

AERE 361

# Computational Techniques for Aerospace Design

Laboratory Manual

**IOWA STATE UNIVERSITY**  
OF SCIENCE AND TECHNOLOGY



Spring 2022

Prof. Kristin Yvonne Rozier

Brian Kempa

Dr. Jianwen Li

Rohit Dureja

Hang Li

Zachary Luppen

# Contents

<b>1</b>	<b>Git, Markdown, and LaTeX</b>	<b>8</b>
1.1	Introduction . . . . .	9
1.2	Git-ing the Assignment . . . . .	9
1.2.1	Sign Up with GitHub . . . . .	9
1.2.2	Joining The Class . . . . .	11
1.2.3	Goodbye to Graphics . . . . .	12
1.3	Logging into Linux . . . . .	12
1.3.1	The Matrix has you Neo . . . . .	13
1.4	The Command Line . . . . .	13
1.4.1	A Shell Game . . . . .	13
1.4.2	Prompt Response . . . . .	13
1.4.3	Where Am I? . . . . .	14
1.4.4	Moving Around . . . . .	14
1.4.5	Master of All You Survey . . . . .	14
1.4.6	Making Some Changes . . . . .	15
1.4.7	There is no trash to save you . . . . .	18
1.5	Git Basics . . . . .	18
1.5.1	GitHub . . . . .	19
1.5.2	Git-ing Help . . . . .	19
1.5.3	Git Going . . . . .	20
1.5.4	A Game of Clones . . . . .	21
1.5.5	Status Symbols . . . . .	22
1.5.6	Branching Out . . . . .	22
1.5.7	No Fear of Commitment . . . . .	22
1.5.8	Sharing is Caring . . . . .	24
1.5.9	Git-Foo . . . . .	24
1.6	Marking Up With Markdown . . . . .	24
1.6.1	Extreme Makeover: Markdown Edition . . . . .	25
1.6.2	Pattern Matching . . . . .	25
1.6.3	Checking the boxes . . . . .	26
1.6.4	Make Your Markdown Masterpiece . . . . .	26
1.6.5	That's some nice work you got there..... . . . .	26
1.7	L <sup>A</sup> T <sub>E</sub> X . . . . .	26
1.8	Turning It All In . . . . .	28
1.8.1	Ready for Launch . . . . .	28
1.8.2	Lift-Off . . . . .	28
1.9	Conclusion . . . . .	30
	References . . . . .	30
<b>2</b>	<b>Operating System Essentials</b>	<b>31</b>

2.1	Standardized Syntax	32
2.2	Simple Commands	32
2.3	The Most Important Command	33
2.3.1	Ask, and You Shall Receive	34
2.3.2	RTFM: Read the Fabulous Manual	34
2.3.3	Keeping Your Options Open	34
2.3.4	System Information	35
2.4	Navigating Linux	35
2.4.1	Creating New Directories	35
2.4.2	Viewing Content In A Directory	36
2.4.3	Changing Directories	37
2.4.4	Creating A File	38
2.4.5	Viewing A File	38
2.4.6	Moving Files and Directories	40
2.4.7	Removing Files and Directories	42
2.5	The UNIX philosophy	43
2.6	The Chain Gang	43
2.6.1	Less is More	43
2.6.2	“grep”ing Text	43
2.6.3	Finding Files and Directories	44
2.6.4	Wildcards	44
2.7	Exercises	44
2.7.1	Grading	44
2.7.2	Report	45
2.7.3	Exercises	45
2.7.4	Lab work submission	46
<b>3</b>	<b>Scripting with Bash</b>	<b>48</b>
3.1	Regular Expressions	49
3.1.1	Literal Matches	49
3.1.2	Matching Any Character	49
3.1.3	Bracket Expressions	50
3.1.4	Repeat Pattern Zero or More Times	50
3.2	Bash as a language	50
3.2.1	Variables	50
3.2.2	Flow Control	51
3.2.3	Functions	51
3.3	Exercises	51
3.4	Report	54
3.4.1	Report Requirements	55
3.5	Lab work submission	55
<b>4</b>	<b>Hello, C!</b>	<b>57</b>
4.1	Emacs and Editing	57
4.1.1	Invoking Emacs	58
4.1.2	Basic Emacs Commands	58
4.1.3	Customizing Emacs for C programming	59
4.2	The GCC Compiler	60
4.3	Hello World!	61
4.4	Data Types	61
4.4.1	Integer Types	61
4.4.2	Floating-Point Types	62
4.4.3	Variable Usage in C	63
4.4.4	Variable Arrays	63

4.4.5	Passing arguments to your program	64
4.5	<code>printf</code> and <code>scanf</code>	64
4.6	Debugging with <code>printf</code>	66
4.7	Mind Your Types!	66
4.8	Functions	68
4.9	Conditionals and Loops in C	69
4.10	A Basic Introduction to Complexity	70
4.10.1	$O(1)$ Complexity	70
4.10.2	$O(n)$ Complexity	70
4.10.3	$O(n^2)$ Complexity	71
4.11	A First Computational Program	71
4.12	Report Requirements	72
4.13	Lab work submission	74
<b>5</b>	<b>More C ... and Complexity</b>	<b>75</b>
5.1	File Management	75
5.1.1	Redirection	75
5.1.2	C File I/O	76
5.1.3	Open a file	76
5.1.4	Read from or write to a file	77
5.1.5	Close a file	78
5.1.6	Special file pointers	79
5.2	Runtime Memory	79
5.2.1	Manual Memory Management	79
5.2.2	Pointer Basics	80
5.2.3	<code>malloc</code>	81
5.2.4	<code>free</code>	82
5.2.5	Example	82
5.2.6	Seg-Fault	83
5.2.7	Foot-Guns	83
5.3	Arrays	83
5.3.1	Initializing Arrays	83
5.3.2	Multi-Dimensional Arrays	85
5.3.3	Array Bounds	86
5.4	Exercises	87
5.4.1	Report	90
5.4.2	Lab Submission	91
<b>6</b>	<b>Debugging with <code>gdb</code></b>	<b>92</b>
6.1	Automatic Memory	92
6.2	Compile-Time Debugging	93
6.2.1	More Warnings!	93
6.3	GDB Basics	93
6.4	Exercises	100
6.4.1	Lab Submission	102
<b>7</b>	<b>All About Pointers</b>	<b>103</b>
7.1	Memory Layout of C Programs	103
7.1.1	The Text Segment	103
7.1.2	The Data Segment	104
7.1.3	The Stack	104
7.1.4	The Heap	105
7.1.5	The <code>size</code> command	106
7.2	Allocating and Managing Memory: beyond <code>malloc</code> and <code>free</code>	106

7.2.1	<code>calloc</code>	106
7.2.2	<code>realloc</code>	107
7.3	Introduction to Valgrind	107
7.3.1	Running example	107
7.3.2	Prepare the program	108
7.3.3	Running your program under Memcheck	108
7.4	A Summary of Errors Valgrind can report	110
7.4.1	Illegal read / write errors	110
7.4.2	Use of uninitialized values	110
7.4.3	Illegal frees	111
7.4.4	Passing system call parameters with inadequate read/write permissions	111
7.4.5	Overlapping source and destination blocks	112
7.5	Use Valgrind in your programming	112
7.6	Function Pointers	113
7.7	Project – Numerical Integration	114
7.7.1	Exercises	115
7.7.2	Report	116
7.7.3	Lab Submission	116
<b>8</b>	<b>Makefiles, Libraries, and Linking</b>	<b>117</b>
8.1	Libraries	117
8.1.1	The gold standard (library)	117
8.1.2	Include Files	118
8.1.3	Compiling Libraries	118
8.1.4	Compiling with Libraries	118
8.2	<code>make</code>	119
8.2.1	Why use <code>make</code> ?	119
8.2.2	What makes a <code>Makefile</code>	119
8.2.3	A sample project	120
8.2.4	Writing a <code>Makefile</code> from scratch	121
8.2.5	<code>Makefile</code> processing	122
8.2.6	Working with <code>Makefiles</code>	123
8.2.7	<code>make</code> is for automating all kinds of compilation	124
8.2.8	Using <code>make</code>	124
8.3	Advanced Data Structures: <code>struct</code>	125
8.3.1	What is <code>struct</code>	125
8.3.2	How to create a structure	125
8.3.3	How to declare structure variables?	125
8.3.4	How to initialize structure members?	126
8.3.5	How to access structure elements?	126
8.3.6	What is a structure pointer?	127
8.3.7	Allocating Structures	127
8.4	Project – Library Usage	128
8.4.1	Building a library	128
8.4.2	Using a library	130
8.4.3	Report	132
8.4.4	Lab Submission	132
<b>9</b>	<b>Machine Numbers and the IEEE 754 Floating-Point Standard</b>	<b>133</b>
9.1	Number Representation	133
9.1.1	Decimal-to-Binary Conversion	133
9.1.2	Binary-to-Decimal Conversion	135
9.2	Integer Representation	136
9.2.1	n-bit Unsigned Integers	136

9.2.2	Signed Integers	137
9.2.3	n-bit Sign Integers in Sign-Magnitude Representation	137
9.2.4	n-bit Sign Integers in 1's Complement Representation	138
9.2.5	n-bit Sign Integers in 2's Complement Representation	139
9.3	Floating-Point Numbers	141
9.3.1	IEEE-754 32-bit Single-Precision Floating-Point Numbers	143
9.3.2	IEEE-754 64-bit Double-Precision Floating-Point Numbers	145
9.3.3	Summary	146
9.4	Machine-Epsilon	147
9.5	Approximation Errors	147
9.5.1	Floating-Point Arithmetic	148
9.6	Type Conversion in C	150
9.7	Exercises	151
9.7.1	Lab Submission	154
9.8	Sources	154
<b>10</b>	<b>The Matrix</b>	<b>155</b>
10.1	Essential Computer Architecture	155
10.1.1	Memory Hierarchy	156
10.1.2	Locality	157
10.1.3	Cache Hits/Misses	157
10.2	Accessing Matrices and Cache Efficiency	159
10.2.1	A Matrix as a 2-dimensional Array	159
10.2.2	An Evaluation on Matrix Allocation	164
10.3	Order matters	166
10.3.1	Counting Cache Misses	167
10.4	Exercises	168
10.4.1	Lab Submission	170
<b>11</b>	<b>Systems of Linear Equations</b>	<b>171</b>
11.1	Matrix Representation of Systems of Linear Equations	171
11.2	Direct Methods	172
11.2.1	Graphical	172
11.2.2	Cramer's Rule	172
11.2.3	Elimination Method	173
11.2.4	Limitations of Elimination Methods	174
11.2.5	Gauss-Jordan	175
11.2.6	LU Decomposition & Solve	176
11.3	Approximate Methods	176
11.3.1	Jacobi Method & Gauss-Seidel	176
11.4	Exercises	176
11.4.1	Lab Submission	178
<b>12</b>	<b>Pair Programming</b>	<b>180</b>
12.1	Introduction to gnuplot	180
12.1.1	Basic Plotting	180
12.1.2	Plotting Data	181
12.1.3	gnuplot Hints	182
12.2	System Calls in C	183
12.2.1	system()	183
12.2.2	popen()	183
12.3	Unions and the void	183
12.3.1	Unions	183
12.3.2	Void Pointers	184

12.4	Project: Drone Surveillance Challenge	185
12.4.1	The Rules	185
12.4.2	The Tools	186
12.4.3	Running Your Program	186
12.4.4	The Twist	186
12.4.5	Challenges	186
12.4.6	Player 2 Has Joined the Game	187
12.4.7	Deliverables	187
12.4.8	Points	187
12.5	L <sup>A</sup> T <sub>E</sub> X Report	187
12.5.1	Lab Submission	188
<b>13</b>	<b>Circuit Solver</b>	<b>189</b>
13.1	Motivation	189
13.1.1	Circuit Representation	189
13.2	Project	192
13.2.1	Solving	192
13.2.2	Input	192
13.2.3	Output	193
13.2.4	Grading	193
13.2.5	Report	193
13.2.6	Lab Submission	193

# 1 | Git, Markdown, and LaTeX

*Reference:*

**Introduction to Scientific and Technical Computing:**  
Chapter 3: Developing with Git and Github

## Purpose

Introduction to tools used for file management and documentation in modern engineering (and the rest of this course).

## Objectives

- Work from the engineering Linux Servers
- Use Git for version control
- Retrieve and submit assignments with GitHub
- Write documents in Markdown and LaTeX

## Motivation

Modern engineering is a team sport, and one of endurance. For example, the Boeing 787 was officially announced in January of 2003 and finally flew a commercial flight in October of 2011[1].

With thousands of engineers working between divisions and across subcontracting companies, pulling data from older programs and handling turnover as staff are hired, transferred and retired it would be impossible to estimate how many thousands of individuals had to collaborate to create this plane.

To manage all of this, companies invest heavily in "knowledge management" systems to store and categorize information. Think of the amount of time and money that would be wasted if the engineers had to redo work or decipher poorly-documented results and code because they couldn't find or understand work already done by one of the many thousands of their predecessors or peers. Not to mention this makes for a boring job for the engineer re-inventing the wheel.



