-prompt: cal nov 2018
shell-prompt: cal jan 3000

# CLI calculator with unlimited precision and many functions
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
```

```
# Show who is logged in and what they are running
shell-prompt: w
shell-prompt: finger

# How much disk space is used by the programs in /usr/local/bin?
shell-prompt: du -sh /usr/local/bin/

# Copy a file to the current working directory
shell-prompt: cp /etc/profile .
shell-prompt: ls

# View the copy
shell-prompt: cat profile

# View the original
shell-prompt: cat /etc/profile

# Remove the file
shell-prompt: rm profile
shell-prompt: ls

# Exit the shell (which logs you out from an ssh session)
# This can also be done by typing Ctrl+d, which is the ASCII/ISO
# character for EOT (end of transmission)
shell-prompt: exit
```

0.2.4 Remote X11 Graphics

Note Use of remote graphics is not required for CS 337. This information is provided only for interested students and will not be covered on quizzes or exams. More information is available in Section 30.2. Instructors and TAs will be available to answer questions.

The Unix graphics API, called X11, inherently supports running graphical programs remotely. The remote computer sends graphic commands (magic sequences) over the network to the computer where the graphics are to be displayed. This is called *indirect rendering* and is generally much slower than *direct rendering*, where the CPU and GPU (graphics processor) are on the same machine, and the CPU can write directly to the video memory. For some applications, such as the Octave GUI, simply graphing and plotting, etc., indirect rendering is more than fast enough. Some other applications require direct rendering in order to function acceptably. Using a VPN may add an additional bottleneck, making graphics slower than they would be over the same network connection without a VPN.

X11 remote graphics is not a remote desktop system or terminal server. A remote desktop system allows a remote user to commandeer the display on a machine, so that the local user sees the same thing as the remote user. A terminal server starts an entire login session on a remote display. With remote X11 graphics, remote users can run individual applications that display on their PC. Any number of users at different locations can be running individual graphical programs on a server with X11 at the same time.

In order to run graphical programs on the remote server that display output on your computer, you will need **ssh** to forward your X11 display. This is done by simply using the -X option:

```
ssh -X username@remote.server.edu
```

Note TA should run **octave --gui** on a remote machine for demonstration, if possible.

Mac and Cygwin users must also start an X11 server on their own computer. Mac users must install the XQuartz X11 package for macOS from https://www.xquartz.org/. Cygwin users will need the xhost and xinit packages. Then follow the instructions in Section 30.2.3.

Some programs, such as those using OpenGL 3D graphics, require additional permissions on your display. Enabling them reduces the security of the connection and assumes complete trust in those who manage the remote system. To enable this feature, replace -X with -Y:

```
ssh -Y username@remote.server.edu
```

Remote OpenGL graphics is a complicated topic. Some applications will work fine, while others will be difficult to get working at all. Rather than fight it, you might prefer to simply download the data files to your PC and run the OpenGL application there, where it can use direct rendering.

Over slow connections such as WiFi or home cable or DSL, you may benefit from using SSH compression. Do not use this on a fast connection, such as campus Ethernet. Doing so will actually slow down response times. Use -C to enable compression:

```
ssh -C -X username@remote.server.edu
ssh -CX username@remote.server.edu
```

Note After running a remote graphical program, your connection may hang when you log out of the remote system. This is due to a background process that was started to support the graphical communication. Simply type Ctrl+c to terminate it, and the logout will complete.

0.2.5 Optional: Using a Virtual Machine

Note This material is optional and will not be covered in lecture or lab. If you want to try it on your own, instructors are available to answer questions.

A *virtual machine monitor* (*VMM*) is a software package that pretends to be a computer. We can install entire operating systems under a VMM the same way we install them on real hardware.

Use of a VM is not required for CS 337. This information is provided only for interested students and will not be covered in lecture, lab, homework, quizzes, or exams. Instructors and TAs will be available to answer questions.

Windows Services for Linux (WSL) is one such VMM. It is a specialized form of the more general Hyper-V VMM from Microsoft. It allows users to run a Linux system as an application under Windows. Resource requirements are much higher than Cygwin and installation takes much longer. WSL is only for Windows.

Another option is a more portable VMM. There are several VMMs on the market. VirtualBox is unique among them in being open source, highly portable (it runs under BSD, Linux, macOS, SunOS, and Windows) and easy to use.

0.2.6 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. What does VPN stand for?
2. What does a VPN do?
3. What is OpenConnect?
4. What is GlobalProtect-OpenConnect?
5. What is SSH?
6. How do you connect to a remote server called cactus.acme.com as username "wile-e-coyote" using **ssh**?
7. All students must log into unixdev1. The TA and other students can assist if necessary. The TA will generate a list of logged in students by running **w > login-list.txt**. This list will be used to determine credit for the assignment.
8. What must one do to run graphical programs on a remote computer over an SSH connection?
9. Are there any limitations when using remote graphics? Explain.

Chapter 1

Introduction

This material will be discussed in the next lab session after Chapter 1 has been covered in lecture.

1.1 How to Proceed

1.1.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. What are "man pages"?
2. Are man pages good tutorials?
3. How does reading man pages compare with reading a book on the subject?

1.2 Why Use C?

1.2.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. What are some advantages of C over other languages? Explain your answer.
2. Does the simplicity of C limit what we can do with it, compared to "higher level" languages?
3. Can we do object-oriented programming in C?

1.3 Why Use Unix?

1.3.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. What is Unix?
2. Which mainstream operating systems are Unix-compatible?
3. Can we run Unix programs on Windows?
4. How many different Unix compatible operating systems are there today?
5. Is Linux a Unix-compatible operating system?
6. Is FreeBSD a Unix-compatible operating system?
7. Is Apple's macOS a Unix-compatible operating system?
8. What are some advantages of Unix-compatible operating systems over proprietary operating systems? Explain your answer.

Part I

Introduction to Computers and Unix

Chapter 2

Binary Information Systems

2.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. Why is it important to understand binary information systems and other number bases?
2. What is binary?
3. How is binary represented in computer hardware?

4. Why does computer hardware use binary rather than decimal?
5. Define bit, byte, word, longword, shortword, nybble.
6. What is the range of a fixed point decimal system with six total digits and 1 fractional digit?
7. What is the relationship between computer integer systems and fixed point systems?
8. What are two advantages of floating point over an integer system with the same number of bits?
9. What are two disadvantages of floating point vs an integer system with the same number of bits?
10. What is BCD and where is/was it used and why?
11. What is the decimal value of the unsigned binary number 10010011.001_2
12. What is the range of a 15-bit unsigned binary integer system in binary, powers of 2, and decimal?
13. What is an LSB?
14. What is a MSB?
15. What is a higher bit?
16. What are the common sizes of computer integer systems?
17. What is the largest integer we can represent in each size?
18. What binary format do computers use to represent signed integers?
19. What is the decimal value of the 8-bit two's complement value 00101001?
20. What is the binary value of 103_10?
21. What is the decimal value of the 8-bit two's complement value 11110110?
22. What is the range of a 24-bit two's complement system in binary, powers of two, and decimal?
23. Add the 6-bit numbers 010001 + 111100 as unsigned and as two's complement. Show the decimal equivalents on the side. Indicate whether an overflow occurs in each case and how we know by looking at the bits.
24. What are the three components of a floating point number?
25. What is the approximate range of 32-bit IEEE floating point?
26. What is the approximate range of 64-bit IEEE floating point?
27. What is the decimal value of 234.6_8?
28. What is the hexadecimal value of 10010011.010_2?
29. What is the binary value of 134.2_8?
30. What is ASCII? Does it include German or Spanish letters? Elaborate.

2.2 Open Discussion Time

If there is time remaining after the assigned lab exercises, and the lab section is caught up with lecture, students can pose questions on material covered so far. Questions should be *specific* and students should be able to demonstrate that they put a sincere effort into answering it on their own, including giving their own solution. Lab time is not to be used as a substitute for homework or studying outside of class. Students should come to lab ready to contribute to the discussion. Asking lab instructors to reiterate lecture material is not acceptable.

Instructors may also create additional examples to work through *with student participation*, to augment material that students seem to need more practice on. These examples will not overlap the course materials, homework, quizzes, or exams.

Chapter 3

Hardware and Software

3.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. What is a CPU? Be specific.
2. Are different CPUs compatible with each other?
3. What do RAM and ROM have in common?

4. How do RAM and ROM differ?
5. Do all computers have both RAM and ROM?
6. What does volatile mean?
7. What are some examples of I/O devices?
8. How does mass storage differ from RAM?
9. How do SSDs differ from magnetic disks?
10. What is the relationship between machine language and a CPU?
11. What is the relationship between assembly language and machine language?
12. Compare assembly language to high level languages such as C and Java.
13. Is Java a compiled language?
14. What is the relationship between algorithms and programs?
15. Develop a top-down design for a binary search, as a bubble diagram, and a document format using indentation to represent levels of refinement.
16. Develop a flowchart for finding the smallest element in a list.
17. Explain the difference between specification, design, and implementation.
18. When does testing occur in the programming process and how often?
19. Why is frequent testing during implementation necessary?
20. When should a program be considered "finished".
21. What is the major problem with implementing code in a rapidly evolving language?

3.2 Open Discussion Time

If there is time remaining after the assigned lab exercises, and the lab section is caught up with lecture, students can pose questions on material covered so far. Questions should be *specific* and students should be able to demonstrate that they put a sincere effort into answering it on their own, including giving their own solution. Lab time is not to be used as a substitute for homework or studying outside of class. Students should come to lab ready to contribute to the discussion. Asking lab instructors to reiterate lecture material is not acceptable.

Instructors may also create additional examples to work through *with student participation*, to augment material that students seem to need more practice on. These examples will not overlap the course materials, homework, quizzes, or exams.

Chapter 4

Unix Overview: Enough to make you dangerous

4.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. What are the major components of an operating system?
2. What is the relationship between cleverness and wisdom in programming?
3. What is a shell?

4. How is a shell related to a terminal emulator?
5. What is a magic sequence with respect to a terminal?
6. What are the three possible parts of every Unix command?
7. What is the difference between a program and a process?
8. What is another name for a directory? Why is it called a directory?
9. If the CWD is /home/joe/Project1, what is the relative pathname of /home/joe/Project1/sort.c? /home/joe/.cshrc?
10. How do we look up details about the C library function qsort()?
11. Does Unix have a GUI?
12. How do Unix and Windows treat disk partitions differently?
13. How can we list all the files in `/etc` with names that contain "net".
14. How do we see a list of commands we entered recently?
15. How can we save a list of all files in `/etc` to a file called `etc-list.txt`?

```
ls /etc > etc-list.txt
```

16. How can we list all the commands in `/usr/bin` and see them one page at a time?

```
ls /usr/bin | more
```

17. What does device independence do?

The Unix commands below should be discussed and practiced during lab/discussion. Students will need a laptop with a Unix shell environment and/or SSH to access a remote Unix system for this session.

4.2 Some Useful 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 'or' 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.

4.2.1 File and Directory Management

Note Run these commands in the exact order presented. Some depend on successful completion of previous commands.

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
```

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 already exist.

```
shell-prompt: ls
shell-prompt: mkdir Temp
shell-prompt: ls                        # Should see Temp now
shell-prompt: mkdir Temp2/C/MPI      # Should fail
shell-prompt: mkdir -p Temp2/C/MPI
shell-prompt: ls Temp2
```

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.

```
shell-prompt: cd
shell-prompt: touch file            # Create file if it doesn't exist
shell-prompt: cp file file.bak      # Make a backup copy
shell-prompt: ls                    # Should see file and file.bak
```

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 /etc/hosts* hosts  # Should fail
shell-prompt: cp /etc/hosts* Temp   # Should work if directory Temp exists
shell-prompt: ls Temp
```

mv moves or renames files or directories.

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

```
shell-prompt: mv file.bak file.bk
shell-prompt: ls
```

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

```
shell-prompt: mv file file.bk file2     # Should fail
shell-prompt: mv file file.bk Temp      # Should work if directory Temp exists
shell-prompt: ls
shell-prompt: ls Temp
```

rm removes one or more files.

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

```
shell-prompt: cd Temp
shell-prompt: ls
shell-prompt: rm hosts*
shell-prompt: ls
shell-prompt: rm file*
shell-prompt: ls
```

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

If there are multiple hard links to a file, removing one of them only removes the link, and remaining links are still valid.

Caution Removing the path name to which a symbolic link points will render the symbolic link invalid. It will become a *dangling link*.

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 via a package manager.

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

```
shell-prompt: df
```

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, so changes made through one name (e.g. using nano) appear in the other.

Each file in a typical Unix file system is described by a structure called an *inode*. The inode contains *metadata*, i.e. information about the file other than its content, such as the file's ownership, permissions, last modification time, and the locations of the disk blocks (chunks of disk space) containing the file's content.

Without `-s`, a standard directory entry, known as a *hard link* is created. A hard link is a directory entry that points directly to the inode of the file. In fact, such a directory entry contains little more than the file's name and the location of the inode. Every file must have at least one hard link to it. For this reason, removing a file is also known as "unlinking".

```
shell-prompt: touch file
shell-prompt: ln file file.hardlink
shell-prompt: ls -l
```

To create a second hard link, the source cannot be a directory, and the source and destination path names must be in the same file system. There is no harm in trying to create a hard link. If it fails, you can do a soft link instead.

```
shell-prompt: ln /etc .         # Should fail
shell-prompt: ln -s /etc .
shell-prompt: ls
shell-prompt: ls etc            # List the link
shell-prompt: ls etc/           # List contents of the directory
```

File systems under Windows appear as different drive letters, such as C: or D:. Under Unix, each file system is *mounted* to a specific directory. The main file system is mounted to `/` and the rest are mounted to subdirectories. The **df** command will list file systems and their mount points within the directory tree. For example, in the **df** output below, `/` and `/data` are separate file systems. The disk `ada0` is divided into three *partitions*. Partition 2, called `ada0p2`, contains a file system which is mounted on `/`. Partitions 0 and 1 are used by the operating system for other purposes. The second disk, `ada1`, has a file system on partition 0, which is mounted on `/data`.

```
shell-prompt: df
Filesystem         Size    Used   Avail Capacity  Mounted on
/dev/ada0p2        447G    266G    146G    64%    /
/dev/ada1p0        978G    172G    729G    20%    /data
```

Everything under `/data` and only things under `/data` are on `ada1p0`. Hence, we cannot create a hard line to `/data/joe/Research/notes.txt` in `/home/joe`, which is on `ada0p2`.

```
# This will fail.
# You cannot run this command, since the partitions are hypothetical
# You can try linking something from a different filesystem based on
# your own "df" output if you like.
shell-prompt: ln /data/joe/Research/notes.txt ~joe
```

With `-s`, a *symbolic link*, or *soft link* is created. A symbolic link is not a standard directory entry, but a pointer to another path name. It is a directory entry that points to another directory entry rather than the inode of the file. Symbolic links to not have to be in the same file system as the source.

```
# This will work
shell-prompt: ln -s /data/joe/Research/notes.txt ~joe
```

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**. To remove a directory and everything under it, use **rm -r directory**.

```
shell-prompt: cd
shell-prompt: rmdir Temp2               # Should fail
shell-prompt: rmdir Temp2/C
shell-prompt: rmdir Temp2/C/MPI
shell-prompt: rm -r Temp2
shell-prompt: rmdir Temp                # Should tail
shell-prompt: rm -r Temp
shell-prompt: ls
```

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. The `-h` asks for human-readable output with gigabytes followed by a G, megabytes by an M, etc.

```
shell-prompt: du -sh /etc
```

Note

The **du** command does *not* add up file content sizes. It adds up the disk space used by each file. In an uncompressed file system, space used is rounded up to a multiple of the block size (commonly 4096 bytes). In a compressed file system, space used is a multiple of blocks used after compression, which can be significantly smaller than the file content. This is often the case with the ZFS file system, which is standard on FreeBSD and Solaris-based systems such as OpenIndiana. "Fluffy" text files that compress easily, such as genomic data, may require only a small fraction of their content size in disk space on ZFS. This make ZFS a great choice for housing genomic data.

4.2.2 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.

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

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 to which you want to return. This is like leaving a trail of bread crumbs in the woods to retrace your path back home, except the pushd stack will not get eaten by birds and squirrels, and you won't end up in a witch's soup pot.

Example 4.1 Practice Break

Try the following sequence of commands:

```
shell-prompt: pwd          # Check starting point
shell-prompt: pushd /etc
shell-prompt: more hosts
shell-prompt: pushd /home
shell-prompt: ls
shell-prompt: popd         # Back to /etc
shell-prompt: pwd
shell-prompt: more hosts
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, which sends an EOT character (end of transmission, ASCII/ISO character 4) to the shell.