In this course we will be using Git to manage documents such as source code files and lab reports, and a combination of Markdown and L<sup>A</sup>T<sub>E</sub>X to provide documentation. The reasoning and advantages will be explained over the next few weeks as we explore these tools. And while these tools were selected partially because they are actually used in industry, the important part is learning how such systems work and how to learn new tools because the tool you may use for your job may not even be invented yet.

## 1.1 Introduction

This lab manual contains instructions, expectations, hints and references for the course. Initially it will be written as a walk-throughs of the labs, but later labs will consist of a guided introduction to new skills followed by a project applying the skill left to the student.

This first lab steps through the procedure that will be used for future assignments, but mostly omits the technical background of how and why various components work. This may seem like magic at first ("just type this incantation"), but the next lab picks up where lab one ends by breaking down what we've accomplished and will begin to make you comfortable in this environment.

For Spring 2022 the TAs for the course will be Gage Harris <[gharris@iastate.edu](mailto:gharris@iastate.edu)> , Ahmed Ellithy <[ellithy@iastate.edu](mailto:ellithy@iastate.edu)> and Prashant Thapaliya <[prashant@iastate.edu](mailto:prashant@iastate.edu)>. Office hours will be announced in class and are posted to the [course website](#). Lab questions, course feedback and manual corrections are accepted at [361ta@temporallogic.org](mailto:361ta@temporallogic.org).

## 1.2 Git-ing the Assignment

Lab assignments will be distributed, submitted and graded through a system called GitHub Classroom. GitHub is a website that hosts Git repositories which will be explained later. To begin, you will need an account on [github.com](https://github.com). We will use a web-browser on the local machine to create and manage your account.

*Note:*

Local machine refers to the computer sitting in front of you. This is to differentiate from the remote machine, which is the server where you will be working later.

### 1.2.1 Sign Up with GitHub

1. Open a web browser like Chrome
2. Navigate to [github.com](https://github.com)
3. If you don't already have an account, skip to the next step. If you already have an account:
  - (a) Click "Sign In" in the upper right corner
  - (b) Complete the sign-in process and skip to the next section [1.2.2](#)

4. Click "Sign Up" in the upper right corner
5. Fill in a username (something you wouldn't mind writing on a resume), an email (preferably your @iastate.edu address, see tip below), and a password. (Figure 1.1) Click "Create an account"

Join GitHub

The best way to design, build, and ship software.

Step 1: Create personal account

Step 2: Choose your plan

Step 3: Tailor your experience

Create your personal account

Username

something-professional

This will be your username. You can add the name of your organization later.

Email address

AerStudent@iastate.edu

We'll occasionally send updates about your account to this inbox. We'll never share your email address with anyone.

Password

Use at least one lowercase letter, one numeral, and seven characters.

By clicking on "Create an account" below, you are agreeing to the [Terms of Service](#) and the [Privacy Policy](#).

Create an account

You'll love GitHub

Unlimited collaborators

Unlimited public repositories

Great communication

Frictionless development

Open source community

Figure 1.1:

6. The default settings are fine on the next page (Figure 1.2), click "Continue"

Welcome to GitHub

You've taken your first step into a larger world, @something-professional.

Completed

Step 2: Choose your plan

Step 3: Tailor your experience

Choose your personal plan

Unlimited public repositories for free.

Unlimited private repositories for \$7/month.

Don't worry, you can cancel or upgrade at any time.

Help me set up an organization next

Organizations are separate from personal accounts and are best suited for businesses who need to manage permissions for many employees. [Learn more about organizations](#)

Send me updates on GitHub news, offers, and events

Unsubscribe anytime in your email preferences. [Learn more](#)

Continue

Both plans include:

Collaborative code review

Issue tracking

Open source community

Unlimited public repositories

Join any organization

Figure 1.2:

7. You can fill out the survey on the third page (Figure 1.3), or click "skip this step" at the bottom

# Welcome to GitHub

You'll find endless opportunities to learn, code, and create, @something-professional.

✓ Completed  
Set up a personal account

🔧 Step 2:  
Choose your plan

⚙️ Step 3:  
Tailor your experience

How would you describe your level of programming experience?

☐ Totally new to programming    ☐ Somewhat experienced    ☐ Very experienced

What do you plan to use GitHub for? (check all that apply)

☐ Development    ☐ School projects    ☐ Project Management  
☐ Research    ☐ Design    ☐ Other (please specify)

Which is closest to how you would describe yourself?

☐ I'm a student    ☐ I'm a professional    ☐ I'm a hobbyist  
☐ Other (please specify)

What are you interested in?

e.g. tutorials, android, ruby, web-development, machine-learning, open-source

[Submit](#) [skip this step](#)

Figure 1.3:

You now have a GitHub.com account!

*Tip:*

If you sign up with your @iastate.edu email address (or add it to an existing account) you can get free stuff! See [GitHub Education Pack](#) for a list of freebies, and [GitHub Help](#) for more information on the verification process (after lab, of course).

## 1.2.2 Joining The Class

When each lab is released, you will receive an invitation link to the assignment.

1. Open your Iowa State email (CyMail) at [cymail.iastate.edu](mailto:cymail.iastate.edu)
2. Log in with your NetID
3. Find an email sent to the class email list with the autograder enrollment link

*Tip:*

If you just created your GitHub account, now would be a good time to click the email verification link they sent you as well.

4. Fill out the enrollment form. Once submitted you will be given the link to lab 1 and the autograder will email you log-in credentials for a server.
5. Click the link in the form completion message.
6. A new page will open, asking you to accept the assignment. Click the "Accept" button

7. The bottom of the next page will state "Your assignment has been created here: " followed by a unique link. This link will contain the name of the assignment and your GitHub username. Click this link.

Congratulations! You now have a Git repository with a copy of the assignment. This repository is special; it is not owned by you, but by the class. This allows your instructors to view your assignment for assistance and grading, but prevents the public or other students from seeing your work.

### 1.2.3 Goodbye to Graphics

Keep this browser window open, we will be using it later. We will now log into the remote server where most of the work of this course will occur. Most of our work in this class will be done from the command line, a text-only way of operating a computer. All these concepts will be explained in the next two sections. For now, let's say goodbye to the graphical user interface (GUI) that we've been using and enter the world of command line interfaces (CLI).

## 1.3 Logging into Linux

The *Operating System* (OS) is a collection of software that works together to run a computer. Microsoft Windows and MacOS are common operating systems for personal computers, but another OS is used in the largest and smallest devices. Linux is run by 100% of the TOP500 ranked supercomputers [4] and is the leading OS for embedded devices, hidden computers that run inside larger machines. The thermostat on your oven, the radio in your phone and countless other devices you touch everyday probably run Linux.

While Aerospace engineers have a diverse set of skills and interests, Linux finds use in almost all of them. We are interested in Linux because its main uses are in control systems; like on a plane, drone, or spaceship, or on supercomputers and computing clusters for use in simulations. Knowing how to work with Linux is a desirable skill for all specialties of aerospace engineering. Also, using Linux allows us to focus on the basics of computers as engineering tools in later lessons instead of dealing with specifics of the operating system.

This course will use the individual, virtual, Linux servers. The next lab will focus on comfortably operating these machines and why they are used. For this assignment, all Linux operations will be walked through step-by-step.

All instructions are given assuming usage of the lab computers. An introduction to accessing these servers from elsewhere will be provided later. It is anticipated that you will complete this assignment during lab time, but if not it can be continued from any lab computer as homework.

1. Open the Start menu by clicking the windows icon in the bottom left corner of the screen
2. Type "Remote" (You don't need to click any text boxes. Once the menu is open, just start typing.)
3. Select "Remote Desktop Connection" from the list

4. In the box labeled "Computer" type the URL that was emailed to you by the autograder and click "connect".
5. A caution pop-up will appear asking if you trust this server, select "yes".
6. You will now be at the login screen for the Linux server, use the username and password provided in the email.

*Note:*

In previous semesters, this course has used the College of Engineering servers, but we will be using the department servers. Don't be confused by the screenshots showing connections to different server names. Use your assigned server.

### 1.3.1 The Matrix has you Neo

You should now be at the Linux server desktop. To get access to the command line, double click the "LXTerminal" icon on the desktop and maximize the resulting window with the plus icon in the corner. You now have a black window with white text and a green blinking box. Does this mean we are hackers now? Well, no. But you do have license to feel like Neo and wear all black and sunglasses to lab if you wish.

You are now connected to the Linux server! Now we have to navigate this strange, text-only world.

## 1.4 The Command Line or How I Learned to Stop Worrying and Love the Shell

You are now at the command line. In the days before GUIs this is how computers were controlled. Actually, they are still used this way. Not only do many programmers and engineers intentionally work at a command line (for some tasks, it is much more powerful and quicker) but GUIs are just wrappers on this lower level of control.

### 1.4.1 A Shell Game

To begin, type "clear" then hit enter. The screen should go blank, then some text will appear in the top left corner. This text is called the prompt. The prompt is written to the screen by a program called a shell. The shell takes your commands, written on the command line or in a file (we will cover this in another lab) and executes them by calling the operating system, other programs, or a combination. There are many different shells, but the default is called bash and stands for Bourne Again SHell. We will be using bash for this course. The next two labs will focus on getting comfortable with the shell, and using it to do useful work — for today you will be given step-by-step instructions.

### 1.4.2 Prompt Response

Let's examine the prompt. By default it looks something close to [Figure 1.4](#)

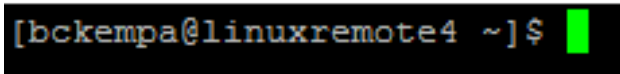


Figure 1.4: A default bash prompt

The prompt can be customized to show many different things, but by default it shows 4 pieces of information:

1. username: before the @ (bckempa in Figure 1.4)
2. server name: after the @ (linuxremote4 in Figure 1.4)
3. working directory: we will cover this next (~ in Figure 1.4)
4. permission level: The \$ indicates a normal user. If you were working with administrator privileges it would be a #

Finally the green block. It isn't actually part of the prompt, it is part of terminal and it is the cursor showing you where text will appear when you type.

### 1.4.3 Where Am I?

When working at the command line, the shell has a "working directory." This is the folder (often called a directory in Linux) the shell is currently sitting in. Let's find out where we are. Type "pwd" and hit enter. you should have gotten "/home/your\_username\_here" in response. The command "pwd" stands for Print Working Directory and asks the shell to print the path to the folder it is sitting in. By default, when you log in the shell starts in your home directory.

A path is a list of folders from the root of the computer (/) to whatever folder or file you want to refer to. We will work much more with paths next week.

#### *Note:*

A ~ symbol is a shortcut for your home directory, that's why the prompt displays ~ instead of "/home/your\_username\_here" when you first log in. More on this next week!

### 1.4.4 Moving Around

To change directories, we use the command "cd" (for "Change Directory") and specify the path. So to go to my desktop directory I would type "cd Desktop" for example.

Try this now, use cd to change into your desktop directory, then use pwd to confirm you are in the right place. Notice how the prompt changed to show the directory?

### 1.4.5 Master of All You Survey

Being in a folder isn't very useful if you don't know what other files and folders are there with you. To shed some light on things type "ls" (as in list) and hit enter.

*Note:*

Linux is case sensitive. That means "desktop" isn't the same as "Desktop" and if you try to "cd desktop" bash will complain that there is no directory.

Now that you are in the Desktop, run the ls command again. This should look very familiar, it has the same shortcut we used to launch the terminal.

### 1.4.6 Making Some Changes

There are several programs available for making changes to text files. You might use Notepad or Word in Windows, or TextEdit or Pages in MacOS, in Linux you might use nano, vi or emacs. We will use emacs to start with, it's very powerful once you get to know it and is pretty intuitive.

**Let's write a file!**

1. Use the "pwd" command to verify that we are in the Desktop directory, if not review the last section
2. Use the "ls" command to look at what files and directories are on your Desktop
3. Type "emacs foo.txt" and hit enter to make a new file called "foo.txt" and open it in the emacs text editor.

*Note:*

Explain the name "foo.txt"

Foo is a common [metasyntactic variable](#). It is a placeholder name often used by programmers when something needs to be referred to, but it doesn't matter what it is. In programming foo, bar, baz, and qux are commonly used. You will see these names used in many programming examples in both textbooks and this manual. Using foo to mean "something" is like using John Doe to mean "someone".

Windows and MacOS normally handle file extensions (the part of the name after the last dot) for you, sometimes even keeping them invisible. In Linux you are often responsible for these yourself. Here we use ".txt" to signify that this is just a text file. We will cover file extensions more next lab.

4. Emacs will open now as in [1.5](#). Note the messages at the bottom of the screen, these will direct you to the tutorials on how to make full use of emacs. Much of its power is beyond the scope of this course, but it is worthwhile to look up and remember some hotkeys. You can get rid of this message (or buffer) by right clicking the bar beneath it.