4.2.3 Simple Text File Processing

cat echoes the contents of one or more text files.

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

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

The **vis** and **cat -v** commands display invisible characters in a visible way. For example, carriage return characters present in Windows files are normally not shown by most Unix commands. The vis and cat -v commands will show them as '^M' (representing Control+M, which is what you would type to produce this character).

```
shell-prompt: cat sample.txt
This line contains a carriage return.
shell-prompt: vis sample.txt
This line contains a carriage return.\^M
shell-prompt: cat -v sample.txt
This line contains a carriage return.^M
```

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

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

If the flag -n 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 /etc/hosts
```

The **head** command can also be useful for generating small test inputs. Suppose you're developing a new program or script that processes genomic sequence files in FASTA format. Real FASTA files can contain millions of sequences and take a great deal of time to process. For testing new code, we don't need much data, and we want the test to complete in a few seconds rather than hours. We can use **head** to extract a small number of sequences from a large FASTA file for quick testing. Since FASTA files have alternating header and sequence lines, we must always choose a multiple of 2 lines. We use the output redirection operator (>) to send the head output to a file instead of the terminal screen. Redirection is covered in Section 4.4.

```
# You cannot run this command unless you have a file called
# reall-big.fasta in the CWD
shell-prompt: head -n 1000 really-big.fasta > small-test.fasta
```

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 the flag -n 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 /etc/hosts
```

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.

The `-u` flag asks for *unified diff* output, which shows the removed text (text in the first file by not the second) preceded by '-', the added text (text in the second file but not the first) preceded by a '+', and some unchanged lines for context. Most people find this easier to read than the default output format.

```
shell-prompt: printf "1\n2\n3\n" > input1.txt
shell-prompt: printf "2\n3\n4\n" > input2.txt
shell-prompt: diff input1.txt input2.txt
shell-prompt: diff -u input1.txt input2.txt
shell-prompt: rm input1.txt input2.txt
```

4.2.4 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 many graphical programs require a fast connection (100 megabits/sec) or more to function comfortably.

Knowing how to use a terminal-based text editor is therefore a very good idea, so that you're prepared to work on a remote Unix system 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 available 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.

Eclipse is a popular open-source graphical IDE written in Java, with support for many languages. It is sluggish over a slow connection, so it may not work well on remote systems over ssh.

4.2.5 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.

4.2.6 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 allowing someone to see the length of your password is a bad idea from a security standpoint.

```
# This may not work on systems using an authentication service
# rather than local passwords
shell-prompt: passwd
```

The **passwd** command is generally only used for setting local passwords on the Unix machine itself. Many Unix systems are configured to authenticate users via a remote service such as *Lightweight Directory Access Protocol* (*LDAP*) or *Active Directory* (*AD*). Changing LDAP or AD passwords may require using a web portal to the LDAP or AD server instead of the **passwd** command.

4.2.7 Terminal Control

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

```
shell-prompt: clear
```

reset resets your terminal to its default state. This is useful when your terminal has been corrupted by bad output, such as when attempting to view a binary file with **cat**.

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
```

4.3 Open Discussion Time

If there is time remaining after the assigned lab exercises, and the lab section is caught up with lecture, students can pose questions on material covered so far. Questions should be *specific* and students should be able to demonstrate that they put a sincere effort into answering it on their own, including giving their own solution. Lab time is not to be used as a substitute for homework or studying outside of class. Students should come to lab ready to contribute to the discussion. Asking lab instructors to reiterate lecture material is not acceptable.

Instructors may also create additional examples to work through *with student participation*, to augment material that students seem to need more practice on. These examples will not overlap the course materials, homework, quizzes, or exams.

4.4 Unix Input and Output

Redirection is covered in the textbook and lecture outline.

Part II

Programming in C

Chapter 5

Getting Started with C and Unix

5.1 Coding Standards

5.1.1 Purpose of Coding Standards

Coding standards exist to save us time and make our work easier and more pleasant. Poorly written code that does not work properly or is difficult to read wastes the user's time by disrupting their work. Code that is difficult to read wastes the programmer's time by making them expend unnecessary effort to understand what it is doing.

There are multiple valid opinions about exactly what good code looks like. However, instructors need to provide clear and unambiguous instructions for all students in order to grade assignments consistently and fairly. If you don't agree with everything in these guidelines, that's fine, but do it this way for this class and feel free to follow different standards elsewhere.

5.1.2 Variable Names

Variable names should clearly state what the variable contains *as specifically as possible*. Do not use abbreviations to save a little typing. This strategy is just laziness and almost always backfires by making you and other readers waste time wondering what the variable represents later.

```
int     c;          // This is meaningless
int     cnt;        // Ambiguous abbreviation
int     count;      // Vague
int     exon_count; // Clearer
```

Amusing Aside
In the early days of home computers, users were actually encouraged to use single-letter variable names in order to save memory. Computers at that time typically had at most 64 KiB of RAM and were mostly programmed in BASIC. Since BASIC is an interpreted language, each variable name in the program is a string variable in the interpreter. Longer variable names take up more memory, and with only 64 KiB, running out of RAM was a common problem. Using cryptic variable names was one of the "solutions". The good old days...

5.1.3 Code Structure

Code inside any control construct, such as a conditional, a loop, or a function, must be indented *consistently*. The number of columns added for each indentation level is a matter of opinion, but the indentation must be consistent. Four spaces is a common standard. This is half of a tab stop, so every second indentation level can use tab characters for convenience.

You can use the **indent** command to reformat code that that was not well-formatted when written. Suggested options are as follows:

```
shell-prompt: indent -bad -bbb -bl -nce -i4 -npsl file.c
```

Note The **indent** command will handle most, but not all formatting. You must still inspect your code after using **indent** to ensure that it meets coding standards.

Lines longer than about 80 columns should be broken up to prevent wrapping. Don't assume that all programmers will use a wide window to view/edit the code.

5.1.4 Code Sectioning

Logical sections of code, such as if blocks and loops, should be separated from each other by a single blank line and a block comment describing what the block does.

```
    // Cramming blocks together makes code hard to read

    for (c = 0; c < MAX; ++c)
    {
        // Some code
    }
    if ( sum > MAX_SUM )
    {
        // Some code
    }
```

```
    // Separating blocks and documenting them with a small block comment
    // makes the code easier to look at

    /*
     * Compute the sum of all the read counts
     */

    for (c = 0; c < MAX; ++c)
    {
        // Some code
    }

    /*
     * Error out of the sum does not make sense
     */

    if ( sum > MAX_SUM )
    {
        // Some code
    }
```

Variable initializations required by a conditional or loop should be done immediately before the loop, not separated from it by other code. Keeping related code together is called *cohesiveness*. It saves us time searching for all the elements needed to understand what the code is doing, hence making the code easier to read. Initializing in the variable definition above is tempting to save a line of code, but not worth the pain is causes when trying to decipher code that you wrote a long time ago or was written by someone else.

```
    // Pain: The reader probably has to scroll up from the loop to see
    // if max_read_count was initialized properly.

    int     max_read_count = 0;
```

```
// 30 lines of code

for (c = 0; c < MAX; ++c)
{
    if ( read_count > max_read_count )
        max_read_count = read_count;
}
```

```
// No pain: The reader can easily see everything pertaining to the loop

int     max_read_count;

// 50 lines of code

max_read_count = 0;
for (c = 0; c < MAX; ++c)
{
    if ( read_count > max_read_count )
        max_read_count = read_count;
}
```

5.1.5 Comments

Comments should add value to what the reader is seeing. To do so, they must explain *why* the code is doing what it is doing, not *what* the code is doing. The reader can see what the code is doing by looking at the code. Useless comments that just paraphrase the code won't help them *or* you.

The goal is to maximize value and minimize words. Comments should not say anything that is apparent from the code itself, but should clearly explain, as briefly as possible, what is not obvious.

```
// A completely useless comment that simply echoes the code.
// It does not clarify WHY read_count is being added to sum.
// This won't help you understand what the code is doing a month
// from now when you come back to fix a bug and it won't help
// other people understand it the first time they read the code.
// The reader will have to waste time reading more code
// (reverse-engineering) since this comment does nothing to clarify it.

sum += read_count;      // Add read count to sum
```

```
// This saves the reader from wasting time reading more code to figure
// out WHY the sum is being computed.

sum += read_count;      // Need sum to compute average later
```

```
// Better than the first comment above, but not as good as the second.
// Still contains fluff that just echoes the code "Add read_count to sum",
// in addition to the text vaguely indicating WHY sum is being computed.
// Reading fluff like this for 8 to 12 hours per day adds up to a lot
// of wasted time and fatigue.

sum += read_count;      // Add read_count to sum for average later
```

Comments to the right of code should be separated from the code and aligned with each other, not crammed up against the code.

```
// Unreadable

discrim = b * b + 4.0 * a * c;//Compute value under radical
if ( discrim >= 0 )//Square root exists
```

```
    {

    // Readable

    discrim = b * b + 4.0 * a * c;  //Compute value under radical
    if ( discrim >= 0 )             //Square root exists
    {
```

Every function should have a block comment above it describing the function INTERFACE. This comment should NOT describe how the function works internally. The algorithms used inside may change and this should not necessitate changing the block comment to correspond. The algorithms should be described by the internal comments embedded in the code. Place one or two blank lines before the block comment (your preference, but be consistent) and one blank line after.

```


/*****************************************************************************
 *  Description:
 *      Compute the square root of any non-negative value n.
 *
 *  Arguments:
 *      n:  Number for which the square root is returned
 *
 *  Returns:
 *      The square root of n if n >= 0, or -1 if n < 0
 *
 *  History:
 *  Date        Name        Modification
 *  2023-04-30  Jason Bacon Begin
 *****************************************************************************/

double  sqrt(double n)

{
    // Code to compute sqrt
}


```

A short block comment should appear above each loop and conditional explaining what the code block does in general. Details should be explained by internal comments inside the loop. Place one blank line before and after the block comment.

```
    /*
     *      Estimate a square root using the Babylonian method.
     *      This is a numerical analysis method that computes successively
     *      better guesses given a reasonable initial guess.
     */

    do
    {
        guess = next_guess;

        // Babylonian formula for next guess
        next_guess = (guess + x / guess) / 2.0;

        // Loop until difference between guesses <= tolerance
        // Note: Using the fabs() function entails function call overhead.
        // A better solution would be a macro, which is covered in the
        // cpp chapter.  Iterative functions like this are expensive enough,
        // so we should do all we can to make them more efficient.
    }   while ( fabs(next_guess - guess) > tolerance );
```

5.1.6 Function Names

Function names should clearly state what the function does *as specifically as possible*.

If you cannot describe what a function does using a simple name, then the function is not cohesive. It is doing more than one thing.

When doing object oriented programming in C or any other non-OOP language, functions that are part of a class should have the class name as a prefix, followed by an underscore.

```
// Member of a class that processes variant call files
int     vcf_read(vcf_t *call, FILE *infile)

{
}
```

The underscore serves the same purpose as :: in C++:

```
int     vcf :: read(FILE *infile)

{
}
```

5.1.7 Picking off the Lint

There are various programs available for analyzing style and formatting of programs written in most popular languages. The **cpplint** and **splint** commands are popular tools for checking C and C++ code.

The names are derived from the analogy of picking lint (loose fuzz) off of your sweater.

5.1.8 Error Handling

Check for *all* possible error conditions, except for failed writes to a terminal screen. Every input, every memory allocation, etc. must be checked for success.

When bad input is received, tell the user what was expected, not just that the input was bad.

```
    // Not very helpful
    if ( scanf("%d", &total_lemmings) != 1 )
    {
        fputs("Bad input.  Please try again.\n", stderr);
        exit(EX_DATAERR);
    }

    // Much more helpful
    if ( scanf("%d", &total_lemmings) != 1 )
    {
        fputs("Bad input.  Please enter a positive integer.\n", stderr);
        exit(EX_DATAERR);
    }
```

5.1.9 Testing

Each program should compile without errors or warnings using **cc -Wall**. It must produce correct output when given correct input, and must produce a meaningful error message that tells the user what was wrong with any bad input, and what good input should look like.

```
Please enter your age in years: Bob
Error: "Bob" is not a number.
```

5.1.10 Code Size

Most programs are much longer and more complicated than they need to be. Think before you code to avoid this. Always look for ways to eliminate useless code and make existing code more elegant. NEVER BE SATISFIED THAT THE CODE WORKS. Working to streamline this code will prepare you to write more elegant code next time, which will save you time for the rest of your career.

Code size will be checked visually and with the **cloc** (Count Lines Of Code) command. Excessively long programs will not receive high grades.

Do not, however, go overboard to save a few characters by using clever tricks that make the code cryptic.

```
// Cryptic and not usefully shorter
// This is just showing off one's cleverness for no actual benefit
while ( c )
{
    // Some code...
    --c;
}

// Readable
while ( c > 0 )
{
    // Some code...
    --c;
}
```

5.1.11 Speed

Programs will be graded by compiling them with `-Wall -O` and running them under the **time** command. Programs that take much longer than the instructor's program will not receive high grades.

Make sure you are using efficient algorithms and eliminating useless code. Shorter code is usually faster code. There are exceptions, but they are rare.

Don't recompute things, e.g.

```
for (exponent = 0; exponent < 100; ++exponent)
    printf("%f\n", pow(2.0, exponent));
```

Each call to `pow()` recomputes 2.0 to the previous exponent and does one more multiplication. It performs 1 + 2 + 3 + ... + 98 + 99 multiplications. We can achieve the same results with only 99 total multiplications:

```
for (exponent = 0, power = 1.0; exponent < 100; ++exponent)
{
    printf("%f\n", power);
    power *= base;
}
```

5.1.12 Makefile

This can be reviewed later, after studying Makefiles. It is included here so that all of the coding standards are in one place for easy review.

Use standard variables, like CC, CFLAGS, and LDFLAGS.

Use portable defaults.

```
# Not portable
CC =    gcc
CC =    clang

# Portable: cc is gcc on Linux, cc is clang on FreeBSD, macOS
CC =    cc
```

Set them conditionally, so that the Makefile respects the user's environment and command-line options. User can run **make CC=clang15** to compile with clang version 15 instead of the default compiler.

```
# Rude: Uses cc even if the user wants to use something else, like icc
CC =    cc

# Rude: Prevents user from controlling optimizations
CFLAGS = -Wall -O2

# Respectful: Sets to cc only if the user doesn't specify
CC ?=   cc

# Respectful: Lets user control optimization and adds -Wall
CFLAGS ?=   -O2
CFLAGS +=   -Wall
```

5.1.13 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. Why do coding standards exist?
2. What constitutes a quality variable name?
3. What is the most important aspect of indentation?
4. What is code sectioning?
5. What is cohesiveness?
6. How do variable initializations relate to cohesiveness?
7. What constitute a useful comment?
8. What constitutes a meaningful function (subprogram, procedure, method) name?
9. How do function names relate to cohesiveness?
10. How do we disable annoying compiler warnings?
11. How long are most existing programs?
12. What are the advantages of more concise code?
13. What are the advantages of faster code?
14. How does code size relate to speed?
15. What should we use as the default C compiler in a Makefile?
16. What make variable should be used for C compiler flags?

5.2 Book Content

5.2.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. For what kinds of programming is C used?
2. Is learning C a waste of time for someone who will need to use C++?
3. What is the major difference between C and C++?
4. Can you do object oriented programming in C?
5. What goes in a header file?
6. Discuss the factors involved in program performance, from most effective to least effective.
   (a) Algorithm
   (b) Programming language (compiled languages are orders of magnitude faster than interpreted)
   (c) Code optimizations (integers vs floating point, minimizing memory use to better utilize cache, etc.)
   (d) Parallelism can sometimes improve speed by orders of magnitude, but often does not help at all, and is very expensive (difficult programming).
   (e) Hardware. Buying a faster computer generally has the highest cost/benefit ratio of any approach. It is unfortunately the only option when using closed source (e.g. commercial) software.

7. How do we intelligently optimize a program?
8. Does C detect out-of-bounds array subscripts and integer overflows? Why or why not?
9. Do interpreted languages face the same penalties for checking array bounds, etc.?
10. What are the advantages of an IDE over a simple text editor?
11. Discuss the stages of compiling a C program. Do we have to do them all manually?
12. What does the object code optimizer do and how do we enable it?
13. What is -Wall? Should we avoid it since it's annoying?
14. What does a lint tool do?

5.3 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

5.4 Team Coding Example

The instructor will type in and run the following program using any text editor, then exit the editor for compilation.

Before the code is entered, the class should discuss what would be an ideal filename for the source code and the executable.