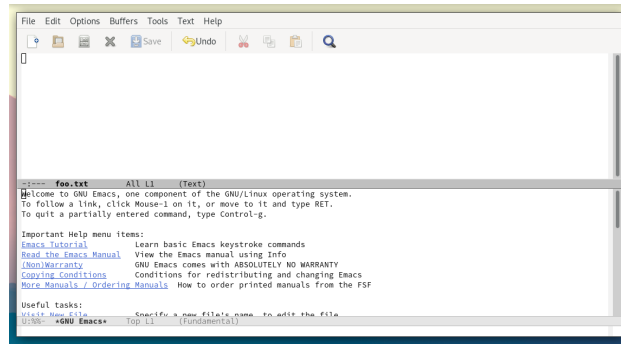


Figure 1.5:

5. We can now add text to our file! Go ahead and start typing something.

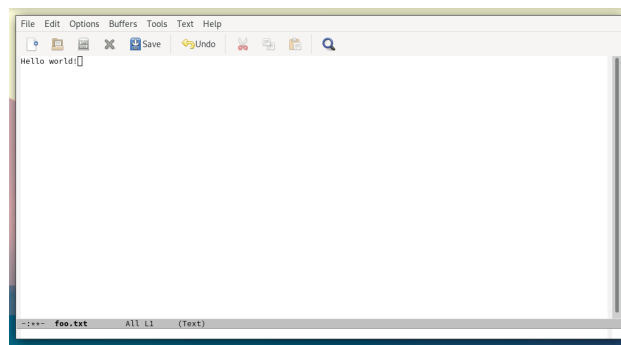


Figure 1.6:

6. We have unsaved changes now. Go ahead and save the file. There is a big "save" button for this at the top, or for you advanced learners, use the C-x C-s hotkey (Ctrl-x followed by Ctrl-s)
7. We now see the status at the bottom says that we wrote our file, so our changes have been saved. Now we can exit by clicking file and selecting Quit, or using the hotkey C-x C-c. Now the terminal you called "emacs foo.txt" from is back and can be used again.
8. Using ls, verify the file is now there. Also, check the desktop of your local machine. The file should have appeared there too.
9. There is also another option of running emacs within the terminal you call emacs from. Try running the command "emacs -nw foo.txt" and try using the hotkeys to save and exit.



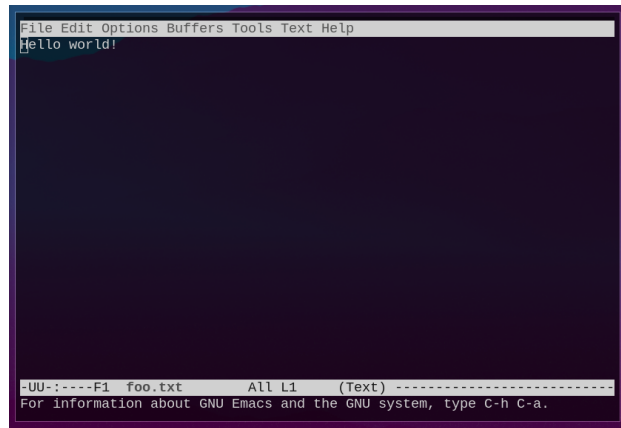


Figure 1.7:

### Reading our file

The next two labs will cover working with files in Linux in much greater depth. For now, a simple way to view a file is with the "less" command. To read our new file, type "less foo.txt" and hit enter. If the view is longer than the screen, you can use the arrow keys to scroll. Press "q" when finished to quit and return to the shell.

#### *Note:*

Why does everything behave slightly differently? Many of these programs and tools were originally written decades ago (and sometimes even decades apart) by many different people working on different systems for different purposes. Modern computing is a pyramid where each layer was made by a different generation. When it first powers on, your powerful modern computer believes it is a tiny weak processor from 1970s for this very reason.

There is plenty of complicated history, filled with cases of "seemed like a good idea at the time" and "if it ain't broke – don't fix it" moments that have led to the near Frankensteinian systems we now have. If this history is relevant or interesting it will be pointed out as we explore.

### Making a quick fix

Let's go ahead and add something extra to our file for fun.

1. Start emacs with "emacs foo.txt"
2. Since we gave emacs the name of an existing file, it will open it for editing rather than making a new one
3. Make some new additions
4. Save with C-x C-s and then exit with C-x C-c
5. Use less to view the changes

Practice this procedure of modifying an existing file

## Goodbye forever!

It's time to say goodbye to `foo.txt`, it has served us well.

1. type `"rm foo.txt"` but don't hit enter yet
2. Re-read that line, is it correct?
3. Ok, and you are sure you are in the correct directory?
4. And that is the file you want to delete?
5. And there is nothing in that file you will ever need again?
6. Ok then, hit enter
7. Check with `ls`, the file is gone. Also check your Windows desktop on your local machine, it is gone there as well

### 1.4.7 There is no trash to save you

Where did the file go? When we use the command line there is no "recycle bin" or "trash" file. Once you delete something, it is gone.

What if we need something back? What if we want to know what a file looked like before a change was made? What if we wanted to see what other changes were made around the same time? Git helps us with all of these questions and more.

## 1.5 Git Basics

You have been repeatedly promised an explanation of repositories — here it is as promised: A repository is a directory with stuff in it.

This sounds like a trick answer but it is important to remember while learning git, at the end of the day the difference between a git repository and a normal directory is a special, hidden `".git"` folder within.

Git is a *Version Control System*. It helps you keep files organized by keeping various versions. This is similar to the process of saving a new copy of a Word document every time you make a change.

- Regret deleting a section? Copy it from an old version.
- Make a total mess of a section while rewriting it? Go back to the way it was before.

You get all these and more benefits without having many copies of the file laying around (which document is the "real" version, `major_paper_final_final_Mon.doc` or `major_paper_complete.doc`?) and it takes much less space on your computer to do so. These are some of the most obvious benefits, and there are many more you will discover as you begin using git.

Another important feature of git is how it is *distributed*. This means that git can help you work with other people on projects by sharing and combining your work. This is even useful when you are the solo developer, because it allows for easy backups. Enter GitHub.

### 1.5.1 GitHub

GitHub is a website that hosts git repositories. This means that you can send a copy of your git repository to GitHub and have a backup which you can access from anywhere with an internet connection.

Not only does this give you flexibility and piece of mind, but it allows us to use GitHub to distribute and collect assignments (that's what GitHub Classroom does, which you signed up for in section [1.2.2](#)).

### 1.5.2 Git-ing Help

Git is very powerful because there isn't actually very much going on under the hood. A proper introduction into the mechanics behind git's operation is beyond the scope of this class, but google is your friend if you are curious.

One trade-off of this power and flexibility: it can be easy to do something unintended and get stuck when first learning git. There are two methods to handle this, one is demonstrated in Figure [1.8](#), the other is by knowing how to ask questions.



Figure 1.8: Don't be like this guy [2]

Git has a built-in help tool which you can access by typing "git help" and pressing enter. This will list some common commands and provide further instructions on using the built-in help system. There are also many options for help available online from sources like [GitHub](#) or [The Git Project](#). If you are really stuck, google your problem, email a TA, or ask your lab instructor.

### 1.5.3 Git Going

Enough talk, let's fight.

— Kung-Fu Panda

Before we can work with git, we need to tell it who we are. It uses this information to make note of who made what changes – this becomes very useful when several people are working on a project together.

To tell git our name, type the following into the terminal, replacing "Your Name" with your name but leaving it quoted and hit enter:

```
git config --global user.name "Your Name"
```

To add our email address, type the following (replacing the dummy value with your email but leaving it quoted) and hit enter:

```
git config --global user.email "your_email@whatever.com"
```

We will now grab a copy of our assignment from GitHub.

#### 1.5.4 A Game of Clones

Many git tutorials begin with instructions on making a new repository. We don't actually have to do this, because GitHub Classroom created a repository for us. However, this repo is on the GitHub servers, and we need a copy on our Linux machine. The answer is cloning!

1. Go to the GitHub page you kept open earlier
2. On the right side of the screen, click the green "Clone or download" button to get a drop down like in Figure 1.9

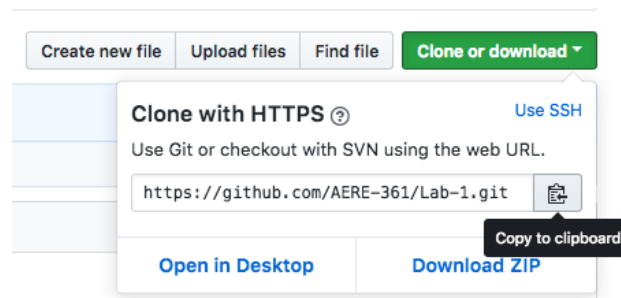


Figure 1.9:

3. Make sure the dropdown says "Clone with HTTPS", if it says SSH instead, look for the blue text to the right that will say "use HTTPS" and click that
  4. Click the clipboard icon to copy the URL, note that your URL will be unique to you
  5. Back in the terminal, make sure you are in your home directory (`cd ~`)
  6. Type `git clone`  (no quotes, note the space at the end and don't hit enter yet...)
  7. To paste the URL from the clipboard, we have to right click the window because `ctrl-V` sends the key command to the shell, not the terminal program itself.
  8. When the command line says `git clone https://github.com/AERE-361/YOUR_REPO_NAME` press enter
  9. Git will ask you for your GitHub username and password
  10. After authenticating, git will create a new directory with the same name as your GitHub repo and clone into it
  11. Change directories (`cd`) into this new folder. If you forget what it is called, you can use `ls` to check the names of files and directories in your current working directory
- Great! We now have a "working copy" of our repository!

### 1.5.5 Status Symbols

The command you will use most often is "git status" which asks git about our current repository. Among other things, git will report what branch we are on (more on that next) and if there are any new or changed files since our last checkpoint.

Try out this command a few times, making some changes or new files to become comfortable with reading the format. When you are done, type the following to remove your changes and make the repository look like when you first checked it out:

*Warning:*

These commands will cause you to lose whatever changes you made since the last commit – this is almost never what you want so think very carefully before doing this in the future.

Type

```
git clean -f && git reset --hard
```

and press enter. Git will remove new files and reset any modified files to the state they were in when cloned.

### 1.5.6 Branching Out

One of the most powerful features of git is branching. Branches let us keep separate sets of changes to our repository and can be used for many useful things. Today we will just use two branches. By default, all repositories have one branch called master. We are going to add a 2nd branch called develop, where we will put our changes while we work. When we are done, we will use GitHub to tell the TAs that we have completed our work on develop and are ready to put the finished assignment back on the master branch. Next week we will talk more about git workflows and branching, for now just follow along.

1. To make a new branch named develop type "git branch develop" and hit enter
2. We have made a new branch, but we are still working on the master branch, which git will tell us if we run git status again. To move over to the new develop branch we have to use the checkout command: "git checkout develop". Run this now, and it should report "Switched to branch 'develop'" to indicate success.
3. Running git status again, the first line should report "On branch develop" and if you run ls you will see... nothing has changed.

This is good! Nothing is different yet because we haven't made any changes. When we ran git branch, we told git "make a new branch that starts from the same state that my current branch (master) is in."

### 1.5.7 No Fear of Commitment

Now to make a commit, or checkpoint. This is the "save" feature in git. Once we make a commit we can always come back to this state or look at what we changed to go from one commit to another.

1. Use emacs to make a test file like "emacs foo.txt" and save some text in it

2. Run `git status` and see that git calls the file "untracked" because we haven't told it to keep track of it yet
3. We need to add the file to the index. The index is a list of what changes we want to save in the next commit. Type "`git add foo.txt`" and hit enter to ask git to start keeping track of the file with the next commit
4. Run "`git status`" again to see that your next commit git will save the new file "foo.txt"
5. Now type

```
git commit -m "Add foo.txt"
```

and press enter to tell git to save a checkpoint of the changes in the index. The -m flag tells git to use the following message as the description of the commit. Make sure to use quotes around the message as shown here

*Tip:*

The procedure for making commits is:

```
git add <files that have changed or been added>
```

```
git commit -m "<Your Message>"
```

making sure to replace the angle brackets with the relevant information

Now that you know how, get in the habit of committing the Chicago way: Early and Often! It's a good idea to train yourself to make a commit every time you complete a logical component of work.

- Finish a function? Commit
- Write a section of your report? Commit
- Finally find that one bug that took longer to track down than the rest of the code took to write? Commit

At first this may seem like overkill, but the internet is full of stories, jokes, memes, and interpretive dance poems by programmers who regret not committing more. The opposite is rarely heard.

One last thing - write a good message. If you end up like this your commits aren't nearly as helpful:

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

Figure 1.10: Don't be like this guy either [3]

### 1.5.8 Sharing is Caring

We use "git push" to send our commits to the GitHub server. Doing this regularly makes use of GitHub as a backup of your work and allows anyone else working on the same repository to get the changes you have made.

Git keeps track of other copies of the same repository as "remotes" and when we cloned it, it automatically created a remote for GitHub called "origin" that we can use to send our changes back. To send our changes:

1. Make sure all our work is committed and that we are on our develop branch with git status
2. Type "git push origin develop" and hit enter. Read this as "Push my commits from the current branch, to a remote repo called origin, onto a branch called develop at the origin repository." Don't worry, we will become more familiar with what this incantation means later.
3. Git may ask you to authenticate with GitHub, then it will send the data

### 1.5.9 Git-Foo

Over the next few weeks as you get comfortable with basic git usage, we will begin exploring more advanced uses of the tool.

Now would be a great time to work through the [Git Tutorial](#) website to practice your git skills. It is a guided series of exercises to get comfortable with using git. Note that you only have to complete the sections of this tutorial listed in the repository markdown file. Completing the exercises is part of your assignment this week and now is a great time to reinforce those newly-learned skills!