```
#include <stdio.h>
#include <sysexits.h>
#include <math.h>

int     main()

{
    double  angle, sine;
```

```
    fputs("Enter an angle in degrees: ", stdout);
    if ( scanf("%lf", &angle) == 1 )
    {
        printf("The sine of %f is %f\n", angle, sin(angle * M_PI / 180.0));
        return EX_OK;
    }
    else
    {
        fputs("Error reading angle.\n", stderr);
        return EX_DATAERR;
    }
}
```

5.4.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. Discuss what compile command should be used to build the program. This includes the command name, the flag arguments and their purpose.
2. Are there any errors or warnings produced by the compiler?
3. Run **cpplint** and **splint** on the program and discuss the results. Can most of the warnings be eliminated using reasonable edits?

Chapter 6

Data Types

6.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. What does a variable definition do?
2. What does a variable declaration do?
3. What problems will the following code cause, and how can we improve it?

```
    double      r, a;

    scanf("%lf", &r);
    a = 3.14 * r * r;
    printf("Area = %f\n", a);
```

4. How much memory does an `int` use?
5. What is the range of an `int`?
6. When should we define a variable as `int`?
7. Define "large" array.
8. What is the disadvantage to using `short` in order to save memory?
9. What is the disadvantage to using `long` in order to save memory?
10. When should we use floating point instead of integer types?
11. How do we choose between `float` and `double`?
12. What is the data type and decimal value of each of the following?
    (a) 0100
    (b) 0xfful
    (c) '1'
    (d) -4.53e2
    (e) 3/4
13. What is the optimal type for a loop variable that will range from -100 to +100.
14. What is the optimal type for a loop variable that will range from 0 to 50,000.
15. What is the optimal type for a large array of house prices with maximum of $10,000,000?

6.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

6.3 Group Coding Example

1. Write a program that inputs the dimensions of a triangle and computes the area. The dimensions can range from 1 meter to 100 meters, in increments of 0.01 meters. Truncate the answer to the nearest meters^2.

```
Please enter the base length in centimeters: 200
Please enter the height in centimeters: 200
The area of the triangle is 2 m^2

Please enter the base length in centimeters: 300
Please enter the height in centimeters: 300
The area of the triangle is 4 m^2
```

6.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

6.5 Code Review Example

For this week, we will review a contrived example.

- Find as many ways as possible to improve the program below:

```c
#include <stdio.h>

int     main()

{
    int     x;

    puts("Enter the distance from the sun in miles: ");
    scanf("%d", &x);
    printf("Distance is %d\n", 22 / 7 * x * x);

    return 0;
}
```

The instructor will show the solutions after the class runs out of ideas.

Chapter 7

Simple Input and Output

7.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. What is the difference between stream I/O and low-level I/O?
2. Describe the three standard streams (one per student).
3. What header file do we need to use the standard stream functions?

4. What functions do we use to input or output a single character?
5. What kind of buffering do streams use by default for terminal I/O? For file I/O?
6. When should we use `gets()`?
7. What is a good way to input a string without storing the newline character in the array?
8. What placeholder should we use to print an `int` in decimal?
9. What placeholder should we use to print a `short` in decimal?
10. What placeholder should we use to print a `char` in decimal?
11. What placeholder should we use to print a `long` in decimal?
12. When should we use `printf()` to print a string?
13. When should we use `scanf()` to input a string?
14. Why do we pass addresses to `scanf()`?
15. What placeholder should we use to read an `int` in decimal?
16. What placeholder should we use to read a `short` in decimal?
17. What placeholder should we use to read a `char` in decimal?
18. What placeholder should we use to read a `long` in decimal?
19. Where do we get a complete list of format specifiers for `printf()` and `scanf()`?
20. What is caveman debugging? Should we use it?

7.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

7.3 Team Coding Example

1. Write a program that asks the user for the dimensions of a box and prints the volume and surface area.

7.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

7.5 Code Review Example

For this week, we will review a contrived example.

- Find as many ways as possible to improve the program below:

```c
#include <stdio.h>
#include <sysexits.h>
#include <stdlib.h>

int     main()

{
    float   a, b, c, r1, r2;

    printf("Enter the coefficients: ");
    scanf("%f %f %f", &a, &b, &c);
    r1 = -b + sqrt(b * b - 4 * a * c);
    r2 = -b - sqrt(b * b - 4 * a * c);
    printf("%f %f\n", r1, r2);

    return 0;
}
```

The instructor will show the solutions after the class runs out of ideas.

Chapter 8

Statements and Expressions

8.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. What is a statement in C?
2. What is an expression in C?
3. What does polymorphic mean?

4. What is the value of y after each of the following?

```
int     a = 1, b = 2, y;

y = 3 / 4;
y = 3.0 / 4.0;
y = 3 / 4.0;
y = 3.0 / 4;
y = a++;
y = ++b;
```

5. What are promotion and demotion?
6. When do promotions occur?
7. When do demotions occur?
8. Which is the higher ranking type in each of the following pairs?
   (a) int, unsigned int
   (b) unsigned short, int
   (c) long, short
   (d) float, double
   (e) double complex, double
   (f) long long, float
9. List the steps involved in evaluating the following expression:

```
unsigned    a = 3;
long        b = 6;
double      x = 4.0;
int         y;

y = a / x + a / b - b / 5 + 2.5;
```

10. How can we replace the integer divisions above with real divisions?
11. When are char and short values promoted?
12. Where are promoted values stored?
13. What is the hexadecimal value of `y` after each of the following statements? Convert to binary if necessary.

```
unsigned char   y = 0x1f;
char            x = 0x80;

y >>= 1;
y = x >> 3;
y = ~y;
y &= 0xf8;
y |= 0xf8;
y ^= 0xf8;
```

14. What operator and mask would we use to clear bits 0 to 3 of a `short`?
15. What operator and mask would we use to set bits 0 to 3 of a short?
16. Rewrite the following statement to make it faster:

```
y = 2.3 * x * x - 4.5 * x * x + 1.9 * x - 7.0;
```

8.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

8.3 Team Coding Example

1. A company makes ball bearings sized as any integer number of mm from 3 to 30. They need an accurate value for the volume in order to price them according to the amount of steel required for each bearing. Write a C program that inputs the radius of a ball bearing as an integer, and reports the volume to two decimal places. Discuss what data types should be used for optimum efficiency and why.

   The instructor will show the solution after the class develops their own.

8.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

8.5 Code Review Example

For this week, we will review a contrived example.

- Find as many ways as possible to improve the program below:

```
#include <stdio.h>
#include <sysexits.h>

int     main()

{
    int     a;
    float   x, y;

    printf("Enter a: ");
    scanf("%d", &a);
    printf("Enter x: ");
    scanf("%f", &x);

    y = 3 / 4 * x * x * x + 2 / 3 * x * x + 1 / 2 * x + a * 64;
    printf("%f\n", y);

    return EX_OK;
}
```

The instructor will show the solutions after the class runs out of ideas.

Chapter 9

Decisions with If and Switch

9.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. How many kinds of statements are there in C?
2. Describe four Boolean operators.
3. What type of expression can be used to control an `if-else` statement in C?

4. What are De Morgan's rules?
5. Simplify the following `if` statement:

```
if ( ! ((x > 10) && (y > 10)) )
    statement;
```

6. What's wrong with the following code?

```
double  x, y;

if ( x != y )
    statement;
```

7. Is it wrong to use '=' in an `if` statement?
8. Write a `printf()` statement that uses a conditional operator to make a word plural when appropriate.

```
We have 1 artichoke remaining.
We have 2 artichokes remaining.
```

9. Why does either the `if` clause or the `else` clause have to be faster than the other?
10. We need to do a large number of calculations that check whether the ocean depth is less than 500 meters or more than 1000 meters. Does it matter which condition we check first? If so, why, and how do we make the choice?
11. Which of the following is "better" code?

```
if ( (income >= 0) && (income < 1000000000) )
```

```
// Income may be less than 0, but we cast to unsigned to eliminate
// a comparison
if ( (unsigned)(income < 1000000000) )
```

9.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

9.3 Team Coding Example

1. Write a C program that asks the user for an airline flight distance in km and a desired trip time in minutes, and reports the minimum average ground speed required to arrive within the desired time. The program should ensure that all inputs are reasonable and print a descriptive error message if they are not.

   The instructor will show the solution after the class develops their own.

9.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

9.5 Code Review Example

For this week, we will review a contrived example.

- Find as many ways as possible to improve the program below:

```c
#include <stdio.h>

int     main()

{
    int     f, c;
    printf("What is the temperature in Fahrenheit? ");
    scanf("%d", &f);
    c = 5 / 9 * f - 32;
    printf("C = %d\n", c);
    return 0;
}
```

The instructor will show the solutions after the class runs out of ideas.

Chapter 10

Loops

10.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. What are some ways we can improve the performance of loops?
2. What is the execution path?
3. How does a `do-while` differ from a `while`? Explain.

4. Why should we not do direct comparisons of floating point expressions in a loop condition?
5. What is the most cost-effective way to optimize programs with loops?

10.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

10.3 Team Coding Example

1. Write a C program that prints the sine of every angle from 0 to 360 in increments of 0.1. Make sure it terminates at exactly the right angle.

   The instructor will show the solution after the class develops their own.

10.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

10.5 Code Review Example

For this week, we will review a contrived example.

- Find as many ways as possible to improve the program below, which reads a list of ages terminated by a sentinel value of -1 and prints some statistics.

```
#include <stdio.h>
#include <sysexits.h>

int     main()

{
    double          sum = 0.0, num, avg, low = 1000, high = 0;
    unsigned long   c = 0;

    puts("Enter -1 when done.");
    do
    {
        scanf("%lf", &num);
        ++c;
        sum += num;
        avg = sum / c;
        if ( num < low )
            low = num;
        if ( num > high )
            high = num;
    }   while ( num != -1 );

    printf("Values = %lu\n", c);
    printf("High = %f\n", high);
    printf("Low = %f\n", low);
    printf("Avg = %f\n", avg);

    return EX_OK;
}
```

The instructor will show the solutions after the class runs out of ideas.

Chapter 11

Functions

11.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. How do we make a function reusable at the source code level?
2. How to we make a function reusable at a practical level?
3. What is the difference between a declaration and a definition?

4. What is a function prototype? Why are prototypes necessary?
5. Where should prototypes be placed? Why?
6. How are arguments passed to C functions?
7. How do we simulate pass-by-reference in C?
8. When do promotions occur in argument passing?
9. What is a stub? How are stubs helpful?
10. What kinds of functions are suitable for recursion? Why?
11. What is the scope of a C variable?
12. Describe the four segments of an executable file and what each contains.
13. When should we use the `register` storage class?
14. What does the `volatile` modifier indicate?
15. What happens when a function is inlined?

11.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

  Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

11.3 Team Coding Example

1. Many mathematical functions, such as sin(), cos(), sqrt(), etc., use iteration to compute their results. Such iterative functions should be avoid where possible, and implemented as efficiently as possible.

   Write a C program that prints the powers of 2 from 2 ** 0 through 2 ** 63, using a function that computes base ** exponent for any non-negative integer exponent. A simple implementation will suffice for this exercise, but discuss how a more efficient approach might be implemented. No need to implement it, since the focus of this lab is C functions, not mathematical optimization.

   The instructor will show the solution after the class develops their own.

11.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

11.5 Code Review Example

For this week, we will review a contrived example.

- Find as many ways as possible to improve the program below:

```c
#include <stdio.h>
#include <sysexits.h>

int     factorial(int n);

int     main()

{
    int     c;

    for (c = 0; c <= 20; ++c)
        printf("%d! = %d\n", c, factorial(c));

    return EX_OK;
}


int     factorial(int n)

{
    int     f = 1, c;

    for (c = 1; c <= n; ++c)
        f *= c;

    return f;
}
```

The instructor will show the solutions after the class runs out of ideas.

Chapter 12

Programming with make

12.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. Is **make** a programming tool?
2. What is a target in a makefile?
3. What is a rule in a makefile?

4. How does make determine the order in which to execute rules?
5. What should be the default C compiler in most makefiles? Why?
6. What are the commands needed to build an executable called `prog1` from `prog1.c` and `math.c`?

```
cc -c prog1.c
cc -c math.c
cc -o prog1 prog1.o math.o
```

7. What Unix feature allows **make** to know when a rule should be executed?
8. How do **make** variables make life easier?
9. Show how to conditionally set make variables so that the default compile command is **cc -Wall -O -g file.c**.

```
CC      ?= cc
CFLAGS  ?= -Wall -O -g
```

10. What is a phony target? How are they handled by make?
11. Describe one pro and one con of makefile generators.

12.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

12.3 Team Coding Example

1. Write a C program and a Makefile to print the square roots of all integers from 0 to 10, inclusive, using a newly written `my_sqrt()` function. The main program and the `my_sqrt()` function should be in separate source files, and the `my_sqrt()` prototype should be in a header file.

   Just write a stub for the `my_sqrt()` function. The purpose here is to succeed in building a program from two source files and a header. The program does not need to be finished, it just needs to build cleanly. This is a good milestone for a typical workday, a lesson in divide and conquer with respect to the programming process.

   Use standard variables for C program builds, such as CC, CFLAGS, etc. Set them conditionally so that they can be overridden with **make** arguments or environment variables. Test using the default **cc**, and override the compiler using **make CC=clang** and **make CC=gcc**. Also test overriding the default compiler flags (e.g. `-Wall -O -g`) with **make CFLAGS='-Wall -O3'**.

   The instructor will show the solution after the class develops their own.

12.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

12.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 13

The C Preprocessor

13.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. Show a command that displays the C preprocessor output of my-prog.c.
2. Show how to define a constant called DEBUG_LEVEL with a value of 2, both within C code and from the command line.
3. Create a macro called `SIN_DEGREES()` that computes the sine of an angle given in degrees. The standard math library, libm, provides a `sin()` function that accepts an angle in radians.

4. Are macro definitions free-format?
5. What are two advantages of macros over functions? Are there exceptions?
6. Where does the preprocessor look for header files enclosed in angle brackets? In double quotes?
7. What language features belong in header files? In source files? Discuss.
8. How do we share a function definition across two different programs?
9. Show how to print the value of variable `sum` only if `DEBUG_LEVEL` is greater than 0, and print `latest_value` only if `DEBUG_LEVEL` is 3 or higher.
10. When should we use predefined platform macros such as `__APPLE__`, `__FreeBSD__`, or `__linux__`, for conditional compilation?
11. What is a shortcoming of the guard macros in header files, such as the following, and what else can we do to remedy the situation?

```
#ifndef _STDIO_H_
#define _STDIO_H_

// Content of stdio.h

#endif
```

13.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

13.3 Team Coding Example

1. Modify the Babylonian square root example in "Top Down Programming and Stubs" to use an `ABS()` macro in place of the `fabs()` function. Also add a caveman debug statement that prints `next_guess` and `next_guess - guess`, only if the macro DEBUG is defined. Compile and test the program with and without debugging output.

```
# Example output with debugging...

next_guess = 3.530392  diff = -1.569608
next_guess = 3.181469  diff = -0.348923
next_guess = 3.162336  diff = -0.019134
next_guess = 3.162278  diff = -0.000058
next_guess = 3.162278  diff = -0.000000
sqrt(10.0) = 3.162278
```

   The instructor will show the solution after the class develops their own.

13.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

13.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 14

Pointers

14.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. What is a pointer variable?
2. How does C differ from other high-level languages in its use of pointers?
3. How is computer memory structured from the perspective of a programmer?

4. What is a memory address?
5. Does ++ do the same thing to a pointer variable as it does to an integer variable?
6. What is the relationship between the size of a memory address and the size of an `int`?
7. Can we store memory addresses in integer variables?
8. In one variable definition, define a `long` variable called `c` and a pointer to a `long` called `p`.

```
    long    c, *p;
```

9. Draw a memory map showing address, variable name, and contents of the following variables. Assume a 64-bit computer.

```
    long        x = 5, y = 7, *p = &y;
    double      z = 1.0;
```

10. Show a `scanf()` call that places a value in `x` using the variable `p`.
11. Is accessing the object in `x` indirectly via `p` exactly the same as just referencing `x` directly?
12. Draw a possible memory map showing the contents of all variables in the following code at the end of the function call. Assume a 64-bit CPU.

```
void    func(int a, int *b)

{
    int     c = 6;

    a = 5;
    *b = 10;
}


int     main()

{
    int     x = 1, y = 4;

    func(x, &y);

    return EX_OK;
}
```

14.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

14.3 Team Coding Example

1. Write a C program that uses a function to read a list of values from `stdin`, returns EOF if successful or another value if there was faulty input. The low, high, and mean values should be returned via pointer arguments.

```
shell-prompt: ./stats
Enter numbers separated by whitespace.  Press Ctrl+d when done.
3
4
5
low = 3.000000, high = 5.000000, mean = 4.000000

shell-prompt: ./stats
Enter numbers separated by whitespace.  Press Ctrl+d when done.
asfdas
Input was incomplete, no stats to report.
```

   The instructor will show the solution after the class develops their own.

14.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

14.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 15

Arrays and Strings

15.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. When should we use arrays and similar in-memory aggregates? Why?
2. What is a scalar variable?
3. What are two problems with fixed size arrays?

4. What is a drawback to auto arrays?
5. What are two alternatives to auto arrays?
6. What subscripts are value for an array of size MAX_LIST_SIZE?
7. If the base address of list, an array of floats, is 1000, what is the address of list[20]?
8. What happens when we use a subscript that is out of range?
9. What is the relationship between arrays and loops?
10. What is the relationship between an array name and a pointer variable?
11. Write a loop that uses a pointer to populate list, an array of 10 ints, with data read from stdin.
12. Are arrays passed by reference in C?
13. Write a C function that returns the square root of any integer from 0 to 4, using a lookup table.
14. Show 4 different function prototypes for a function `strlen()` which returns the length of a character string.
15. Do formal argument array variables need to indicate the dimensions of the array? Why or why not?
16. How does increased memory use affect memory performance?
17. What does row-major mean and why is it important?

15.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

15.3 Team Coding Example

1. Write a C program that implements a bubble sort on a list of integers from the standard input. The maximum list size is 1 billion elements. The minimum and maximum values possible are 1 and 50,000. Justify all chosen data types.

   Note It is OK if this program is not completely finished and polished. The point of the exercise is to have a discussion about programming with arrays and functions. When it seems the class has gotten all it can from that discussion, you can move onto the next topic.

   The program should include a `read_list()` function, a `sort_list()` function, a `swap()` function, and a `print_list()` function.

   Code and test `read_list()` and `print_list()` before beginning on `bubble_sort()`. Use stubs to eliminate compiler errors before writing the body of each function.

   The `main()` function should be first. Macros, `typedefs`, and prototypes for all functions should be placed in a header file. Source files should contain only function definitions and `#includes`.

   Optional: If time permits, place the `bubble_sort()` and `swap()` functions in separate files, and use a makefile to build the program.

   The input stream contains the list size as the first value, followed by each element of the list. The elements may be separated by spaces, tabs, or newlines.

   The instructor will show the solution after the class develops their own.

15.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

15.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 16

Dynamic Memory Allocation

16.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. How does dynamic memory allocation improve programs?
2. Is address space free?
3. What is a garbage collector?

4. Which is better, garbage collection or manually freeing memory?
5. Show a C code segment that allocates an array of `vector_length` doubles.
6. What are two down sides of linear linked lists?
7. What is the up side and the down side of allocating an array all at once, as opposed to a linear linked list?
8. What are two major advantages of pointer arrays over fixed 2D arrays?
9. What are `argc` and **argv** for the following command?

```
shell-prompt: ./myprog input.txt output.txt 3
```

16.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

16.3 Team Coding Example

1. Modify the bubble sort program from the previous chapter so that the list is dynamically allocated. The `read_list()` function should return the size of the list, and a pointer to the allocated array should be sent back the caller via a pointer argument. The input stream will contain the size of the list as the first item, followed by the list elements, all separated by whitespace.

   The instructor will show the solution after the class develops their own.

16.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

16.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 17

Function Pointers

Skip this chapter for now.

Chapter 18

Structures and Unions

18.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. Are structures the same as the classes we use in object-oriented languages?
2. Typedef a structure containing a person's first name, last name, age, and height in cm, hair color, and eye color. Justify the data types for each field.
3. Define a pointer to a `person_t` structure and show how to set `hair_color` to `BLACK` using both a `.` and a `->` operator.

4. Why do we usually pass structure pointers to functions rather than passing structures by value?
5. What is encapsulation with regard to OOP? Can it be done in C?
6. Write a "constructor" function for the `person_t` structure that takes no arguments other than the `person_t` object and initializes the structure to "blank" values.

```
void    person_init(person_t *person)

{
    person->first_name[0] = '\0';
    person->last_name[0] = '\0';
    person->age = 0;
    person->height = 0;
    person->hair_color = HAIR_UNKNOWN;
    person->eye_color = EYES_UNKNOWN;
}
```

7. Show how to allocate an array of `num_people` `person_t` structure pointers, set each of them to the address of a newly allocated structure, and initialize the new structure using the constructor from the previous problem (which is probably a waste of time, but an easy practice exercise).

```
    person_t    *people;
    size_t      num_people, c;

    if ( (people = malloc(num_people * sizeof(person_t *))) == NULL )
    {
        // Error out
    }

    for (c = 0; c < num_people; ++c)
    {
        if ( (people[c] = malloc(1 * sizeof(person_t))) == NUL )
        {
            // Error out
        }
        person_init(people[c]);  // No &, people[c] is a pointer
    }
```

8. How does a union differ from a structure?

18.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.

- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

18.3 Team Coding Example

1. Write a C program that reads an inventory list from `stdin` and displays the items in human-readable format. This is similar to the process of reading directories, as done by the **ls** command, which is covered in a later chapter.

```
Item name               Price  Stock
-----------------------------------
Shampoo               $  4.99     10
Swabs                 $  1.99     12
TP                    $  6.99      3
```

The program should use a structure that contains an item name, the price in pennies, and the inventor count. A sample input file is shown below.

```
Shampoo     499     10
Swabs       199     12
TP          699     3
```

Treat the structure like a class, placing the structure definition and other class-related items in a header file, and the member functions in a separate source file. The class should have two member functions that take a pointer to an object as the first argument. The main program below demonstrates the interface.

```
#include <stdio.h>
#include <sysexits.h>
#include "item.h"

int     main()

{
    item_t  item;

    while ( (item_read(&item, stdin)) != EOF )
        item_print(&item, stdout);

    return EX_OK;
}
```

Note Though reading a string with `fscanf()` is not safe, it is acceptable for this exercise in the interest of saving class time. A real program should use a string input function that prevents buffer overflows.

Write a simple makefile to build the program from a source file containing main and the source file containing the member functions.

The instructor will show the solution after the class develops their own.

18.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

18.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 19

Debugging

19.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. What is the down side of becoming dependent on debuggers?
2. What should be our first goal in debugging a program. Why?
3. Why should debug output be sent to `stderr` rather than `stdout`?

4. What is a break point and how is it used?
5. How do we get the most out of a backtrace?
6. What is a core file?

19.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

19.3 Team Coding Example

1. The program shown below produces incorrect output and crashes. Use caveman debugging to diagnose the incorrect output and use **lldb** on unixdev1 to pinpoint the crash location.

   You can copy and paste the code from this document, or to save time, run the following:

```
curl -O https://acadix.biz/C-Unix/debugging-team.c
```

```
#include <stdio.h>
#include <sysexits.h>
#include <stdlib.h>

typedef double  real_t;

ssize_t read_list(real_t **list_ptr)

{
    real_t  *list = NULL;
    ssize_t list_size, c;

    scanf("%zd", &list_size);
    for (c = 0; scanf("%lf", &list[c]) == 1; ++c)
        ;
```

```
    return list_size;
}


void    swap(real_t n1, real_t n2)

{
    real_t  temp;

    temp = n1;
    n1 = n2;
    n2 = temp;
}


void    sort_list(real_t list[], ssize_t list_size)

{
    ssize_t start, c, low;

    for (start = 0; start < list_size - 1; ++start)
    {
        // Find smallest
        for (c = start + 1; c < list_size; ++c)
            if ( list[c] < list[low] )
                low = c;

        swap(list[low], list[start]);
    }
}


void    print_list(const real_t list[], ssize_t list_size)

{
    ssize_t c;

    for (c = 0; c < list_size; ++c)
        printf("%f\n", list[c]);
}


int     main()

{
    real_t  *list = NULL;
    ssize_t list_size;

    if ( (list_size = read_list(&list)) > 0 )
    {
        sort_list(list, list_size);
        print_list(list, list_size);
    }

    return EX_OK;
}
```

The instructor will show the solution after the class develops their own.

19.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

19.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Part III

Unix Library Functions and Their Use

Chapter 20

Building Object Code Libraries

20.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. How hard is creating libraries? What does this imply?
2. Which is better, static libraries, or dynamic libraries?

20.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

20.3 Team Coding Example

1. Create a makefile that builds a static library called `libmath.a` from two source files, `fastfact.c` and `power.c`.

   You can download the source code as follows:

```
curl -O https://acadix.biz/C-Unix/fastfact.c
curl -O https://acadix.biz/C-Unix/power.c
```

```
#include <stdint.h>

uint64_t   fastfact(unsigned int n)

{
    // This is initialized at compile time, so the values are already
    // in the array before the function is called.  The only costs to
    // this function are function call overhead and a simple address
    // calculation table + n * sizeof(uint64_t).  Much faster
    // than a loop or recursion.
    static uint64_t table[] = { 1ul, 1ul, 2ul, 6ul, 24ul, 120ul, 720ul,
        5040ul, 40320ul, 362880ul, 3628800ul, 39916800ul, 479001600ul,
        6227020800ul, 87178291200ul, 1307674368000ul, 20922789888000ul,
        355687428096000ul, 6402373705728000ul, 121645100408832000ul };

    return table[n];
}
```

```
double  fastpower(double base, unsigned exponent)

{
    if ( exponent == 0 )
        return 1.0;
    else
    {
        double  half = power_fast(base, exponent / 2);

        if ( (exponent & 0x1) == 0 ) // 1 clock cycle check for odd integer
            return half * half;
        else
            return half * half * base;
    }
}
```

The instructor will show the solution after the class develops their own.

20.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

20.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 21

Files and File Streams

21.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. Why do we use FILE streams with their buffering mechanisms. Wouldn't it be simpler to access files and devices directly?
2. What happens when a character is written to a stream buffer that is not yet full?
3. What happens when a character is written to a stream buffer that is full?

A low level `write()` is issued to transfer the entire buffer contents to the file or I/O device. The buffer pointer is then set back to the beginning of the buffer.

4. Why is writing to a buffer faster than writing to a disk?
5. Why must we also check for the success of I/O operations?
6. What happens if we don't close a file as soon as possible?
7. Why do we need to know about `feof()` and `ferror()`?
8. What is a race condition regarding temporary files? How do we avoid it?
9. What is a filter program?
10. What FILE stream function ensures that output is written to the disk or I/O device before the program proceeds?
11. How do we handle whole-file manipulations in C, such as creating directories, renaming files, etc?

21.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

21.3 Team Coding Example

1. Write a C program that implements a rudimentary **grep** command that functions as a filter. It should take one string argument on the command line (argv[1]), followed by zero or more filenames (argv[2] through argv[argc-1]), and display all lines in input files that contain the string. If no filename arguments are provided, it should read from `stdin`.

```
./grep string file [file ...]
```

Use the `strstr()` function to check each line of input for the substring.

The program should return EX_OK if a match was found in any file, or a non-zero return value if no matches were found.

It should display the filename before each matching line, if possible (i.e. if not reading from `stdin`).