## 1.6 Marking Up With Markdown

Markdown is a markup language so to understand Markdown we first need to understand markup.



A markup language embeds information about the content or presentation of data along with the data itself. Some examples include:

- **.doc** — The Microsoft Word Document Format  
Word documents store information about how the document is formatted along with the text and images of the document in the .doc files., although you never see it
- **HTML** — HyperText Markup Language  
The standard for websites, it uses a system of tags to note the purpose of each section (title, link, paragraph, etc.) but how each element looks on a given site is defined elsewhere
- **Markdown** — The go-to for README files  
A type of markup called "lightweight markup" it is designed to be readable in plain text and easily written by hand, but easily "rendered" into prettier formats

### 1.6.1 Extreme Makeover: Markdown Edition

Go back to the web browser on your local machine and look at the GitHub page with your lab repository. See all the text below the list of files? That is a README document, a standard inclusion in software repositories explaining their purpose and giving basic instructions. It is written in Markdown (notice the .md extension).

In the terminal, use the less command to view the README.md file. You'll notice that the file is plain text and instead of the nice formatting of the GitHub page, there are some specific patterns used in the document.

Markdown is a set of patterns that can be used to write a document that can be read in plain text (like with less). When a document follows those rules, a computer can display that document with some nicer formatting, like on the GitHub page.

### 1.6.2 Pattern Matching

If you study the README.md file and compare how it looks in less to how it looks on the website, you can probably begin to unravel some of the patterns.

For example, a line that starts with one octothorp (the # sign, or "hash tag") is made to look like a large, bold title. Lines starting with two octothorps become slightly smaller, subtitles.

For a list of all the patterns, see [Mastering Markdown](#). Review this now to prepare for the next segment.

*Note:*

To try out any of the patterns, use [dillinger.io](https://dillinger.io), an online Markdown previewer. Type your Markdown on the left and see the output on the right.

### 1.6.3 Checking the boxes

The README files of the lab assignments are the official source of requirements and grading rubrics for this course. If you peruse the file, you will find a number of empty brackets ([ ]) all denoting an objective or deliverable. As explained in the Mastering Markdown guide linked above, empty braces are rendered as empty checkboxes, and by replacing the space with the letter x ([x]) the box will be rendered as checked. As you go, keep track of your remaining tasks by filling in the checkboxes of objectives you complete.

In the README, you will find the first task under the Mastering Markdown section is to read the Mastering Markdown guide. If you did this as instructed, use emacs to modify the README to reflect this by "checking the box" (replacing the space with an x between the brackets beside the tasks). If you don't remember how to use emacs to modify an existing file refer to section [1.4.6](#).

### 1.6.4 Make Your Markdown Masterpiece

Now that you have some practice with Markdown, it is a good time to begin one of the graded sections of the assignment. In the README, under the section "Mastering Markdown" the final checkbox lists an assignment to write a markdown file about git. Start a draft of this document now.

*Note:*

To accomplish this, here are two more shell commands to help you:

1. "mkdir foo" will create a new directory named foo (or whatever name you place there) in the current working directory
2. "cd .." is a shortcut for "go up one level" and will take you to the parent directory. So if your current working directory was /home/vm-user/Desktop and you ran "cd .." your current working directory would now be /home/vm-user

I always find it helpful to make an outline out of headings before filling them in to organize my thoughts. Once you have some ideas down, move on to the rest of the lab.

### 1.6.5 That's some nice work you got there.....

It would be a shame if somebody forgot to commit that..... Just a friendly reminder.

## 1.7 L<sup>A</sup>T<sub>E</sub>X

Now emacs and markdown can get you a long ways toward a well-documented project, but it is no replacement for Word. That's what L<sup>A</sup>T<sub>E</sub>X is for!

L<sup>A</sup>T<sub>E</sub>X is a system for typesetting and is used for textbooks, journal papers, and can even generate presentation slides. Many of the books you have read, especially technical literature, have been typeset in L<sup>A</sup>T<sub>E</sub>X. So has this lab manual. Advantages of L<sup>A</sup>T<sub>E</sub>X over other formats include ease of handling mathematical formulas, automatically numbering figure references,

automatic bibliography generation and the ability to track L<sup>A</sup>T<sub>E</sub>X documents with version control systems like Git.

You get what you put into L<sup>A</sup>T<sub>E</sub>X, those experienced with its usage often claim they are more productive and spend drastically less time formatting than with "What You See Is What You Get" (WYSIWYG) editors like Word – but if you don't learn the tool now I won't magically save you when you could really use it. So to build these skills, all assignments will require a report created in L<sup>A</sup>T<sub>E</sub>X as part of the submission (think technical report, not large paper).

To assist with this, we will introduce more L<sup>A</sup>T<sub>E</sub>X skills throughout the semester and the expectations for the reports will scale accordingly. This week you will only make a minor modification to see what L<sup>A</sup>T<sub>E</sub>X is.

1. Type "module load texlive" and hit enter. This loads L<sup>A</sup>T<sub>E</sub>X
2. Change directories into the reports directory of your git repository
3. Type "pdflatex main.tex" and hit enter
4. Using your local machine and take a look at the .pdf file that was generated
5. It may have some fancy math, but only placeholder names and dates, let's fix that.
6. Open main.tex with emacs
7. It looks messy and complicated, and there are 2 reasons for that.
  - (a) This file is doing some fancy things to show off
  - (b) emacs can be nice for some quick edits, but is not meant to handle documents like this

*Note:*

Next lab we will do more with L<sup>A</sup>T<sub>E</sub>X. We will start simple, and use the proper tools to make a better writing experience.

8. Now scroll down about 80 lines and find a section marked like this:

```
%
% BEGIN MODIFICATIONS HERE
%
% Homework Details
%   - Title
%   - Due date
%   - Class
%   - Section/Time
%   - Instructor
%   - Author
%
\newcommand{\hmwkTitle}{TITLE}
\newcommand{\hmwkDueDate}{DUE DATE}
```

```

\newcommand{\hwkClass}{CLASS}
\newcommand{\hwkClassInstructor}{INSTRUCTOR}
\newcommand{\hwkAuthorName}{\textbf{NAME}}
%
% END MODIFCATIONS HERE
%
```

9. Change the placeholder information to be correct (Your name for NAME, "Lab 1" for TITLE, etc.) make sure you change the actual content (on the lines starting with "\newcommand") not the comments (on the lines starting with "%")
10. Save and quit emacs, rerun pdflatex and check the file – better now?

*Note:*

When committing, make sure the main.tex and main.pdf files are included for the grader, any other files like main.aux or main.log can be deleted

## 1.8 Turning It All In

Much like your skill with Linux, our process will evolve throughout the semester but for now we will use a GitHub feature called a "Pull Request."

A pull request is a communication feature on GitHub, and not a part of git itself. It gives collaborating members a chance to make a decision together. Instead of rudely pushing your code onto the master branch, a pull request allows you to announce "I have some changes I would like to recommend – what does everyone think." For the purposes of this lab, it will announce to the grading team that you have completed the assignment and you're ready for grading.

### 1.8.1 Ready for Launch

Before submitting your assignment, check the README.md and make sure you have completed every task (check-box)

*Warning:*

Do not proceed until you have accomplished everything listed in the README! If you haven't completed all sections of this assignment, come back later.

Check one last time that you made your commits into a branch (it must be called "develop") and that the head of the branch is the attempt you wish to turn in. Everything completed? Good, we are GO for launch (feel free to count backwards from 10).

### 1.8.2 Lift-Off

1. Use "git push origin develop" to update GitHub with your branch
2. Open a web browser, log into your GitHub account and find your repository

3. Above the README and list of files is a row of buttons similar to Figure 1.11. Click the "New pull request" button toward the lower left



Figure 1.11:

4. Notice the arrow between base and compare like in Figure 1.12? This signifies that the pull request proposed putting the contents of the compare branch on the base branch



Figure 1.12:

5. Use the 2nd drop down to set the develop branch to be merged into the master branch as shown in Figure 1.13






Figure 1.13:

6. Your screen should resemble Figure 1.14. Now fill in a title, any (Markdown-formatted) summary you feel like, and click the big green "Create pull request" button when ready.

## Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across](#)

 base: **master**  compare: **develop** ✓ Able to merge. These branches can be automatically merged.

 Ready for Grading


Write

Preview

AA B i “ <> ↻ ⋮ ⋮ ⋮ ↶ @ 📎

Leave a comment

Attach files by dragging & dropping, [selecting them](#), or pasting from the clipboard.

 Styling with Markdown is supported

Create pull request

Figure 1.14:

## 1.9 Conclusion

Congratulations! Navigating this assignment required flexibility using new tools and concepts before being fully acclimated – a vital engineering skill. As you practice and develop an understanding of what you just walked through, it may seem trivial (I challenge you to time yourself doing the same assignment halfway through the semester and see the difference). But these fundamentals enable the rest of the course material the way arithmetic is needed for calculus or differential equations.

## References

- [1] Tim Kelly. *Dreamliner carries its first passengers and Boeing’s hopes*. <https://www.top500.org/statistics/details/osfam/1>. Oct. 2011.
- [2] Randall Munroe. *Git*. <https://xkcd.com/1597/>. Oct. 2015.
- [3] Randall Munroe. *Git Commit*. <https://xkcd.com/1296/>. Nov. 2013.
- [4] TOP500. *Operating System Family / Linux*. <https://www.top500.org/statistics/details/osfam/1>. Jan. 2018.

## 2 | Operating System Essentials

*Reference:*

**Introduction to Scientific and Technical Computing:**

Chapter 1: Operating Systems Overview

### Purpose

Gain familiarity using a Bash shell on a Linux computer and common command line programs.

### Objectives

- Familiarity with basic shell commands
- Use pipes to combine command line programs
- Write a report on your progress in LaTeX
- Submit assignment with Git

### Background

A computer is a matrix of switches; like a standard light switch, each can be "on" or "off" at any time. In a computer everything is either "on" or "off" which we can represent as 1 or 0 in a system called *binary*. Normally this doesn't matter to the user, but will eventually lead to some surprising behavior that is explained by this two-state system. We will see the consequences of this in future labs.

The architecture of taking this network of switches and imbuing abstract concepts like files and folders is layered like a fossil record; as new technologies were developed they were overlaid their predecessors rather than repeating the existing work. The layer on top of this stack is the *Operating System* (OS). The OS runs the physical mesh of switches that is the computer hardware and presents an interface to the world, a vocabulary of requests and commands it can perform. The OS can be asked to read a file, use a printer, connect to the internet and more — this common set of features greatly simplifies programming, and enables interoperability between programs and more.

The operating system we are using in this lab is called 'Linux', and follows a pattern of behavior called the Single UNIX Specification. This pattern describes (among other things)

what a file and directory are, how a command line works, what commands the command line must have at a minimum and what services the operating system will provide for programs. These standards are why your skills with the Linux servers here can be used with other operating systems like BSD and even MacOS (but not Windows).

We don't ask the OS for these services directly, as users of the system we run programs that do this for us. As we mentioned last time, the shell is a program that takes input on a command line and launches programs for us.

## 2.1 Standardized Syntax

As the course progresses, the lab manual will become terser and more technical — dropping the familiar tone and walkthrough style of early labs. The goal is to ease students into comfort and literacy with technical documentation and encourage self-directed research to accomplish tasks by the end of the course. In furtherance of this goal, each chapter will introduce standardized syntax and formatting used until it resembles technical documentation.

For this chapter, there are 2 important formalizations:

1. Text set in the `typewriter` indicates either a command or part of a command. It doesn't necessarily indicate that the text should be typed at that moment, simply that the text could be used on the command line.
2. The use of angle brackets in such text indicates replacement. For example `cd /Documents/<filename>` refers to the command "cd" followed by the path the user's myfiles directory. The student is expected to replace "<filename>" with their own username.

## 2.2 Simple Commands

The primary way to interact with the Linux OS is through issuing commands. You used some commands (`cd`, `git`, `mkdir`, etc.) in the last lab. As a warm up, here are some basic commands for getting your bearing on a system:

- Who Am I?

The `whoami` command lets you see who you are logged in as. Type in `whoami` in the command line and hit Enter. The output should be your assigned login id.

A screenshot of a terminal window with a title bar showing three colored buttons (red, yellow, green) and the text 'dureja@linuxremote1:~'. The terminal content shows the command '[dureja@linuxremote1 ~]\$ whoami' followed by the output 'dureja' on the next line. The prompt '[dureja@linuxremote1 ~]\$' is followed by a black cursor block. A small icon of a notepad and pencil is visible in the bottom right corner of the terminal window.

```
[dureja@linuxremote1 ~]$ whoami
dureja
[dureja@linuxremote1 ~]$
```

- Where Am I?

In the last lab we were continuously creating new directories and moving around directories. It is not unusual to lose track of the directory we are working with. The `pwd` command lets you see the current working directory. Type in `pwd` in the command line and hit Enter.



```
dureja@linuxremote1:~  
[dureja@linuxremote1 ~]$ pwd  
/home/dureja  
[dureja@linuxremote1 ~]$
```

- Who Else is Logged In? On a shared computer, multiple users can be active at the same time. The `who` command lets you see the names (ISU-NetID in our case) of other users logged in. Type in `who` in the command line and hit Enter. The output will be a list of users logged in.

```
dureja@linuxremote1:~  
[dureja@linuxremote1 ~]$ who  
jedicker pts/0      2018-01-12 15:36 (ents-rhel7.engineering.iastate.edu)  
dureja   pts/1      2018-01-14 18:10 (dureja.vpn.iastate.edu)  
[dureja@linuxremote1 ~]$
```

- What Date and Time Is It?

Since we are dealing with a black-and-white command line interface for our labs, it can be difficult to keep track of time without a clock on our computer screens. Linux's `date` utility shows us the current date and time. Type in `date` in the command line and press Enter. No excuses now for a late submission.

```
dureja@linuxremote1:~  
[dureja@linuxremote1 ~]$ date  
Sun Jan 14 18:32:13 CST 2018  
[dureja@linuxremote1 ~]$
```

To keep track of the number of days to submission deadline, Linux also offers a `cal` command that display a full-month calendar for the current month on the command line. Type `cal` in the command line and press Enter.

```
dureja@linuxremote1:~  
[dureja@linuxremote1 ~]$ cal  
January 2018  
Su Mo Tu We Th Fr Sa  
1  2  3  4  5  6  
7  8  9 10 11 12 13  
14 15 16 17 18 19 20  
21 22 23 24 25 26 27  
28 29 30 31  
[dureja@linuxremote1 ~]$
```

## 2.3 The Most Important Command

Thus far, most commands have been either one or two words. On a command line, the first word is the name of the program and all other text is passed to the program as *arguments* or *parameters*. For example when we use the command `cd` we have given one argument, which directory we want to change to. How do we know what additional information to give a program?

Some programs will give you a hint on what input they expect if you call them with no (or non-sensical) arguments. However, some commands like the ones you just learned in Section 2.2 are perfectly happy with no arguments! What then? Even with `cd` will work without arguments (`cd` without arguments will change to the local home directory).

### 2.3.1 Ask, and You Shall Receive

The most important command to know is `man` (short for manual) which will display a manual page of the command passed as an argument. For example, to learn more about the `cal` command used in the last section, try running `man cal`. Manual pages are opened with the `less` command – as we used to read files last lab. So you can navigate the file by using the arrow keys to scroll and press "q" to exit.

### 2.3.2 RTFM: Read the Fabulous Manual

These pages have a particular format that will become familiar with use, but a few highlights:

- In the synopsis section, brackets indicate the enclosed bits are optional. The braces are not typed.
- Most pages will have examples of common uses toward the end of the page

Look at the manual pages for a few of the commands you know, see if you can learn a new feature of a command you already know. Try `man man` to get the manual of the manual viewer, or even better `man intro` brings up a brief overview of using a command line!

*Note:*

If you ask for `man cd`, the manual page will be called "BASH\_BUILTINS" and contain mini-manuals for several commands. These are commands built-into the shell itself, we will do much more with these next week.

### 2.3.3 Keeping Your Options Open

Many of the manual pages will reference "options" and have a list of them all beginning with hyphens. We will cover how to use these options in Section 2.4.2, but one is worth pointing out: `-h`.

Most commands will accept `-h` or `--help` as arguments and print a short hint to their usage in response. Try it with `cal -h` to get a brief review of potential options. As you try this with different commands, you will find that some accept both, others will only take either the "short form" of the option (`-h`) or the "long form" (`--help`), a few will accept neither. Just experiment and see what gives you what you want!

*Tip:*

If you need help with a command, or just want to know if it can do more — try `foo -h`, `foo --help` or `man foo` until you get what you need!

More advanced ways for using the help system will be covered in the next lab.

### 2.3.4 System Information

If you are given a computer at a job, a common first task is to check the version of the OS (Windows 7/10/XP, MacOS?) and the hardware specs, i.e. CPU speed, RAM capacity, etc. If you only have an SSH connection to a machine (like our linux remote servers) how do you get this information?

Try the following commands (use any additional parameters you find helpful):

- `uname`
- `lscpu`
- `lsblk`

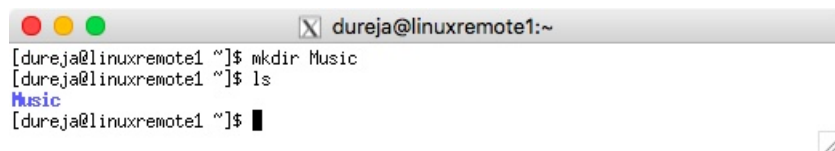
**NOTE:** Remember this is a shared server, so your access might be restricted to system files and other users' information.

## 2.4 Navigating Linux

The main purpose of an operating system is to help you navigate the stored files and directories. Linux has several commands that help you manage your stored data. Some of them (`cd`, `mkdir`, `rm`) were introduced in the last lab, and are reproduced here with their advanced usage for completeness. NOTE: We use the words 'directory' and 'folder' interchangeably. For better understanding, please type the commands in your command line and see what happens. Your output should match (or be similar) to the one shown in the figures.

### 2.4.1 Creating New Directories

- `mkdir <directory_name>`: In this command, `<directory_name>` is the name of the directory to be created. The new directory is created in the current active directory (`pwd`). The created directory is initially empty.



```
dureja@linuxremote1:~  
[dureja@linuxremote1 ~]$ mkdir Music  
[dureja@linuxremote1 ~]$ ls  
Music  
[dureja@linuxremote1 ~]$
```

- `mkdir <directory_name_1> <directory_name_2> <directory_name_3>`: In this command, `<directory_name_1>`, `<directory_name_2>`, and `<directory_name_3>` are the names of directories to be created in the current active directory (`pwd`). This command is used to create more than one directory. To verify, use the `ls` command. The created directories are initially empty.

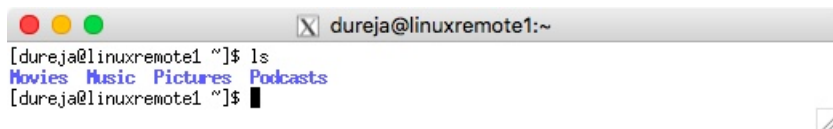


```
dureja@linuxremote1:~  
[dureja@linuxremote1 ~]$ mkdir Pictures Podcasts Movies  
[dureja@linuxremote1 ~]$ ls  
Movies Music Pictures Podcasts  
[dureja@linuxremote1 ~]$
```

## 2.4.2 Viewing Content In A Directory

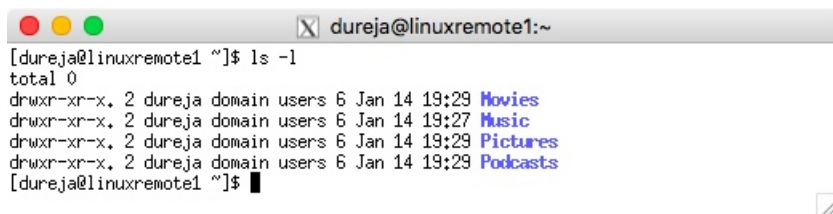
We have been using the `ls` command for a while now. It shows the contents of the current active directory. However, it also accepts several options using '-'. Some of the most commonly used are:

- `ls`: View the contents of a directory and show them in a multi-column layout.



```
dureja@linuxremote1:~  
[dureja@linuxremote1 ~]$ ls  
Movies Music Pictures Podcasts  
[dureja@linuxremote1 ~]$
```

- `ls -l`: The `-l` stands for "long." View the contents of the directory in a multi-row layout with additional information about directory creation, ownership, etc. More on this later.



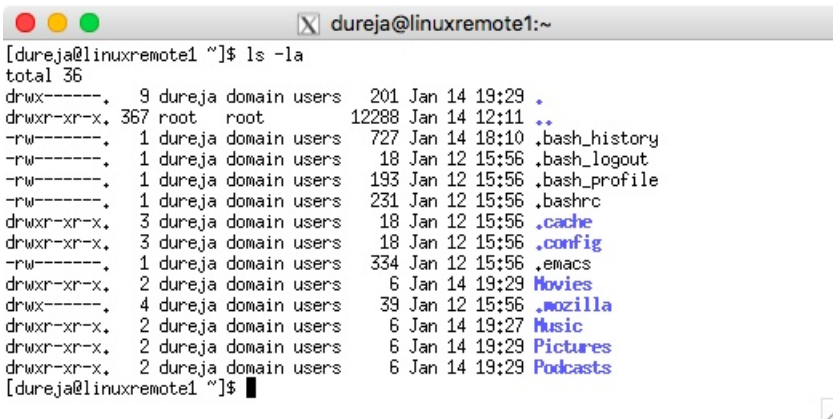
```
dureja@linuxremote1:~  
[dureja@linuxremote1 ~]$ ls -l  
total 0  
drwxr-xr-x. 2 dureja domain users 6 Jan 14 19:29 Movies  
drwxr-xr-x. 2 dureja domain users 6 Jan 14 19:27 Music  
drwxr-xr-x. 2 dureja domain users 6 Jan 14 19:29 Pictures  
drwxr-xr-x. 2 dureja domain users 6 Jan 14 19:29 Podcasts  
[dureja@linuxremote1 ~]$
```

- `ls -a`: The `-a` stands for "all." View all files in the directory, even the hidden files, in a multi-column layout. Hidden files in Linux start with '.', and are not shown when simply using the `ls` command.



```
dureja@linuxremote1:~  
[dureja@linuxremote1 ~]$ ls -a  
. .bash_history .bash_profile .cache .emacs .mozilla Pictures  
.. .bash_logout .bashrc .config Movies Music Podcasts  
[dureja@linuxremote1 ~]$
```

- `ls -la`: View all files in the directory, even the hidden files, in multi-row layout with additional information.

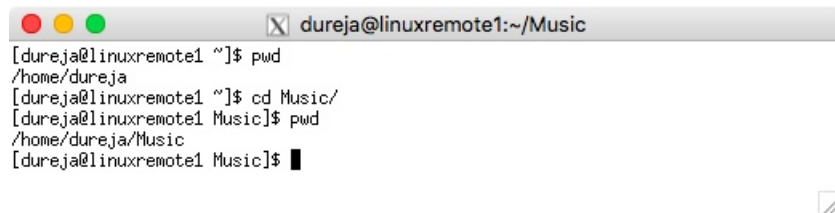


```
dureja@linuxremote1:~  
[dureja@linuxremote1 ~]$ ls -la  
total 36  
drwx-----. 9 dureja domain users 201 Jan 14 19:29 .  
drwxr-xr-x. 367 root root 12288 Jan 14 12:11 ..  
-rw-----. 1 dureja domain users 727 Jan 14 18:10 .bash_history  
-rw-----. 1 dureja domain users 18 Jan 12 15:56 .bash_logout  
-rw-----. 1 dureja domain users 193 Jan 12 15:56 .bash_profile  
-rw-----. 1 dureja domain users 231 Jan 12 15:56 .bashrc  
drwxr-xr-x. 3 dureja domain users 18 Jan 12 15:56 .cache  
drwxr-xr-x. 3 dureja domain users 18 Jan 12 15:56 .config  
-rw-----. 1 dureja domain users 334 Jan 12 15:56 .emacs  
drwxr-xr-x. 2 dureja domain users 6 Jan 14 19:29 Movies  
drwx-----. 4 dureja domain users 39 Jan 12 15:56 .mozilla  
drwxr-xr-x. 2 dureja domain users 6 Jan 14 19:27 Music  
drwxr-xr-x. 2 dureja domain users 6 Jan 14 19:29 Pictures  
drwxr-xr-x. 2 dureja domain users 6 Jan 14 19:29 Podcasts  
[dureja@linuxremote1 ~]$
```

### 2.4.3 Changing Directories

The most common operating system procedure is to move around folders and view their contents, and create new folders. Linux has several commands that help us do that. You are already aware of a few from the last lab.

- `cd <directory_name>`: In this command, `<directory_name>` is the name of a directory in the current active directory. The command changes the current active directory to `directory_name`. The change can be verified by running `pwd`.