```
shell-prompt: ./grep strstr grep.c
grep.c:      if ( strstr(line, string) != NULL )
```

The instructor will show the solution after the class develops their own.

21.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

21.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 22

String Functions

22.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. When should we use `strcpy()`?
2. What is the best way to handle a string that is too long for the target character array?
3. Can we copy strings using the '=' operator?

4. How does `strdup()` work and what should always accompany it?
5. When should we use `strlen()`?

22.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

> **Note** It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

22.3 Team Coding Example

No team coding exercise for this chapter.

22.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

22.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 23

Odds and Ends

23.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. What is a portable way to find out what math functions are available on a Unix system?
2. Show how to convert the contents of a string variable called `numeric_input` to a long long, assuming the text in the string is in hexadecimal format.
3. Show how to convert a `size_t` variable to a string containing its value in octal.

4. Write a loop that prints 100 random numbers. It should print a different sequence each time the program is run.
5. What is a polymorphic function? Does C support this?

23.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

> **Note** It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

23.3 Team Coding Example

No team coding for this chapter.

23.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

23.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 24

Working with the Unix Filesystem

24.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. Show how to check if members of the group have write permission on the file `results.txt`.
2. What is the difference between `stat()` and `fstat()`?
3. Show how to add read and execute permissions for "other" on the file "results.txt".

24.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

24.3 Team Coding Example

No team coding for this chapter.

24.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

24.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 25

Low-level I/O

25.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. When should low-level I/O be used?
2. How is low-level I/O related to FILE stream I/O?
3. How is the open mode specified when calling `open()`?
4. What does `read()` return when end-of-file is reached? If we don't know this, how can we find out?

25.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

25.3 Team Coding Example

1. Write a C program that implements a very basic **cp** command using low-level I/O. It should require exactly two command-line arguments, a source and a target, and use `open()`, `read()`, `write()`, and `close()`.

   The instructor will show the solution after the class develops their own.

25.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

25.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 26

Controlling I/O Device Drivers

26.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

No questions for this chapter.

26.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

26.3 Team Coding Example

No team coding for this chapter.

26.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

26.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 27

Unix Processes

27.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. What is a process?
2. When should we use `system()` and when should we use `fork()` or `posix_spawn()`?
3. How do we know if the `execve()` function failed?

27.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

27.3 Team Coding Example

1. Write a C program that forks a child process, which runs the command "ls /usr/include". The parent then waits specifically for the child process to exit, as indicated by the WEXITED passwd to `waitpid()`. The parent should print the exit status of the child process.

   Discuss which `exec` function is best for this task, and run **man waitpid** for details on using the `waitpid()` function.

   The instructor will show the solution after the class develops their own.

27.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

27.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 28

Interprocess Communication (IPC)

28.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. Briefly describe five ways that processes can communicate with each other.

28.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

28.3 Team Coding Example

1. Write a C program that reads data from the standard input and pipes it through a separate process running the **more** command. (Use input redirection when running it to avoid confusion.)

   The instructor will show the solution after the class develops their own.

28.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

28.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 29

Threads

29.1 Practice

Instructions