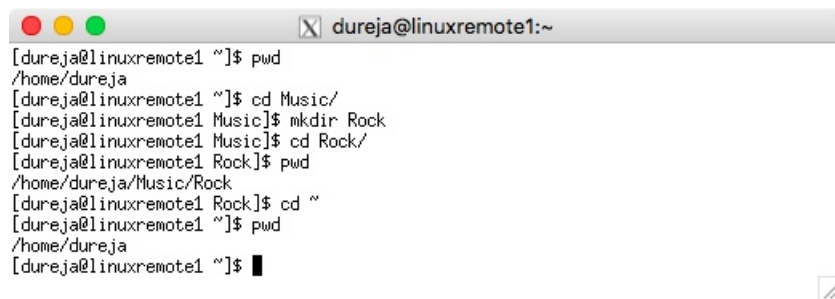
```
dureja@linuxremote1:~/Music
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$ cd Music/
[dureja@linuxremote1 Music]$ pwd
/home/dureja/Music
[dureja@linuxremote1 Music]$
```

- `cd ..`: The `..` operator in Linux is a shortcut for the directory containing the current active directory. Therefore, `cd ..` takes you to the directory containing the directory returned by `pwd`. This is often called "moving up" a directory.



```
dureja@linuxremote1:~
[dureja@linuxremote1 Music]$ pwd
/home/dureja/Music
[dureja@linuxremote1 Music]$ cd ..
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$
```

- `cd ~`: The `~` is a special operator and denotes the **HOME** directory. This directory is the main directory for the current active user, and is the directory that is active when you login to the `linuxremote` servers using the Remote Desktop Connection.



```
dureja@linuxremote1:~
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$ cd Music/
[dureja@linuxremote1 Music]$ mkdir Rock
[dureja@linuxremote1 Music]$ cd Rock/
[dureja@linuxremote1 Rock]$ pwd
/home/dureja/Music/Rock
[dureja@linuxremote1 Rock]$ cd ~
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$
```

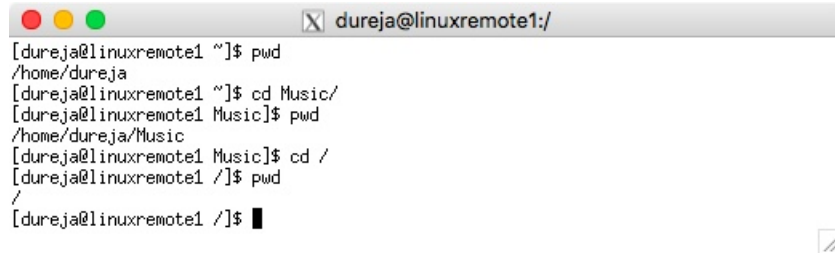
*Warning:*

In the past, since we needed to worry about syncing our work across all school computers, we needed to be in a special directory that would carry over across computers. Because everyone is assigned their own virtual machine however, we don't need to worry about this.

- `cd /`: The `/` operator is also a special operator and denotes the **root** directory. This directory represents the entirety of the computer, with all files and devices connected to the computer found as subdirectories of root. This provides a starting point for all paths in the system, and is somewhat of an abstract concept.

For Windows users, think of root as similar to the "My Computer" view.

The textbook explains the special subdirectories under root, for those interested.

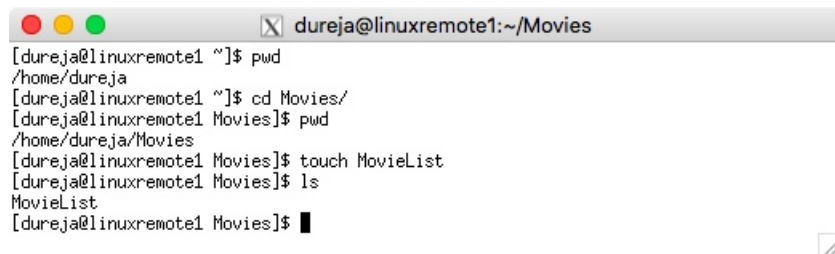


```
dureja@linuxremote1:/$ pwd
/home/dureja
[dureja@linuxremote1 ~]$ cd Music/
[dureja@linuxremote1 Music]$ pwd
/home/dureja/Music
[dureja@linuxremote1 Music]$ cd /
[dureja@linuxremote1 /]$ pwd
/
[dureja@linuxremote1 /]$
```

#### 2.4.4 Creating A File

A computer with empty directories is not of much use. Linux has several commands/programs to create a new file. One such program, **nano** was used in the last lab. To recall, `nano foo.txt` opens up an editor for adding contents to the file `foo.txt` that is saved when **nano** is exited ( `Ctrl+O`, `enter`, and `ctrl+X`). Linux also allows the user to create empty files. These empty files can then be opened with an editor, like **nano**, to add content.

- `touch <filename>`: In this command, `<filename>` is the name of the file to be created. The new file is created in the current working directory.



```
dureja@linuxremote1:~/Movies
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$ cd Movies/
[dureja@linuxremote1 Movies]$ pwd
/home/dureja/Movies
[dureja@linuxremote1 Movies]$ touch MovieList
[dureja@linuxremote1 Movies]$ ls
MovieList
[dureja@linuxremote1 Movies]$
```

Fire up **nano** on this file by typing `nano MovieList` and press `Enter`. Type in five movie names and their year in the editor, and exit (`Ctrl+X` and then press `Y`). For example, type in:

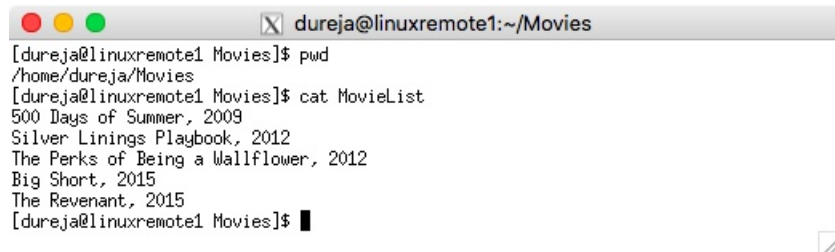
```
500 Days of Summer, 2009
Silver Linings Playbook, 2012
The Perks of Being a Wallflower, 2012
Big Short, 2015
The Revenant, 2015
```

#### 2.4.5 Viewing A File

Now that we have created a file, one way to view it is to open it again using an editor (**nano**), and exit without making any changes. However, this method is prone to errors since a user

might accidentally press a few key strokes and edit the contents of the file. Linux provides several commands to view a file without the possibility of editing it.

- **cat <filename>**: In this command, **filename** is the name of the file you want to view. The **cat** command prints the entire contents of the file on the screen. If the file is long, it might be required to scroll up to view the entire contents.

A terminal window titled 'dureja@linuxremote1:~/Movies'. The user has entered 'pwd' and 'cat MovieList'. The output of 'cat MovieList' is displayed on the screen.

```
[dureja@linuxremote1 Movies]$ pwd
/home/dureja/Movies
[dureja@linuxremote1 Movies]$ cat MovieList
500 Days of Summer, 2009
Silver Linings Playbook, 2012
The Perks of Being a Wallflower, 2012
Big Short, 2015
The Revenant, 2015
[dureja@linuxremote1 Movies]$
```

- **more <filename>**: In this command, **filename** is the name of a file. The **more** command is used to view contents of the file that fit in the screen. To scroll forward through the contents of the file, press Space multiple times, or until the end-of-file is reached. Important: **more** doesn't allow to scroll backward in a file! Press Q to exit.

A terminal window titled 'dureja@linuxremote1:~'. The user has entered 'pwd' and 'more .bash\_history'. The output of 'more .bash\_history' is displayed on the screen.

```
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$ more .bash_history
```

Type **more ~/.bash\_history** and press Enter. The output will be a history of commands you have entered in the command line terminal.

A terminal window titled 'dureja@linuxremote1:~'. The user has entered 'more .bash\_history'. The output of 'more .bash\_history' is displayed on the screen.

```
xclock
clock
time
xclock
logout
uname
whoami
uname -a
uname -h
--More--(8%)
```

Hitting Space scrolls you through the command history. Notice the percentage of file viewed.

A terminal window titled 'dureja@linuxremote1:~'. The user has entered 'more .bash\_history'. The output of 'more .bash\_history' is displayed on the screen.

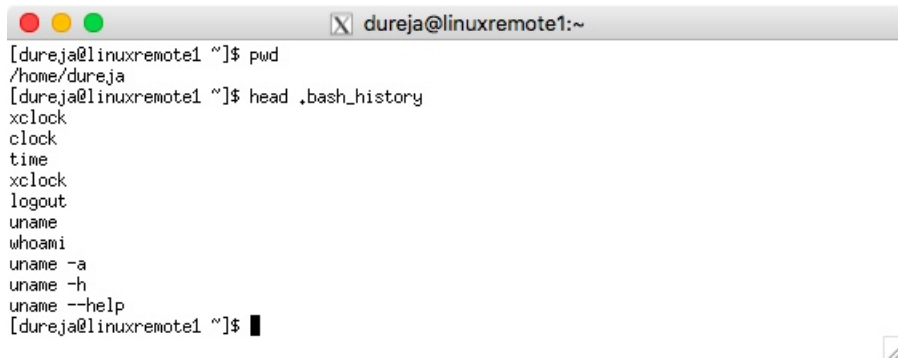
```
last
id
id -g
id -gn
id -u
uptime
last reboot
who
pwd
--More--(70%)
```

- **less <filename>**: In this command, **filename** is the name of a file. The **less** command is like the **more** command, except that it allows both forward and backward scrolling of the file. The arrow keys can be used to scroll. Press Q to exit.



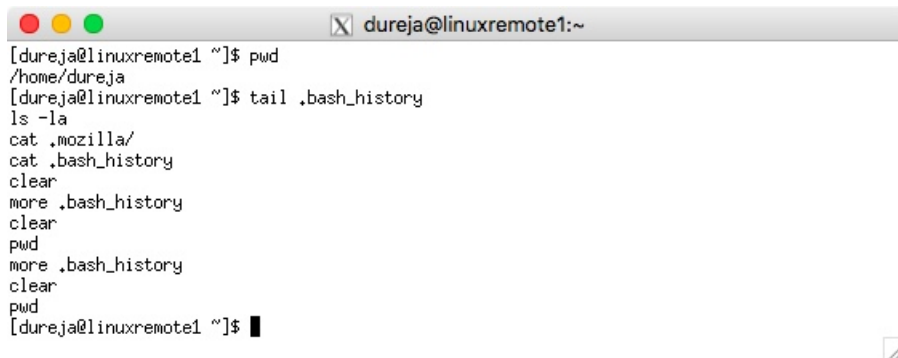
```
dureja@linuxremote1:~
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$ less .bash_history
```

- **head <filename>**: In this command, **filename** is the name of a file. The **head** command shows the first few lines of the file you want to view.



```
dureja@linuxremote1:~
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$ head .bash_history
xclock
clock
time
xclock
logout
uname
whoami
uname -a
uname -h
uname --help
[dureja@linuxremote1 ~]$
```

- **tail <filename>**: In this command, **filename** is the name of a file. Opposite to **head**, the **tail** command shows the last few lines of the file you want to view.



```
dureja@linuxremote1:~
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$ tail .bash_history
ls -la
cat .mozilla/
cat .bash_history
clear
more .bash_history
clear
pwd
more .bash_history
clear
pwd
[dureja@linuxremote1 ~]$
```

## 2.4.6 Moving Files and Directories

Once files and directories have been created, it is often required to copy files from one directory to another, copy entire directories, or move entire directory contents. Linux has several commands that help us move data around the computer. These commands are the equivalent of Cut-Copy-Paste operations.

- **cp <filename> <dest\_directory>**: The copy command. In this command, **<filename>** is the file to copy and **<dest\_directory>** is the destination directory the file needs to be copied to.



```
dureja@linuxremote1:~/Pictures
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$ cp Movies/MovieList Pictures/
[dureja@linuxremote1 ~]$ cd Pictures/
[dureja@linuxremote1 Pictures]$ ls
MovieList
[dureja@linuxremote1 Pictures]$
```

- `cp -r <src_directory> <dest_directory>`: A recursive option for copy. The `-r` option with `cp` is used to copy a directory to another directory. In this command, `<src_directory>` is the source directory to be copied, and `<dest_directory>` is the destination directory. Try copying directories without the use of `-r` option and see what happens.

```
dureja@linuxremote1:~/Podcasts
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$ cp -r Movies/ Podcasts/
[dureja@linuxremote1 ~]$ ls
Movies Music Pictures Podcasts
[dureja@linuxremote1 ~]$ cd Podcasts/
[dureja@linuxremote1 Podcasts]$ ls
Movies
[dureja@linuxremote1 Podcasts]$
```

- `mv <filename> <dest_directory>`: The move command. The `cp` command makes a copy of the given file or folder, leaving the original untouched. However, sometimes it is required to copy an entire file or folder, and delete the original copy. The `mv` command is used for this purpose. In this command, `<filename>` is the file to move and `<dest_directory>` is the destination directory.

```
dureja@linuxremote1:~/Pictures
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$ cd Movies/
[dureja@linuxremote1 Movies]$ ls
MovieList
[dureja@linuxremote1 Movies]$ mv MovieList ../Pictures/
[dureja@linuxremote1 Movies]$ ls
[dureja@linuxremote1 Movies]$ cd ../Pictures/
[dureja@linuxremote1 Pictures]$ ls
MovieList
[dureja@linuxremote1 Pictures]$
```

- `mv <src_directory> <dest_directory>`: The `mv` command can be used to move complete directories as well. In this command, `<src_directory>` is the source directory to be copied, and `<dest_directory>` is the destination destination.

```
dureja@linuxremote1:~/Podcasts
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$ ls
Movies Music Pictures Podcasts
[dureja@linuxremote1 ~]$ cd Podcasts/
[dureja@linuxremote1 Podcasts]$ ls
Movies
[dureja@linuxremote1 Podcasts]$ cd ..
[dureja@linuxremote1 ~]$ mv Pictures/ Podcasts/
[dureja@linuxremote1 ~]$ ls
Movies Music Podcasts
[dureja@linuxremote1 ~]$ cd Podcasts/
[dureja@linuxremote1 Podcasts]$ ls
Movies Pictures
[dureja@linuxremote1 Podcasts]$ █
```

## 2.4.7 Removing Files and Directories

If users can create new files and folders, they can also delete it. Recall from the last lab that Linux offers the `rm` command to remove unwanted data from the computer. Files once removed using `rm` cannot be retrieved. There is no Trash or Recycle bin where deleted files go to after using `rm`. The file are erased permanently from the computer storage. **Use with caution.**

- `rm <filename>`: The remove command. In this command, `<filename>` is the file to remove.

```
dureja@linuxremote1:~/Movies
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$ ls
Movies Music Podcasts
[dureja@linuxremote1 ~]$ cd Movies/
[dureja@linuxremote1 Movies]$ ls
MovieList
[dureja@linuxremote1 Movies]$ rm MovieList
[dureja@linuxremote1 Movies]$ ls
[dureja@linuxremote1 Movies]$ █
```

- `rm -r <directory>`: The recursive remove option. In this command, `<directory>` is the directory to remove. The `-r` option with `rm` is used to remove entire directories. Try removing a directory without the `-r` options. Be careful, this will immediately delete the directory and all it contains.

```
dureja@linuxremote1:~/Movies
[dureja@linuxremote1 ~]$ pwd
/home/dureja
[dureja@linuxremote1 ~]$ ls
Movies Music Podcasts
[dureja@linuxremote1 ~]$ cd Movies/
[dureja@linuxremote1 Movies]$ ls
MovieList
[dureja@linuxremote1 Movies]$ rm MovieList
[dureja@linuxremote1 Movies]$ ls
[dureja@linuxremote1 Movies]$ █
```

## 2.5 The UNIX philosophy

All of the programs introduced so far are quite limited in scope, to the point of seeming almost comically limited in capability. GUI-based file explorers like Windows Explorer, MacOS Finder or even Linux GUIs (yes, they exist) like GNOME all have the capability to perform the functions of all the commands you've been introduced to. So why all the micro-programs? The answer is about perspective.