- These questions are meant to be answered by the students during lab sessions and are counted as part of the participation grade. The instructor will pose the question to the class and one or more students should attempt to answer. ANSWERING 100% CORRECTLY IS NOT NECESSARY FOR PARTICIPATION POINTS. A reasonable attempt with roughly the right idea is sufficient. Hail Mary's and wise cracks don't count.
- The class should have a brief discussion about each question, usually not more than a minute or so, but it depends on the question. Don't just move on after one student provides a reasonable answer. After the class comes to some consensus about the right answer, the instructor will share the official solution on the board, as well as augment the discussion with their own perspective.
- The class should feel free to make up more examples if needed to clarify the concept. These questions are only an outline of topics to practice.
- Unlike the practice problems in the lecture outline, the key for these questions is not available to students. You must attend lab sessions in order to find out the answers to these questions.
- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest versions of the lecture outline and this document. They are updated frequently.
- Read the relevant sections of this document and corresponding materials BEFORE COMING TO LAB.
- State the answer in your own words. Do not just read from the lecture notes or the book. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Brevity and clarity are the most important aspects of good communication. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Verify your results by trying out Unix commands, testing code, etc. Double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.

---

1. Discuss at least four ways in which threads and heavyweight processes differ.
2. Briefly describe two standardized threads implementations.

29.2 Team Coding Instructions

Below is a guide for participating in an interactive group coding session, as will be done during each lab/discussion session, once we get into C programming.

- Students suggest a name for the source file. Discuss options and come up with the best filename possible.

Note It is a common habit to use underscores '_' to separate words in a filename. The habit was probably a carryover from their use in variable names. Note, however, that filenames can use a dash '-', which is easier to type than an underscore, since it does not require using the shift key. It is also easier to see in some situations since underscores occasionally get squashed against the character below, such as a 'T' with certain fonts.

- Write a small section of code first, then define the variables that it requires. Defining variables before writing code is a common tendency among beginners, but it's absurd. We don't know what variables we'll need until we write the code. All text editors allow us to move the cursor up.
- Students to suggest the next statement to write, including the best possible variable names within the statement. The instructor should ask for multiple suggestions before typing anything in.
- Discuss what is the best data type for each variable. Discuss multiple options and why one type is preferable. Integers vs floating point, char vs short vs int vs long vs int64_t vs unsigned. For floating point, double vs float.
- Discuss what should be done next. Write more code? Test the code we have so far? Do we need to add temporary debug statements to test the partially completed code? No more than a few lines of new code should be written before testing the program again.

29.3 Team Coding Example

1. Write a C program using OpenMP to compute averages of values in 4 different files, called `input1.txt`, `input2.txt`, `input3.txt`, and `input4.txt`.

   Hint: Use `omp_get_thread_num()` and `snprintf()` to construct the filename processed by each thread.

   Run the program several times and check the order of the outputs.

   Do you think this is an effective use of threads? Why or why not?

   The instructor will show the solution after the class develops their own.

29.4 Code Review Instructions

If all students in the course have turned in the most recent programming project, conduct a group code review to help everyone learn how to improve their own and other peoples' code.

- Review the lead instructor's solution to the previous programming project.
- Review the lab/discussion instructor's solution to the previous programming project.
- Review the solutions to the previous programming project from one or more students. All students should have their code reviewed at least once during the semester. Student code may be anonymous.

29.5 Code Review Example

For this week, review some past code written by instructors and students. You might also choose to review some random code taken from a web forum such as Stack Exchange or Geeks for Geeks.

Chapter 30

Appendix

30.1 Cygwin: Try This First

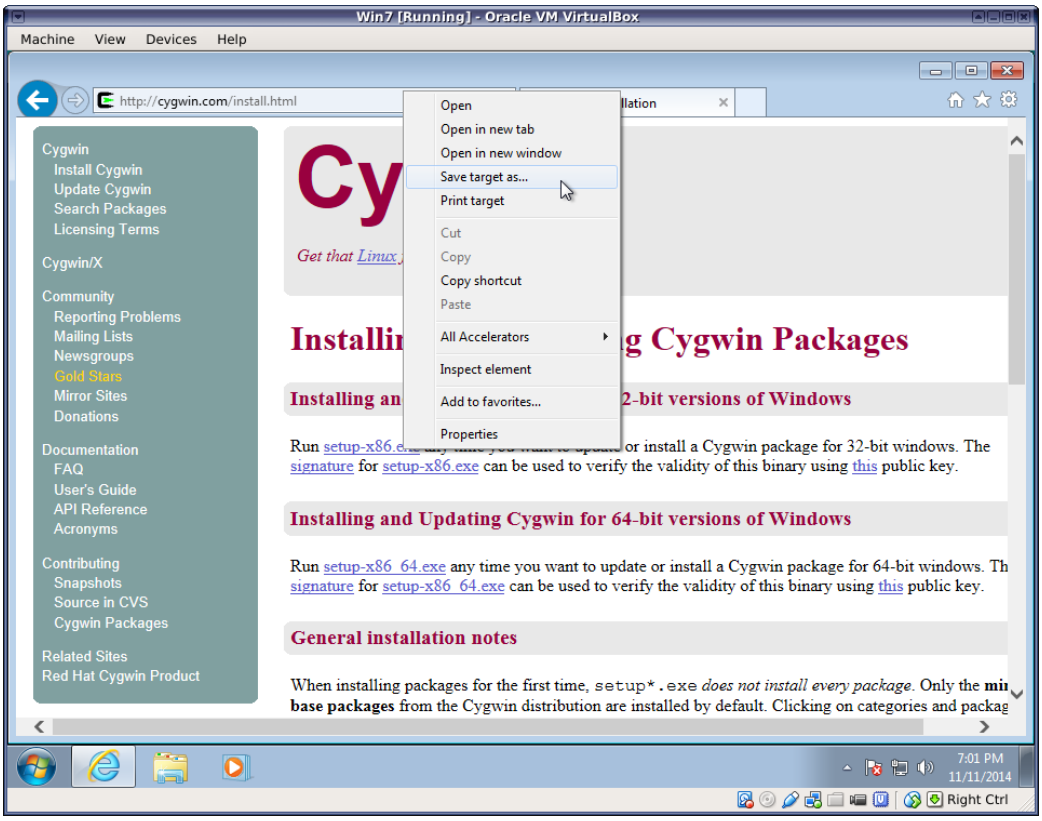
Cygwin is a free collection of Unix software, including many system tools from Linux and other Unix-compatible systems, ported to Windows. It can be installed on any typical Windows machine in about 10 minutes and allows users to experience a Unix user interface as well as run many popular Unix programs right on the Windows desktop.

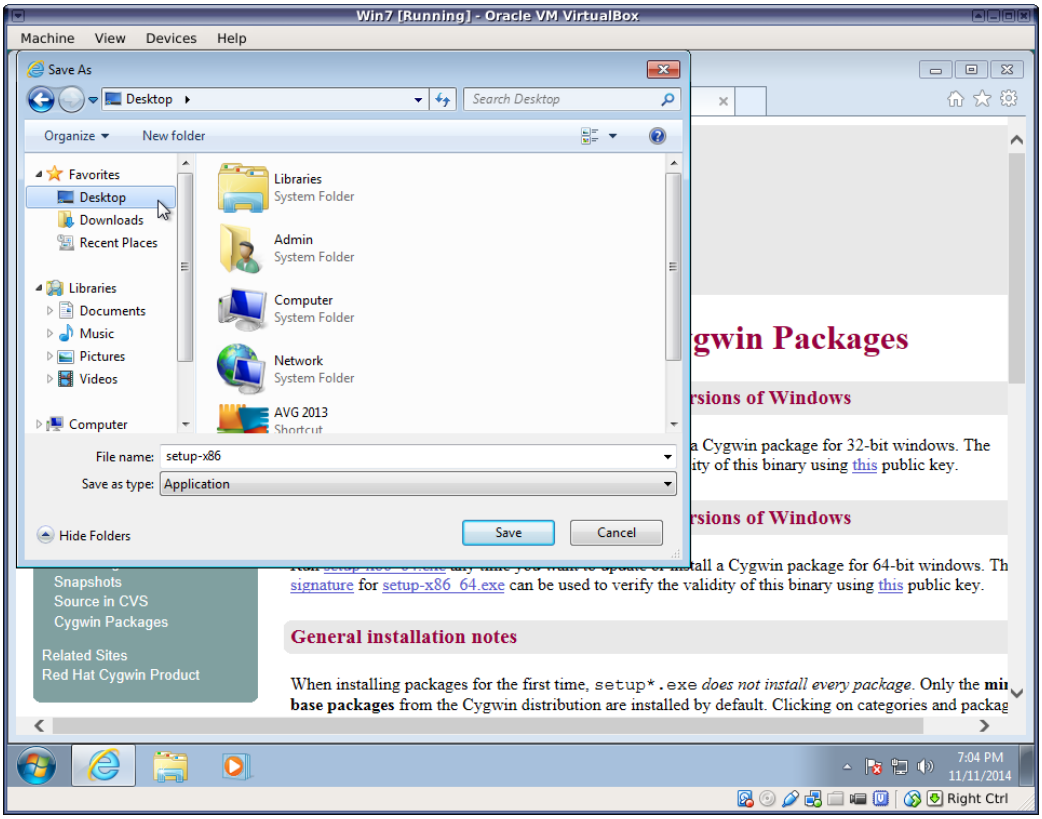