The designers of the original UNIX system had a core belief: instead of thinking of the programs as applications, think of them as building blocks. Rather than predict every possible task the user could want to perform, each little program should "do one thing well" and be easily combined with other programs. One of the advantages of the command line, is that all input is text and all output is text. We can then use the output of one command as the input to another!

To chain these commands together, we use a pipe. The pipe symbol `|` (the shifted form of backslash on the keyboard) is placed between two commands to send the output of one into the input of another. The next section explores the uses of shell pipes.

## 2.6 The Chain Gang

### 2.6.1 Less is More

Some command produce too much output for the screen to display, and many will automatically use `less` to display one screen-full at a time. If the command doesn't provide this feature, we can do so ourselves with pipes. The command `ps` is used to report on the process state, showing all of the jobs the OS is currently working on. Try running `ps aux` on the server.

Way too much information, we can't tell what's going on at all! So let's pipe it through `less`:

*Note:*

If you press the up arrow key, the shell will fill in the last command you ran. This can save lots of typing if you need to re-run a command with a slightly different option.

Run `ps aux | less` where the pipe character separates the two commands. Much easier to handle!

### 2.6.2 "grep"ing Text

One of the most frequently used command line programs is `grep`. `grep` is like a filter, and only allows text through if it matches a pattern. One of the most common ways it can be used is like a search function for a file or input.

Try this, using `ps aux` to list running processes, filter them for processes you are running

like this: `ps aux | grep <username>`

Notice how only lines that featured your username were returned.

### 2.6.3 Finding Files and Directories

Grep works very well for searching within files but to search for a file, we use the `find` command. The usage of the command is left as an exercise to the student (Hint: toward the end of the incredibly detailed manual page, there are some helpful examples for 80% of use-cases) and will be required for one of the graded tasks.

### 2.6.4 Wildcards

Useful both in `grep` and `find` expressions, the asterisk (\*) can be used to indicate "some characters here, but I don't care what" as in `find ~ -name "*.txt"` which could be read as: "Find all the files in my home directory who's name ends in the ".txt" expression.

## 2.7 Exercises

Begin your lab submission by:

1. Clicking the GitHub Classroom link in the lab email
2. Copying the clone URL from the green "Clone or download" button
3. Moving to your Desktop (hint: type `ls` to see where you're at)
4. Running `git clone` followed by the URL you copied from the web site
5. Using `cd` to move into your new lab directory (lab-2-netid)
6. Follow the directions in the README.md file to checkout a branch named "develop"

Remember to use `git add` and `git commit -m "Your message here"` after you finish each exercise and open a pull request when you are finished with the lab.

#### *Warning:*

These are extension exercises — you will have to combine the knowledge you have been given in a new way and do some research (read: googling) to complete these exercises. A practiced Linux user can do this entire section in minutes, but it might take you much longer. Do not leave this until the last minute, because you don't know how long this will take you — you have been warned.

### 2.7.1 Grading

There are 5 exercises in this lab. Each is worth 10 points: 5 for creating a valid submission (a file of the proper name, with the requested type of information such as a number, name or path) and 5 for the correct answer. Your report will have one section per exercise (see next section for more information on the report format) and each of those sections is worth an additional 10 points for a total of 100 for lab 2. A note of caution - if we cannot build your

report, you can lose half the points of the lab! We are supplying a script to help ensure your lab follows the submission requirements for the autograder, including building the report, which will be detailed at the end of the lab instructions.

### 2.7.2 Report

In the report folder you will find a starter "main.tex" file. It is much simpler than last weeks's example; we will be increasing the complexity of our LaTeX throughout the year as you gain confidence.

To start, modify the header info (title, date, and author) at the top of the document, then run `pdflatex main.tex` from the reports directory. Verify that the pdf builds before continuing.

For each of the following 5 exercises, your report will contain:

1. Executive summary: a one sentence description of how you completed the exercise
2. Command: What you typed in the terminal to complete the lab
3. Deciphering output: How you read the output from the computer to get the information want
4. Options Used: A list with descriptions of the flags, options and arguments you added to the commands
5. Sources of information: A list of all the people, places, and programs you used to get assistance to solve the lab

The given "main.tex" file contains a partially filled "mad-libs" style report for the first few exercises. You will be responsible for creating a section with all the above components for each exercise you do. It is much easier to do this as you work on the respective exercise. If you have trouble understanding what is expected in the report, contact a TA.

### 2.7.3 Exercises

**Exercise 1.** (10 Points) Use the `ls` command to find the latest modified/changed file in your lab directory and write the file name to a new file named `latest-file.info`. There is a `ls` option that will assist you with this. Make sure the only content of the `latest-file.info` file is the name - no other extraneous text. Remember that this is graded by a computer. When you are satisfied with your answer, commit your work.

**Exercise 2.** (10 Points) Use the command `wc` to count how many objects are in your lab directory. Write the number to a file called `count.info` (Hint: You will need to use the `ls` and `wc` commands together and both will need an option to further specify what you want them to do) Make sure the only content of the `count.info` file is the number of items in your lab directory at the time you ran the command. It is very easy to have an incorrect count if you don't properly specify your options. When you are confident with your answer, commit your work.

**Exercise 3.** (10 Points) In Windows, if you want to run a executable file (program), you just find the file and double-click it. After that the Windows system will start to run the file.

However, the commands (executable files) you tried so far in Linux haven't required you to know their location – only the name is needed to execute them. The question is, where is the `ls` program installed? Use the command `whereis` to locate the binary (program executable) of the program `ls`. Write the full path of the directory to the file `ls.info` in your lab directory. Be careful, by default `whereis` gives you more info than you need, make sure to pick the right path. (Hint: A full path starts with a "/" and ends with the name of the file in question, so in this case it should end in "ls") Make sure the only text in this file is the path to the `ls` binary, then commit your work.

If you `cd` into the directory hold the "ls" file, you can find many other commands that can be executed in Linux. Linux pre-defines some directories as the default path where it can search for the command. So unless you specify the location of the program to run, the system will search in these directories and will return error if it cannot find the program under these directories. We will explore this more in future labs.

**Exercise 4.** (30 Points) Try to find all the files with the name "patents.txt" in the whole Linux file systems. Write the full paths to a file named `find.info` in your lab directory. Do not include lines indicating permission errors, only valid, full paths. (Hint: use the `find` command to search from the root directory "/") Do not manually sort through the permission denied errors, there are many ways of making the computer do that work for you. The number of files will be different depending on the server you use, that is ok. Just ensure that each line of the resulting `find.info` file is a valid path to a "patents.txt" file. You can verify this by trying to read each path with `less`. When you have completed this exercise, commit your work to git.

**Exercise 5.** (10 Points) You might be familiar with the Task Manager in Windows, find the comparable command in Linux that lists the information of running programs, CPU and memory usage and updates live. Write the command name to the file `taskmanager.info` in your lab directory. This file should only contain the name of this command and nothing else. Be careful when googling, there are many commands that statically list parts of the requested information, but do not display all the request information or update constantly until stopped. (Hint: it isn't `ps`) When you are done with this exercise, commit your changes.

#### 2.7.4 Lab work submission

When you are finished with your lab, follow this procedure to turn in your work for grading:

1. Ensure all 5 exercises have a correctly named file in your lab directory
2. Ensure each of the 5 info files contain only the requested information and nothing more
3. Use `git status` to verify all the work you want to be graded has been committed to the develop branch
4. Use `git push` to update GitHub with all your changes
5. Run the provided "check.sh" script to check your lab submission, correct any problems it lists. To run the check script:

- From your lab directory run `chmod +x ./check.sh` to tell the computer to treat the file like a program
  - Run `./check.sh` (notice the leading period before the slash) to check your lab
6. Commit and push and final changes and open a pull request on GitHub
  7. **DO NOT** click the "close and merge" button - your PR should remain open for the autograder to find
  8. Run the `check.sh` script one more time and verify it sees your PR
- Congratulations, you are 14.28571429% of the way though 361!

## 3 | Scripting with Bash

*Reference:*

**Introduction to Scientific and Technical Computing:**

Chapter 4: Introduction to Bash Scripting

**Optional Reference Material:**

[Learning the bash Shell](#) (Available free from ISU library)

### Background

In the last lab, we introduced some basic commands for the Bash shell. We entered a command (with options), hit Enter, and waited for the command to finish executing. What if we had a million commands that were executed? Should we do it manually by hitting enter after each command? This is where Bash scripting comes into play.

In addition to the interactive mode where the user types one command at a time, Bash also has the ability to run an entire script of commands, known as the "*Bash shell script*". A script contains a list of command, or it might contain a single command. In addition to commands, scripts also support functions, loops, if-else statements (recall the MATLAB language). In this lab we walk through some examples that use a script. Before we actually get into writing scripts, we introduce the regular expression: a powerful way to process information that forms an integral part of Linux.

After logging into the Linux servers and while in the `$Desktop` directory, grab the file for the example we will be using. To do that use the command:

```
wget https://git.io/vNaF7
```

(`wget` is a command line utility to download stuff from the internet. This command downloads the file `vNaF7` from `https://git.io`.) As the next step, rename the file.

```
mv vNaF7 top250.txt
```

Now you are all set for what is to follow! The file you just downloaded is a list of the Top 250 Movies of all time from IMDb. Your `$HOME` directory (`/home/<USERNAME>`) should contain



the file `top250.txt`.

## 3.1 Regular Expressions

Recall `grep` from the last lab. We used `grep` to search within text files. Another useful addition to `grep` is *Regular Expressions*, or *regexes* for short. Other commands also support regexes - more on that later.

A regular expression is a text string that describes a particular search pattern. `grep` uses the search pattern and matches it with each line, character, or character sequences in the file.

### 3.1.1 Literal Matches

The examples we have seen so far in the previous lab use literal matching with `grep`. In these examples, we specify the complete word or character to match in the file. For example: to find all “Star Wars” movies in the Top 250 list we use the command

```
dureja@linuxremotel1:~$ grep "Star Wars" top250.txt
0000000124 286926 8.8 Star Wars: Episode V - The Empire Strikes Back (1980)
0000000124 330059 8.7 Star Wars (1977)
0000001222 217458 8.3 Star Wars: Episode VI - Return of the Jedi (1983)
[dureja@linuxremotel1 ~]$
```

Similarly, try finding all movies in the list from the year “1995”. Your output should be similar to

```
dureja@linuxremotel1:~$ grep "1995" top250.txt
0000000123 275216 8.7 The Usual Suspects (1995)
0000000232 290808 8.6 Se7en (1995)
0000001123 248935 8.3 Braveheart (1995)
0000001222 149989 8.2 Heat (1995)
0000001222 159204 8.1 Toy Story (1995)
0000001222 112789 8.1 Casino (1995)
0000001221 175458 8.1 Twelve Monkeys (1995)
[dureja@linuxremotel1 ~]$
```

### 3.1.2 Matching Any Character

The period character (`.`) is used in regular expressions to mean that any single character, including whitespace, can exist at the specified location. For example, if we want to find all movies that contain the words “King”, “Kill”, “Kind”, or “Kick” in their titles we use

```
dureja@linuxremotel1:~$ grep "Ki.." top250.txt
0000000115 376934 8.8 The Lord of the Rings: The Return of the King (2003)
0000001223 85361 8.5 To Kill a Mockingbird (1962)
0000001222 240923 8.2 Kill Bill: Vol. 1 (2003)
0000001223 144014 8.2 The Lion King (1994)
0000001222 59491 8.1 Butch Cassidy and the Sundance Kid (1969)
0000001223 15141 8.1 The Kid (1921)
0000001321 25600 8.0 The Killing (1956)
0000001222 13589 8.0 Kind Hearts and Coronets (1949)
0000001222 190137 8.0 Kill Bill: Vol. 2 (2004)
0000001222 104681 8.0 Kick-Ass (2010)
0000001212 40569 8.0 King Kong (1933)
[dureja@linuxremotel1 ~]$
```

Can you think of a reason as to why movie names with “Kid” in their title are included in the result? How will you find all movies in the years 1990-1999?