Cygwin is a *compatibility layer*, another layer of software on top of Windows that translates the Unix API to the Windows API. As such, performance is not as good as a native Unix system on the same hardware, but it's more than adequate for many purposes. Cygwin may not be ideal for heavy-duty data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

Cygwin 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.

Installing Cygwin is quick and easy:

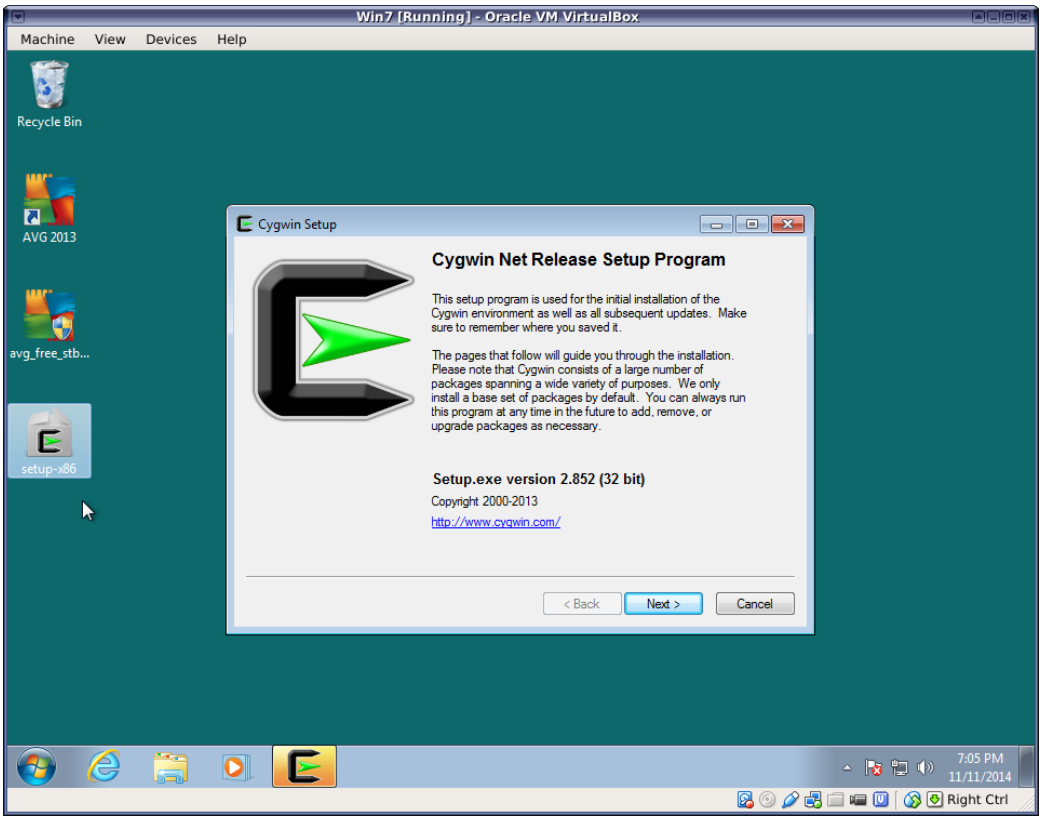
1. Download **setup-x86_64.exe** from https://www.cygwin.com and save a copy on your desktop or some other convenient location. You will need this program to install additional packages in the future.



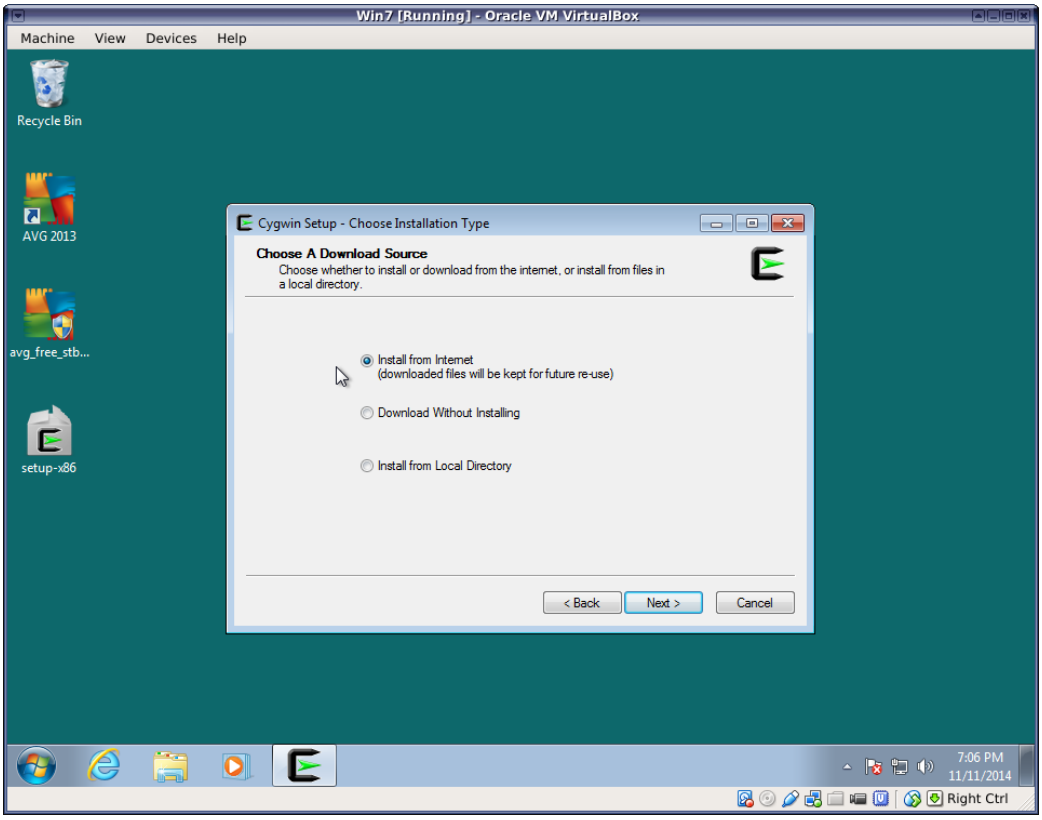


2. Run **setup-x86_64.exe** and follow the instructions on the screen.

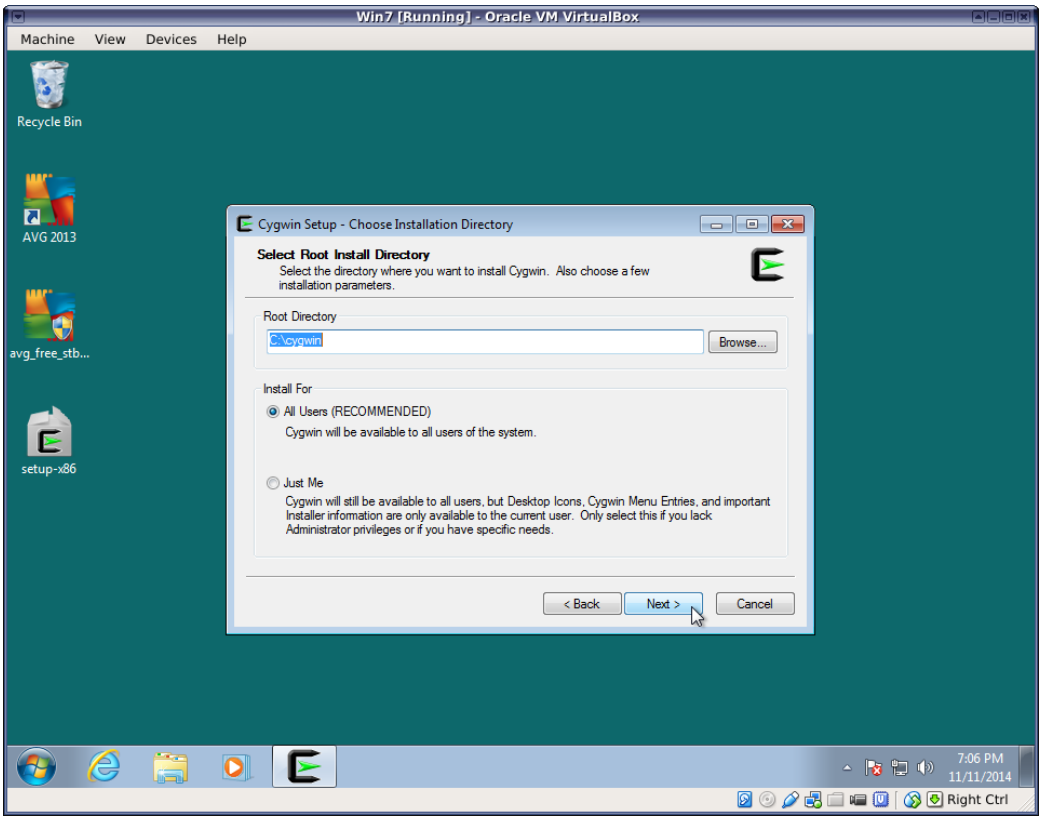
   Unless you know what you're doing, accept the default answers to most questions. Some exceptions are noted below.



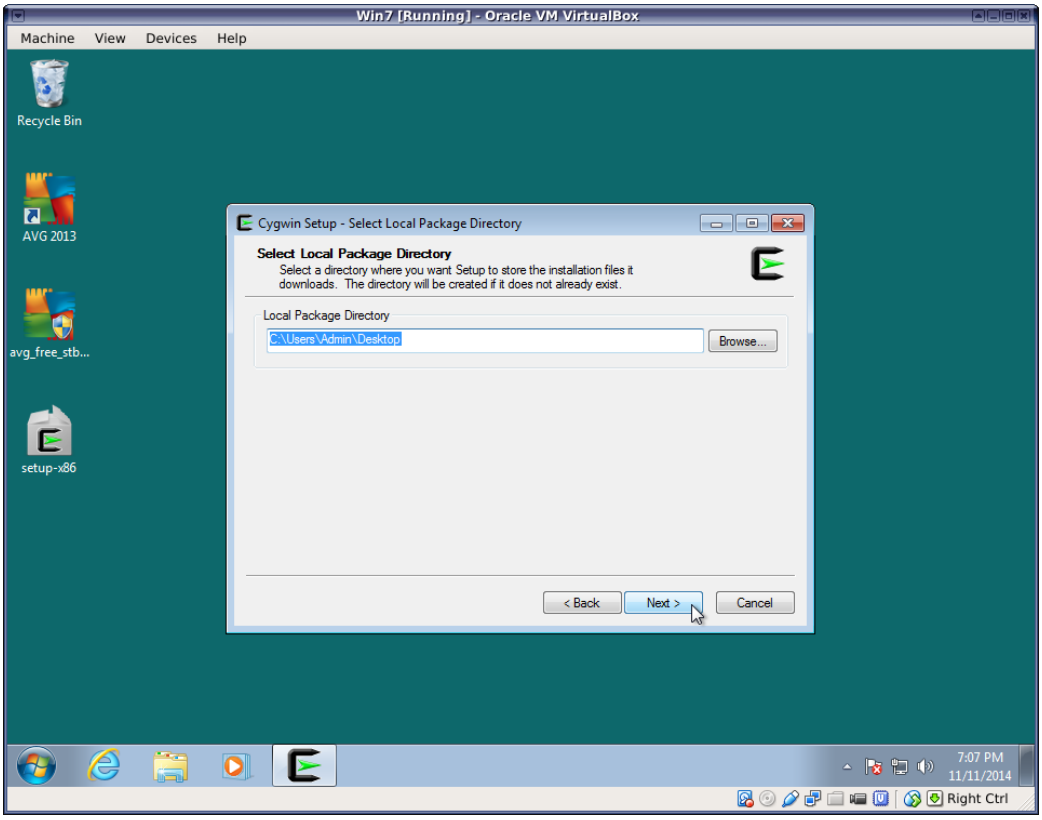
3. Unless you know what you're doing, simply choose "Install from Internet".



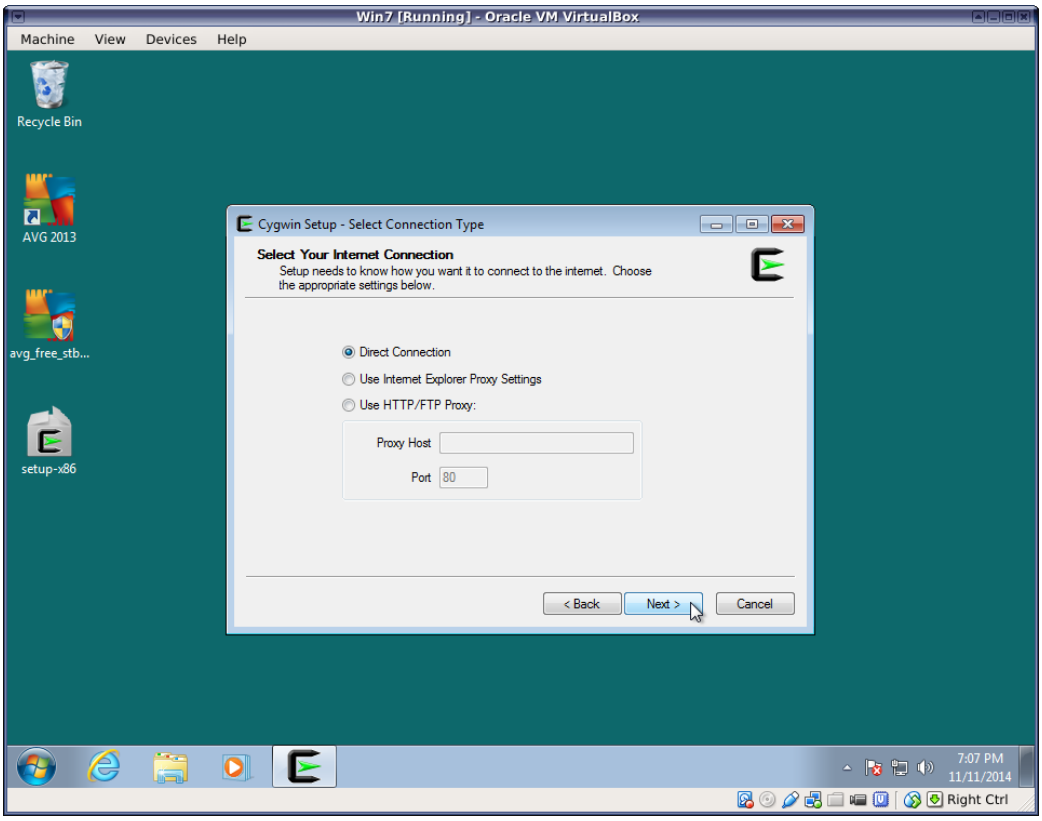
4. Select where you want to install the Cygwin files and whether to install for all users of this Windows machine.



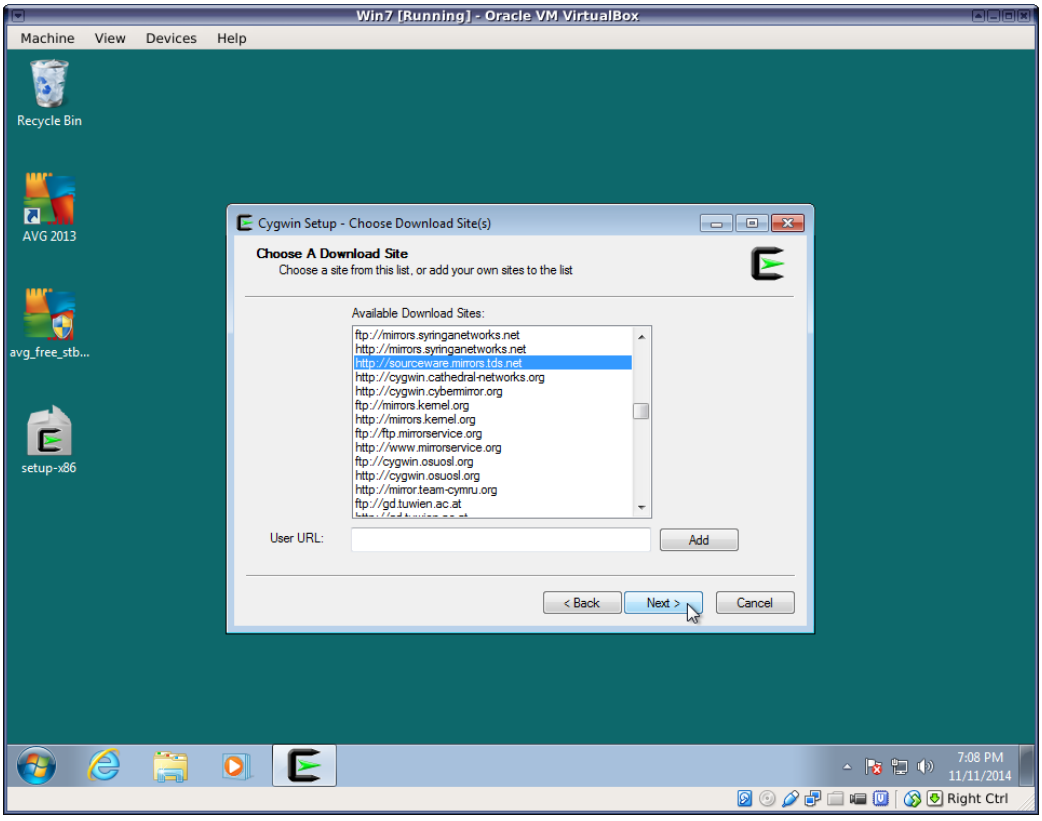
5. Select where to save downloaded packages. Again, the default location should work for most users.



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.



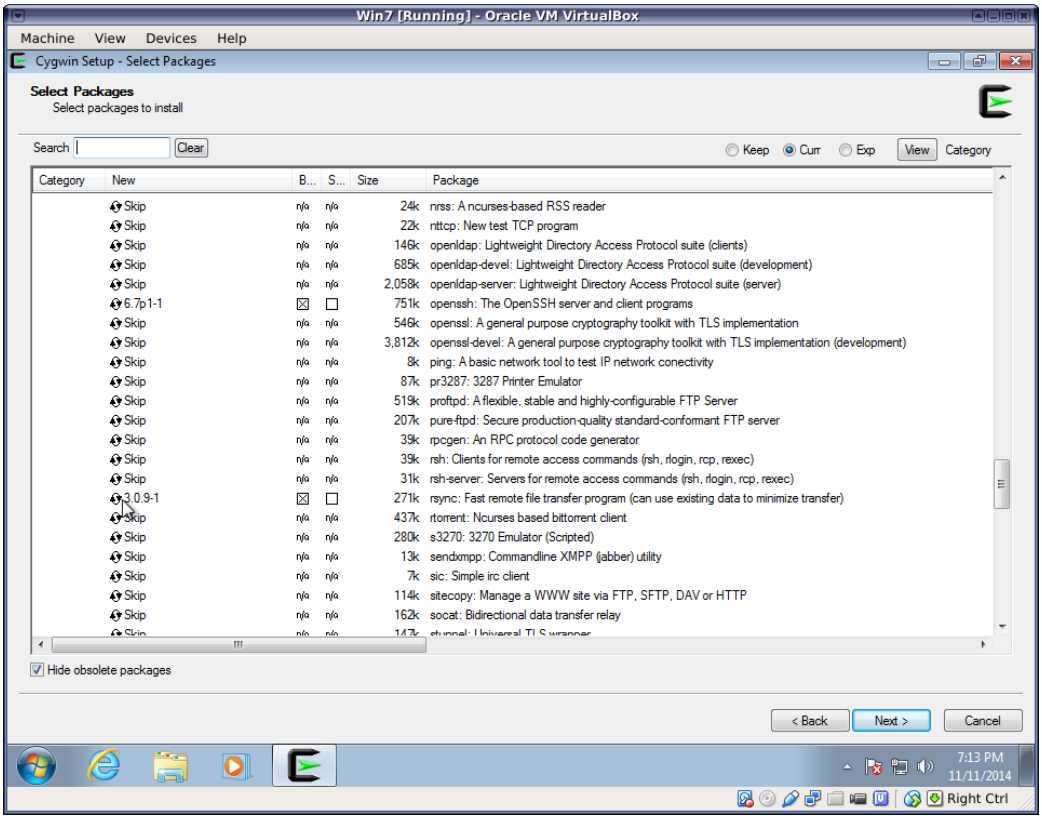
8. When you reach the package selection screen, select at least the following packages in addition to the basic installation:

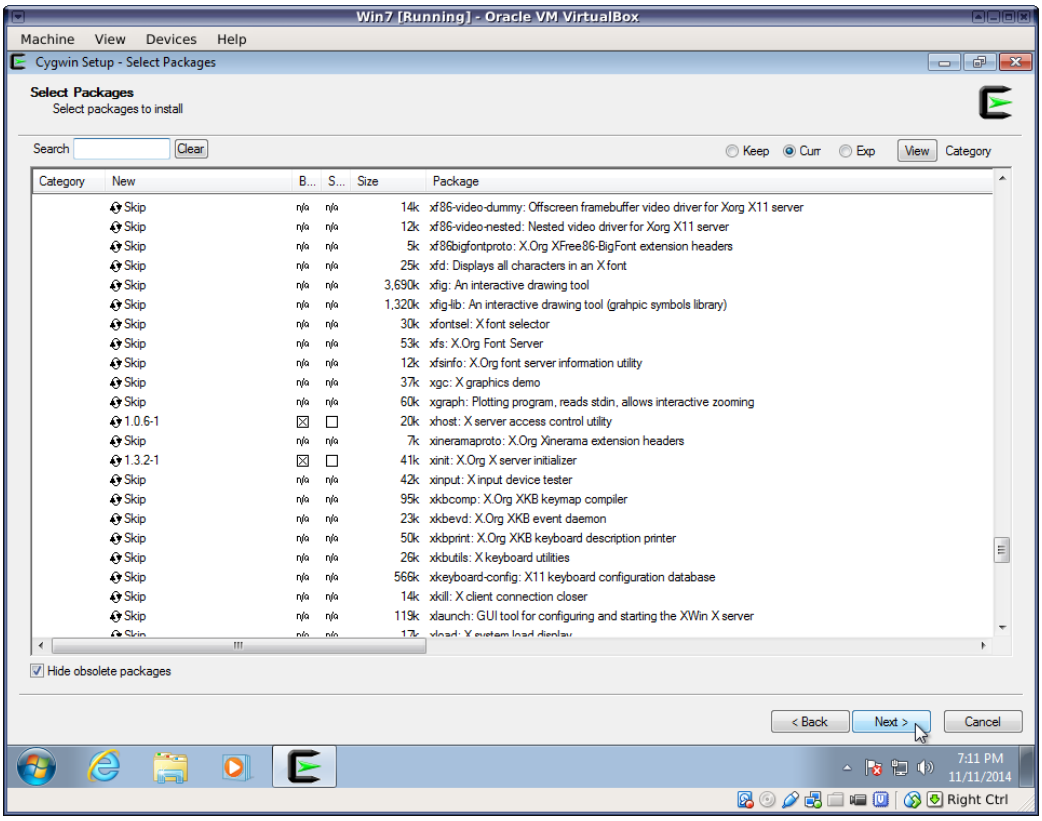
- net/openssh
- net/rsync
- x11/xhost
- x11/xinit

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.



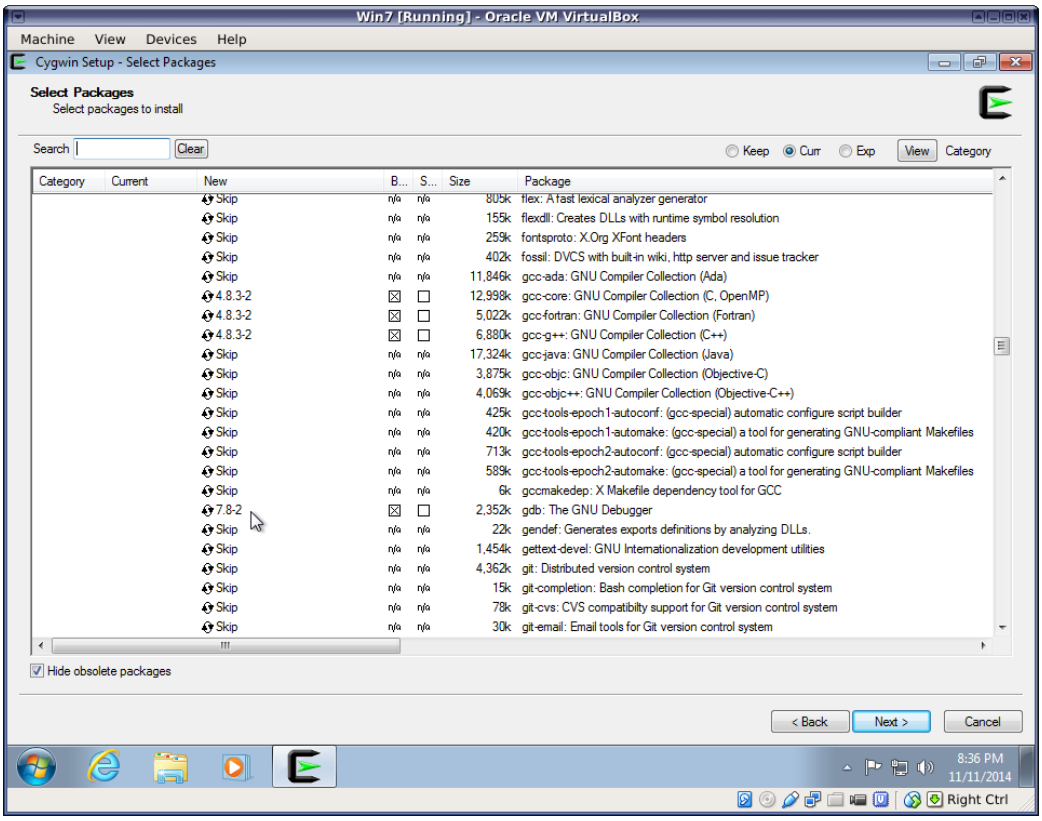


Cygwin can also enable you to do Unix program development on your Windows machine. There are many packages providing Unix development tools such as compilers and editors, as well as libraries. The following is a small sample of common development packages:

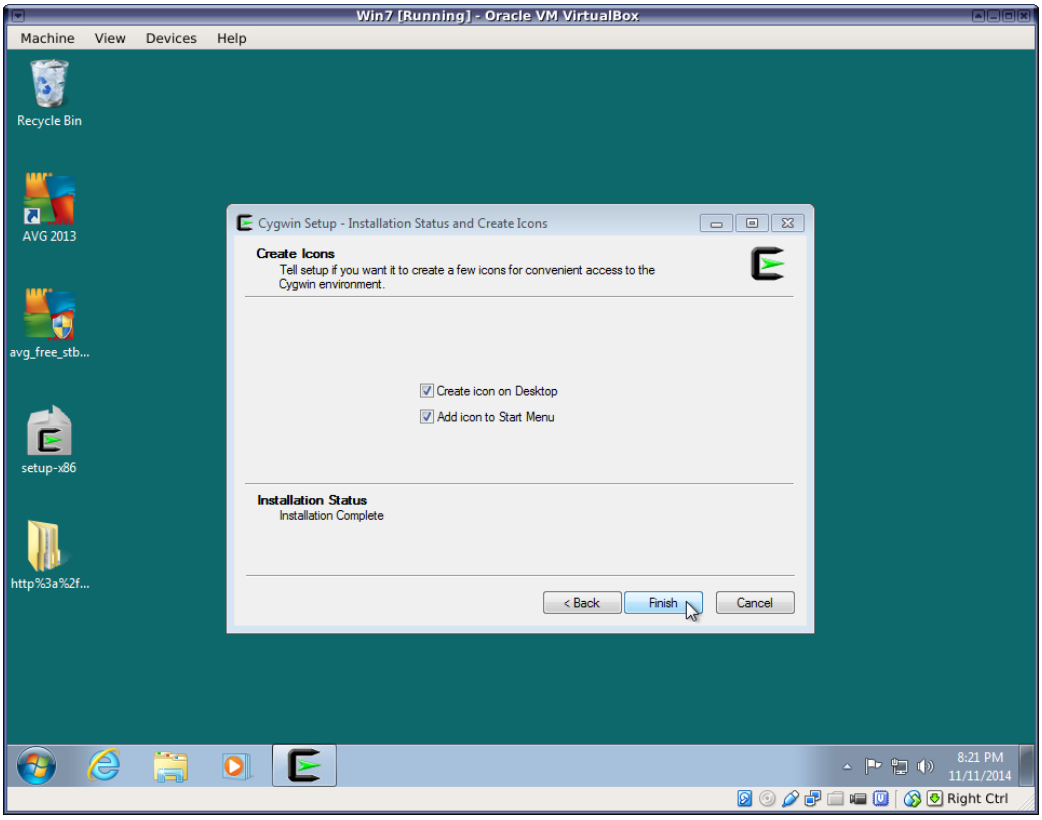
Note

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)

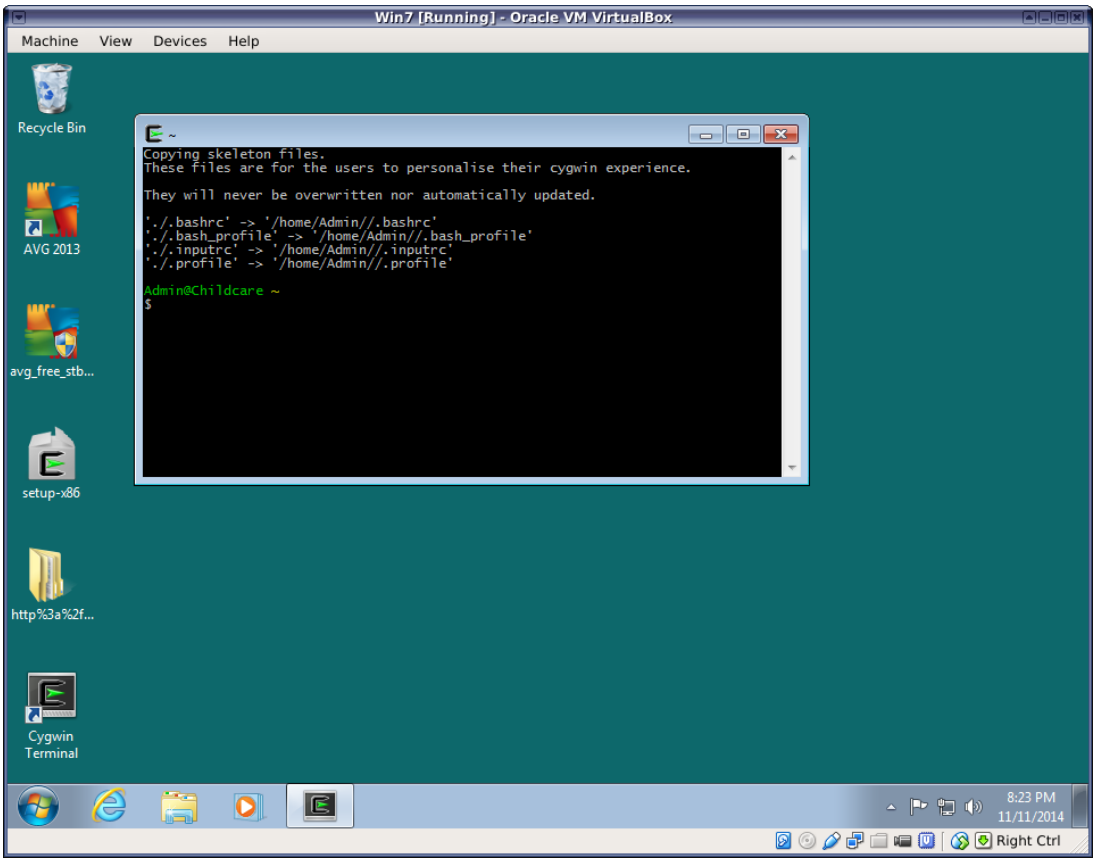


9. Most users will want to accept the default action of adding an icon to their desktop and to the Windows Start menu.

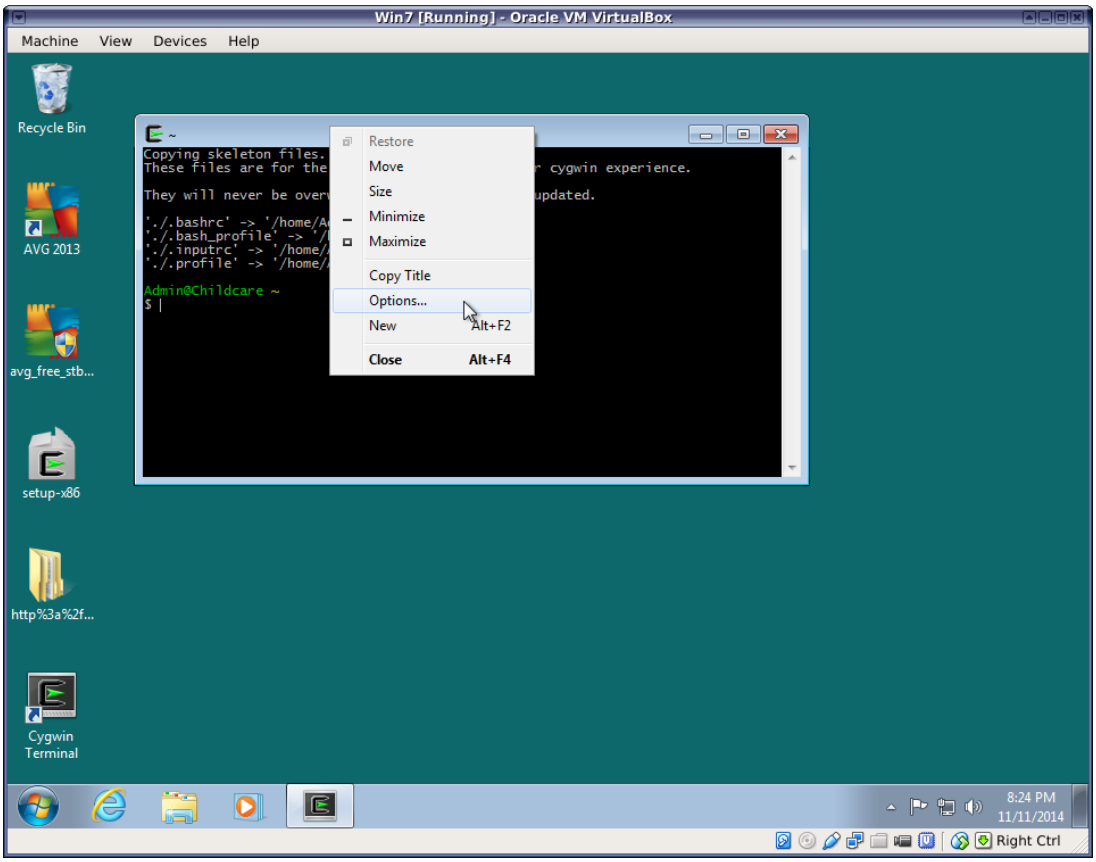


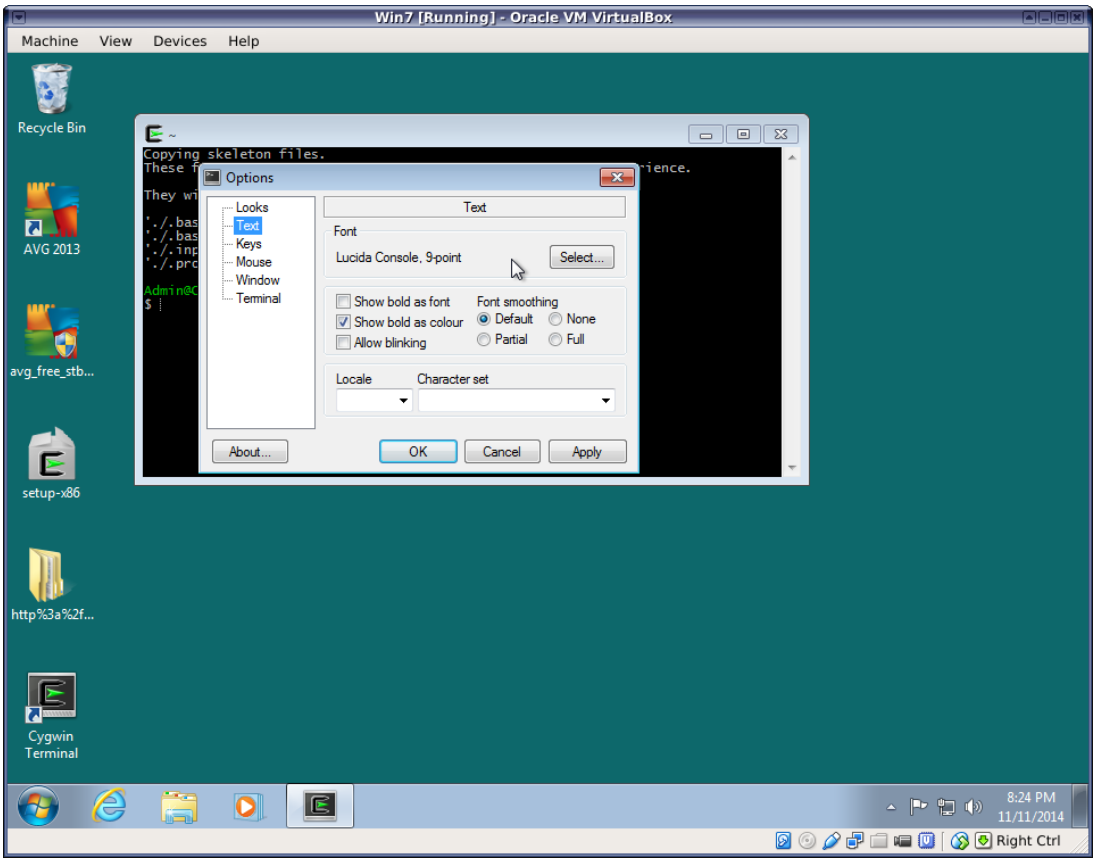
When the installation is complete, you will find Cygwin and Cygwin/X folders in your Windows program menu.

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



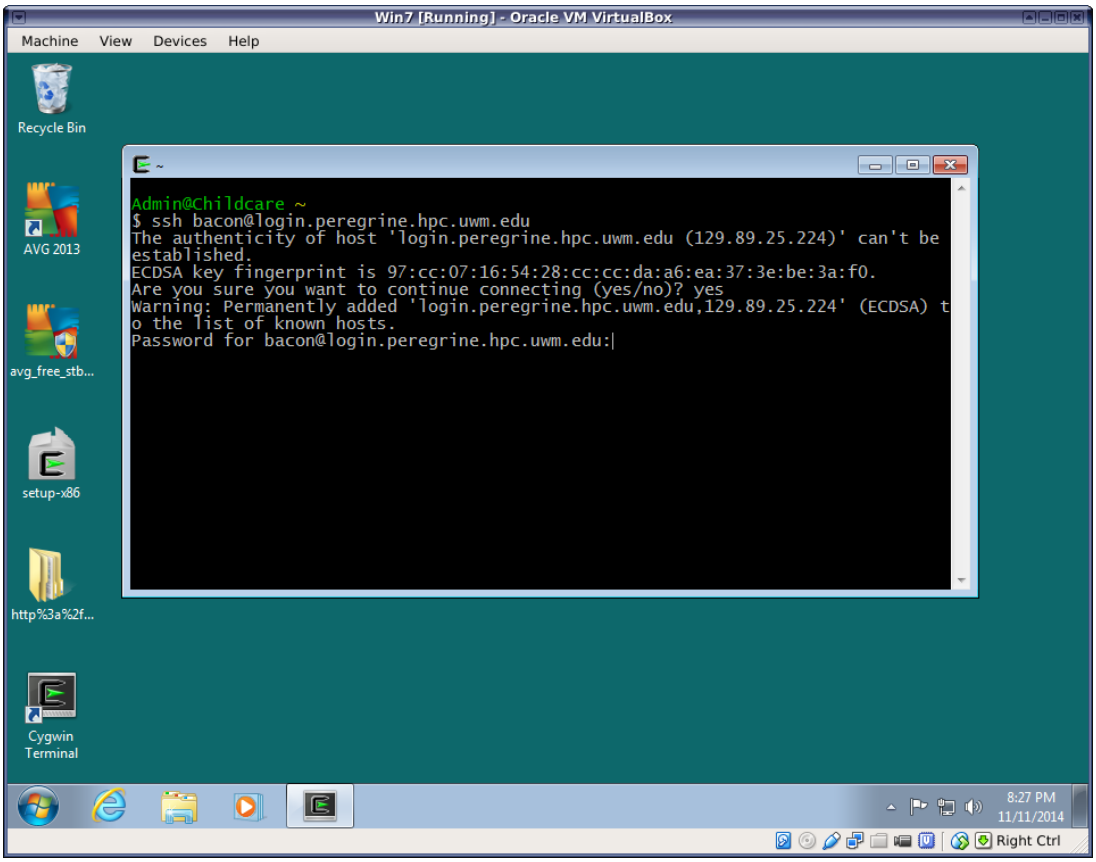
If you'd like to change the font size or colors of the Cygwin terminal emulator, just right-click on the title bar of the window:





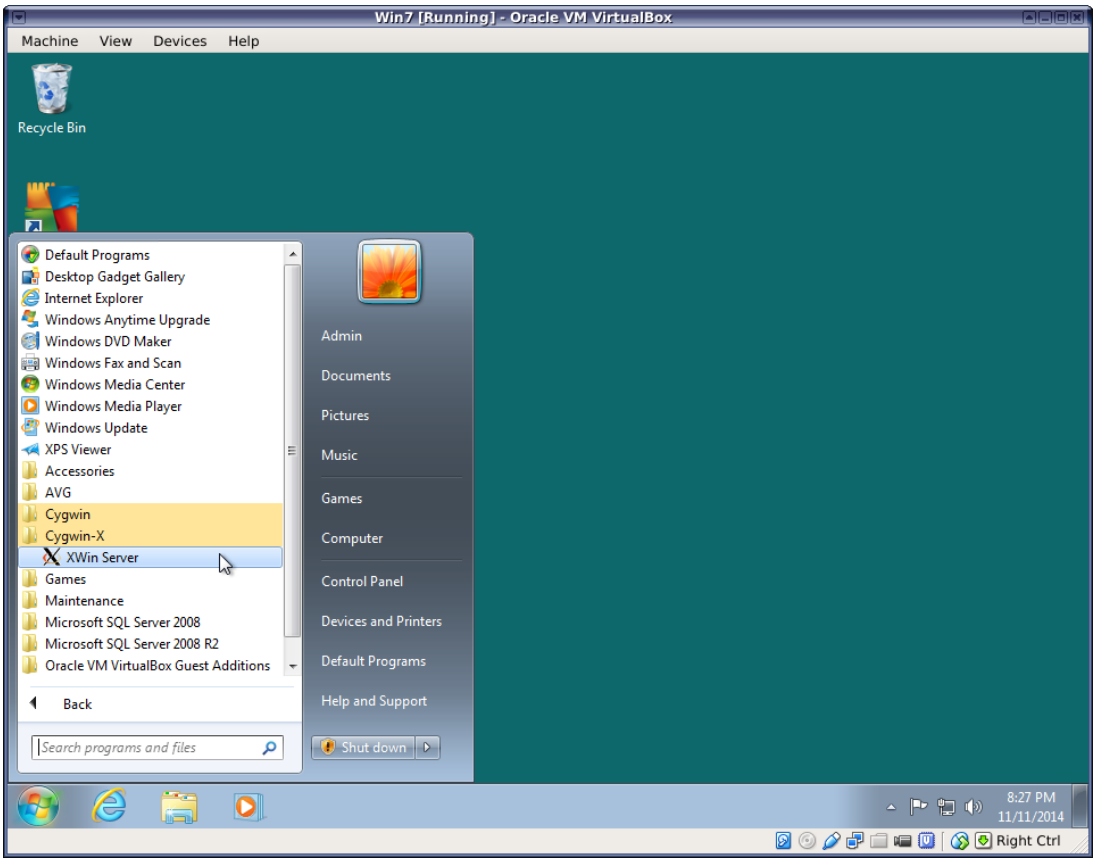
Within the Cygwin terminal window, you are now running a "bash" Unix shell and can run most common Unix commands such as "ls", "pwd", etc.

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.

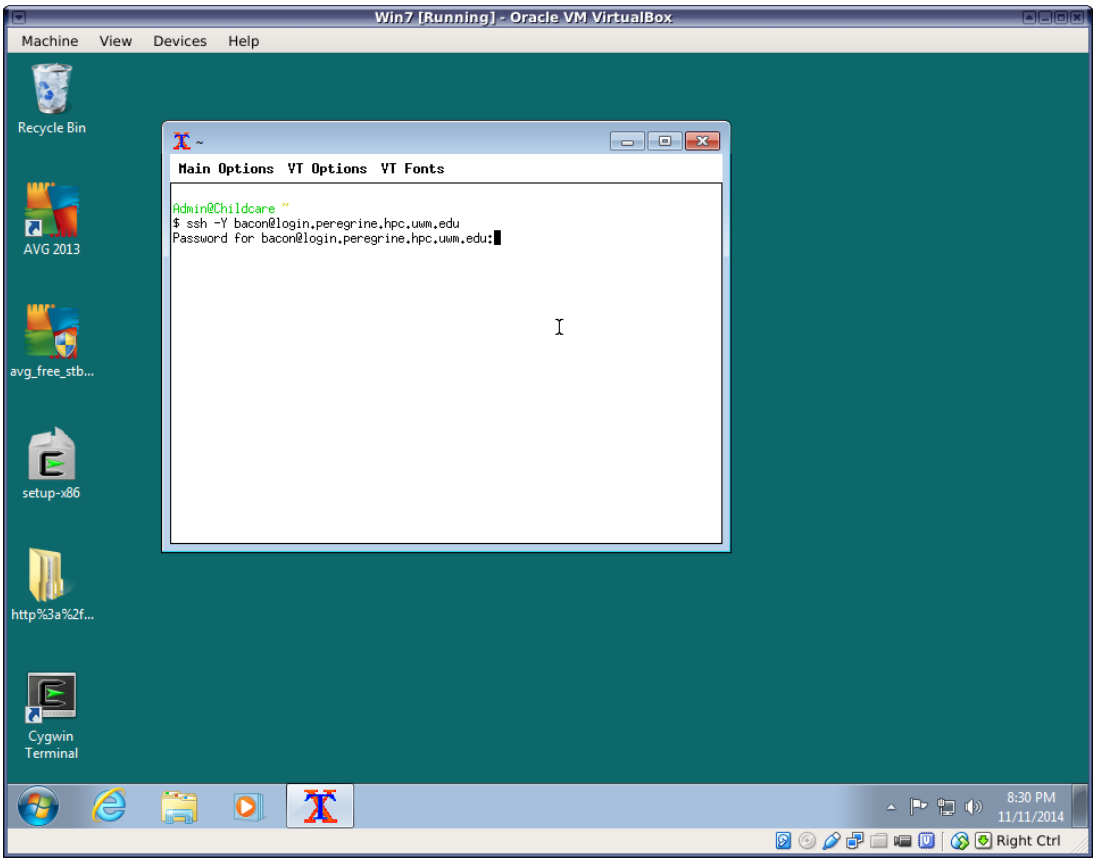
If you want to run Unix graphical applications, either on your Windows machine or on a remote Unix system, run the Cygwin/X application:



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.

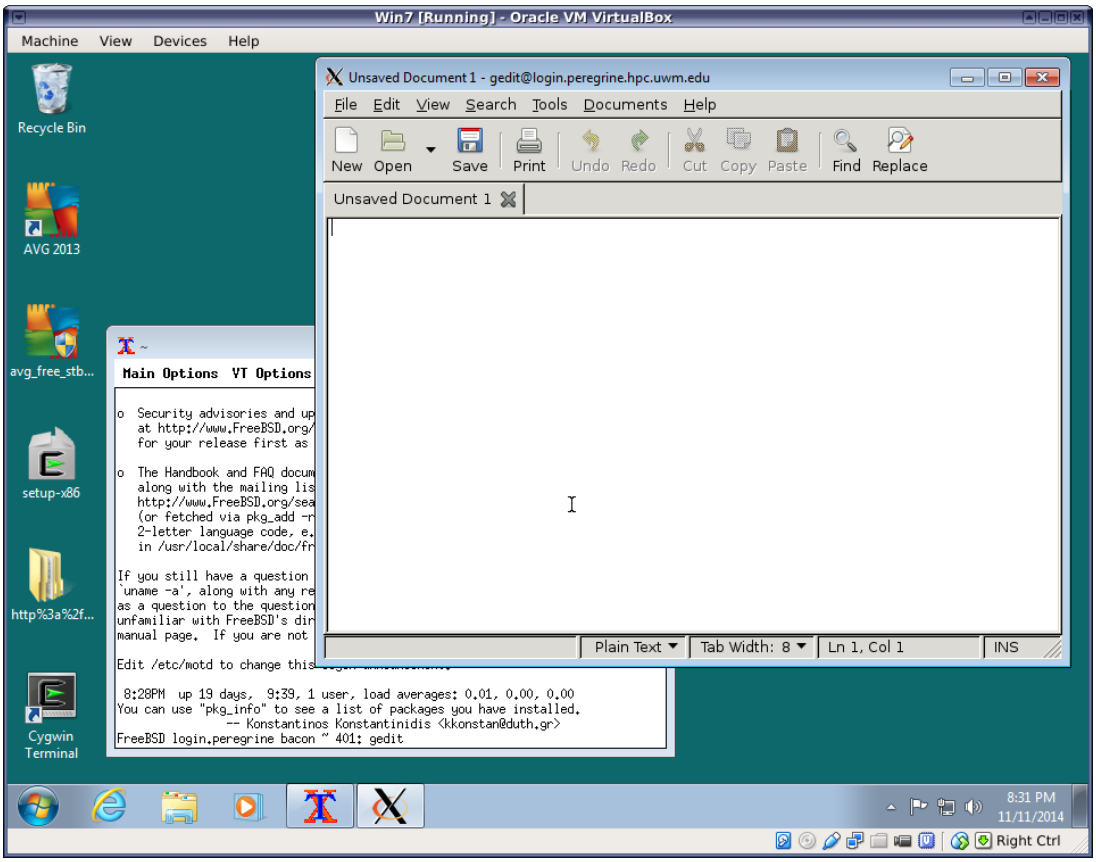
Depending on your Cygwin setup, this might automatically open a terminal emulator called "xterm", which is essentially the same as the standard Cygwin terminal, although it has a different appearance. You can use it to run all the same commands you would in the standard Cygwin terminal, including ssh. 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 30.2.3 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.

30.2 Remote Graphics

30.2.1 Background

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 eclipse 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 remote Unix system to display graphics on your screen. The programs running on the remote system are called *clients*, and they display graphical output by sending commands (magic sequences) such as "draw a line from (x1,y1) to (x2,y2)" to the *X11 server* on the machine where the output is to be displayed. The computer in front of you must be running an X server process in order to display Unix graphics, regardless of whether the client programs are running on your machine or a remote machine.

Apple's macOS, while Unix compatible at the API and command-line, does not include X11 by default. It instead uses Apple's proprietary GUI. X11 for macOS is provided by the XQuartz project: https://www.xquartz.org/. XQuartz is free open source software (FOSS) that can be downloaded and installed in a few minutes. It should start automatically when either local or remote X11 clients attempt to access your display, but you can also start it manually.

30.2.2 Configuration Steps Common to all Operating Systems

Modern Unix systems such as BSD, Linux, and macOS have most of the necessary tools and configuration in place for running remote graphical applications. Some Unix servers may be configured without a GUI. If you want to remotely log into such a server and run X11 programs from your desktop or laptop system, you will need to at least install the **xauth** package on the remote system. This allows your system to configure X11 permissions for the remote system when you log in using **ssh -X** or **ssh -Y**.

```
# Debian
shell-prompt: apt install xauth

# FreeBSD
shell-prompt: pkg install xauth

# RHEL
shell-prompt: yum install xorg-x11-xauth
```

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. Additional steps that may be necessary for Cygwin systems are discussed in Section 30.2.3.

If you want to run graphical applications on a remote computer over an ssh connection, you will need to 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@unixdev1.ceas.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 trick you into giving them your 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 instead of `-X` to open a *trusted* connection.

```
shell-prompt: ssh -Y joe@unixdev1.ceas.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 we connect 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.

```
shell-prompt: ssh -C -X joe@unixdev1.ceas.uwm.edu
```

Caution Using `-C` over a fast connection, such as a gigabit network, may actually slow down the connection, since the CPU may not not be able to compress data fast enough to use all of the network bandwidth.

30.2.3 Graphical Programs on Windows with Cygwin

It is possible for Unix graphical applications on the remote Unix machine to display on a Windows machine with Cygwin, but this will require installing additional Cygwin packages and performing a few configuration steps on your computer in addition to those discussed in Section 30.2.2.

Installation

You will need to install the x11/xinit and x11/xhost packages using the Cygwin setup utility. This will install a basic 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 30.2.2.

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. For **tcsh**, add the following to your `.cshrc` or `.tcshrc`:

   ```
   setenv DISPLAY ":0.0"
   ```

   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 the Cygwin 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 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 as described above.

30.2.4 Remote 3D Graphics

It is possible to run Open 3D graphical applications on remote systems as well, but performance may or may not be acceptable. There are also numerous problems that can arise that depend on the operating system and video drivers used on each end. In any case, OpenGL applications will require additional X11 components to be installed on both the remote machine running the application and the display machine running the X11 server. Determining the minimal set of packages required for every platform would be excessively difficult, so we recommend simply install the entire Xorg system on the remote system. This will likely enable at least some OpenGL applications to run remotely. This proved sufficient to run the 3D mesh visualizer MeshLab comfortably on a remote FreeBSD machine from a FreeBSD display. A similar application called VMD proved too sluggish to use remotely.

```
# Debian
shell-prompt: apt install xorg

# FreeBSD
shell-prompt: pkg install xorg
```

You may find it easier to simply download the data files to your local machine and run the 3D graphics applications there. Performance will be much better when using *direct rendering*, where the display is on the same machine running the 3D application. When using *indirect rendering*, where the 3D application is running on a different machine than the display, sending graphics commands over a network can be a bottleneck.

30.2.5 Practice

Note Be sure to thoroughly review the instructions in Section 30.3 before doing the practice problems below.

---

1. What is X11?
2. Does Apple's macOS use X11? Explain.
3. What must be installed at minimum on a remote computer to allow X11 client programs to run there and display graphics on the X11 display in front of you?
4. Show a Unix command that logs into unixdev1.ceas.uwm.edu and allows us to run remote graphical programs.
5. Show a Unix command that logs into unixdev1.ceas.uwm.edu and allows us to run remote graphical programs over a slow network.
6. Why is setting ForwardX11Trusted not generally a good idea?
7. What packages are needed on a Cygwin setup to enable X11?
8. What alternative do you have if running a 3D graphics program remotely proves to be too complicated or slow? How will this help?

30.3 Practice Problem Instructions

- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest version of this document.
- Read one section of this document and corresponding materials if applicable.
- Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
- Do the practice problems *on your own*. Do not discuss them with other students. If you want to help each other, discuss *concepts* and illustrate with different examples if necessary. Coming up with the correct answer on your own is the only way to be sure you understand the material. If you do the practice problems on your own, you will succeed in the subject. If you don't, you won't.

  If you're still not clear after doing the practice problems, wait a while and do them again. This is how athletes perfect their game. The same strategy works for any skill.
- Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Remove all words that don't add value to the explanation. Brevity and clarity are the most important aspects of good communication. Unnecessarily lengthy answers are often an attempt to obscure a lack of understanding and may lead to reduced grades. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Check the answer key to make sure your answer is correct and complete.

  DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE BEST OF YOUR ABILITY. In doing so, you only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.
- ALWAYS explain your answer. No exceptions. E.g., justify all yes/no or other short answers, show your work or indicate by other means how you derived your answer for any question that involves a process, no matter how trivial it may seem, draw a diagram to illustrate if necessary. This will improve your understanding and ensure full credit for the homework.
- Verify your own results by testing all code written, and double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.
- For programming questions, adhere to all coding standards as defined in the text, e.g. descriptive variable names, consistent indentation, etc.

Chapter 31

Index