### 3.1.3 Bracket Expressions

By placing a group of characters within brackets ("[" and "]"), we can specify that the character at that position can be any one character found within the bracket group. This means that if we wanted to find the lines that contain "too" or "two", we could specify those variations succinctly by using the pattern `t[wo]o`. For example, can you think of the regex to show movies names containing “King” or “Kind” in their titles?

### 3.1.4 Repeat Pattern Zero or More Times

Finally, one of the most commonly used meta-characters is the `"*"`, which means "repeat the previous character or expression zero or more times". It is most frequently used similarly to the last lab to indicate that we do not care about the rest of the string, as long as it contains the remaining elements.

#### *Useful links:*

1. (Recommended) For an excellent tutorial on basic and advanced usage of regular expressions: <https://ryantutorials.net/regular-expressions-tutorial/>
2. An online tool for writing regular expressions, with live updating and syntax hints: <https://regexr.com/>
3. Research tool on learning regular expression from examples: <http://regex.inginf.units.it/>. For a computer to learn a regular expression from examples is a computationally hard problem. Use this website with caution.
4. (Optional Reference) *Mastering Regular Expressions* (O'Reilly book), available for free from ISU at Safari Books Online.

## 3.2 Bash as a language

Bash, the shell you have been using, is a fully programming language. Because of this, concepts you are familiar with from use of MATLAB or other languages have direct analogs in Bash. We will dive deeper into these throughout the course as they become relevant, so in this lab we will just do a high-level overview.

Chapter 4 in your textbook is considered required reading – it's only 14 pages. While not necessary to solve this week's exercises, you will be expected to know the concepts introduced by the book in the future.

### 3.2.1 Variables

We have already made use of a variable, the `LAB2WORKDIR` from last lab was a shell variable. A shell variable is set with an equals sign and no space between the variable name and the value as in: `TESVAR="Hello, world!"` or `PI=3.14159`

To use a variable we add a dollar sign to the name — the shell will replace all instances of a variable name preceded by a dollar sign with their value. To see this in action, compare the output of the two following commands that make use of the variable `PATH` which is set by default in all shells:

- `echo PATH`
- `echo $PATH`

What difference does the dollar sign make?

### 3.2.2 Flow Control

For control flow and looping, bash offers the following constructs. These should be familiar from MATLAB and right now knowing what options are available is sufficient. When you feel you need to use one of these, you are encouraged to research usage.

- if-then-else blocks
- switch cases
- for loops
- while/until loops

### 3.2.3 Functions

Just like MATLAB, bash has functions – consult your textbook to learn more.

## 3.3 Exercises

Lab 2 focused on gaining skill parity with other operating systems (like Windows). From this lab onwards we will be developing capabilities not easily obtained in a GUI environment. These skills form the foundation for the power of command line systems and the reason they are often preferred in technical computing environments.

Begin your lab submission by:

1. Clicking the GitHub Classroom link in the lab email
2. Copying the clone URL from the green "Clone or download" button
3. Moving to your Desktop directory
4. Running `git clone` followed by the URL you copied from the web site
5. Using `cd` to move into your new lab directory (lab-3-netid)
6. Follow the directions in the README.md file to checkout a branch named "develop"

### Exercise 1. (10 Points)

This lab has exercises that require some extra data files — to reduce the size of the Git repository, they have to be downloaded separately. Use the following procedure to download these files, uncompress the download, and finally tell Git not to commit them to GitHub.

1. Use the command `wget` to download the lab data from:  
<http://temporallogic.org/courses/AERE361/data.tar.gz>
2. Unzip the compressed file with the command `tar` by running `tar xf data.tar.gz`
  - This tells the computer to "eXtract" the "File" named `data.tar.gz`

In your repository, you now have a directory named "data", with 2 child directories "simplic3-best1/" and "csv/". The first directory contains data from experiments as target files for this lab. The "csv/" directory contains one csv file which is also necessary for your assignment.

To tell git we don't want to track these files, we will create a file called `.gitignore` which lists patterns we are not interested in. Use a text-editor to create a file named `.gitignore` in your lab directory (notice the leading period), add the line `/data/` to this file and save it. Now run `git status`. Notice that the `data` directory is not listed as an untracked file? Git will now ignore it, and all the files inside. Using these ignore files is helpful in preventing things we don't care about from taking up space in our git repositories, like all the extra files generated by `pdflatex`.

Please do not commit the data folder, the data zip file, or any of the files contained within to Git. These dramatically increase the size of your repositories and slow down grading.

**Exercise 2.** (10 Points)

Please count how many files whose name contains ".trace" in the directory "data/simplic3-best1". Write this number to a file named `exercise-2.res` in your lab directory.

**Exercise 3.** (10 Points)

Find the files in "data/simplic3-best1" whose name contains "oski". Write these file names (not paths) to a file named `exercise-3.res` one per line.

Hint: One solution is to use the command "find". **NOTE: Only files names can be written to a file named `exercise-3.res`.** That is, if the "find" command returns you something like "data/simplic3-best1/oski2b5i.trace", only "oski2b5i.trace" is allowed to be written to the res file.

The constraint is strict, and we will match your `exercise-3.res` file verbatim to obtain the output that meets the above constraint, the "sed" command is recommended.

**Exercise 4.** (20 Points)

Please count the files in "data/simplic3-best1" which contains the string "Unsafe" and "Safe" (case sensitive) respectively. Write these two numbers – in order, one per line – to a file named `exercise-4.res`.

Hint: One solution is to use the combination of commands “grep” and “wc”. Recall that in Exercise 2, you know the number of files in “data/simplic3-best1/” whose name contains “.trace”. What is the relationship between this number and the two numbers required in Exercise 4?

**Exercise 5.** (30 Points)

Search within the files in the directory “data/simplic3-best1/”. If the file contains a line with the string “total\_time”, then extract the number from that line. Write all the file names and the numbers to a file named **exercise-5.res**, in the format “file-name: number”, one entry per line.

For example, the file “data/simplic3-best1/visemodel.log” has a line containing the string “total\_time”, and the number in that line is 0.037. So your **exercise-5.res** file will contain the line “visemodel: 0.037”. Notice the file name does not include the extension (i.e. “visemodel” not “visemodel.log”).

Point breakdown:

1. Correct file names (all files with “total\_time”) listed in file — 10 points
2. Correct numbers pulled from “total\_time” line — 10 points
3. Correct format (filename, colon, space, number, newline) — 10 points

Hint: One solution is to use the combination of commands “grep” and “sed”.

**Exercise 6.** (10 Points)

Please extract the first two columns of the file “data/csv/simplic3-best1.csv”. Note that in the csv (comma separated value) file, each column is separated by a comma. Write the output to a file named **exercise-6.res**.

Hint: One solution is to use the command “awk”.

**Exercise 7.** (10 Points)

Insert a new column before the original first column of the file “data/csv/simplic3-best1.csv”. Each number of the new column is the line number. So in the first line, the number is 1 in the new column, and the second line, the number is 2 in the new column... Write the new csv to a file named **exercise-7.res** and remember that columns should be separated by commas.

**Exercise 8.** (10 Points)

In “data/csv/simplic3-best1.csv”, the contents in each line are separated by commas. However, some fields are empty, which is written as two commas like this: “,”. Replace these empty fields with the string “empty”. The new field will then look like this “,empty,”. Write

the new csv to a file named `exercise-8.res`.

Example — The line

```
power2eq65536,Unknow,,,0
```

would become

```
power2eq65536,Unknow,empty,empty,empty,0
```

Warning: multiple empty columns can occur in sequence in a row – ensure that you correctly handle this case and have the proper number of columns at the end.

For example if part of the line is `,,,`, make sure you have three 'empty' columns at the end (as in `,empty,empty,empty,` and not `2` with a double comma (as in `,empty,,empty,`), this is the most common mistake.

### Exercise 9. (20 Points)

In your lab repository is a shell file called `intvari.sh` which is incomplete. This file takes two or more integers as input and should output the integer variance when the comments are replaced with appropriate bash commands.

The formula for variance is:

$$\frac{1}{N} \sum (x_i - \mu)^2$$

where  $N$  is the number of samples,  $x_i$  are the individual samples, and  $\mu$  is the mean (or the sum of the samples divided by  $N$ ). Since bash variables can only represent integers, both the mean and the variance end up floored - the decimal parts are dropped. In a future lab, we will learn ways around this limitation.

The point breakdown for this exercise is:

**Sample Check** Fix the guard at the start of the program that ensures it only runs with 2 or more inputs. 5 points.

**Mean** Fix the summation that computes the average value. 5 points.

**Variance** Now, compute the variance. You can base your solution on the loop structure used for the mean. 10 points.

To test your program, we have provided a set of random inputs in the file `samples.dat` for you to test. Use your knowledge of bash to present the contents of the file as the arguments (Hint: what programs do you know that can print out file contents?) and compare your answer to the correct value: 826

## 3.4 Report

Your report this week is worth 30 points.

### 3.4.1 Report Requirements

Your report should include the following three components, each worth 10 points:

- a title page with your name, course name, assignment name, etc.
- a personal command “cheat sheet” with at least ten commands and common options you used in the exercises or find useful
- any sources or resources you use while completing the exercises

For your command cheat sheet, add at least ten lines to the following table:

Command	Option	What this command/option combination does
kill	-9	Kills a <i>process</i> (a running instance of a program) identified from its PID
Your	input	here ...

The L<sup>A</sup>T<sub>E</sub>X source for this table is:

```
\begin{tabular}{|c|c|p{4.2in}|}  
\hline  
Command & Option & What this command/option combination does \\  
\hline  
\hline  
\texttt{kill} & \texttt{-9} & Kills a \emph{process} (a running instance  
of a program) identified from its PID \\  
\hline  
Your & input & here \ldots \\  
\hline  
\end{tabular}
```

Use your creativity for formatting your list of sources! This may be a simple itemized list, a footnote with a source for each entry in your table (Hint: `\footnote{Footnote Text}`), or something fancier, such as a bibliography compiled with Bib<sub>T</sub>E<sub>X</sub>. For urls, you should include the package `url` (Hint: `\usepackage{url}`); then the L<sup>A</sup>T<sub>E</sub>X for listing a source for the above table entry is:

```
\url{http://man7.org/linux/man-pages/man7/signal.7.html}
```

You may use the template given in Lab 2 or start fresh. Your report should be in a file named `main.tex` in a directory named `report` in your lab directory. While you are allowed to commit the pdf, it will not be used – your `.tex` source file will be graded. It is actually preferable not to include pdfs in git repos if the source to generate them is included as it dramatically inflates the size of the download.

Ensure your report builds without errors before submitting!

## 3.5 Lab work submission

When you are finished with your lab, follow this procedure to turn in your work for grading:

1. Ensure all 9 exercises have a correctly named file in your lab directory
2. Ensure each of the 7 res files contain only the requested information and nothing more
3. Ensure that your report is named correctly, is in the correct directory, and builds without errors
4. Use `git status` to verify all the work you want to be graded has been committed to the develop branch
5. Use `git push` to update GitHub with all your changes
6. Commit and push and final changes and open a pull request on GitHub
7. **DO NOT** click the "close and merge" button - your PR should remain open for the autograder to find

Congratulations, you are 21.4285714286% of the way though 361!



## 4 | Hello, C!

*Reference — C Programming Language, 2nd Edition:*

Chapter 1 – A Tutorial Introduction

Section 1.1 – Getting Started

Section 1.3 – The For Statement

Section 1.5 – Input and Output

Section 1.7 – Functions

Chapter 2 – Types, Operators, and Expressions

Chapter 3 – Control Flow

Chapter 7 – Input and Output

### Purpose

To gain familiarity with the basics of C programming for scientific computing, starting with the syntax and basic control structures for intelligent data analysis.

### Objectives

- Compile and execute C programs with `gcc` in Linux
- Define variable types
- Format input and output
- Construct loops and conditions to structure programs
- Write functions to aid in program organization and reduce code duplication
- Estimate program complexity: number of operations

### 4.1 Emacs and Editing

Until now, nano has been sufficient for the limited text editing necessary to complete the labs. However, programming in languages like C can benefit from features like syntax highlighting and paren-matching (both explained in section 4.1.3) not available in nano. To provide these ergonomic improvements in an environment that matches the demos in class, the next few sections will provide an introduction to using emacs for writing C.

### 4.1.1 Invoking Emacs

There are CLI and GUI interfaces to emacs and the guidance in this manual will apply to both. The major difference between the GUI and CLI versions for the purposes of this course is the accessibility of many common functions from mouse-driven drop-down menus in the GUI version. The same functionality can be called upon with command keys in both versions.

To launch the CLI version, connect to your Linux remote server via "Remote Desktop Connection" as usual, then run emacs.

To launch the GUI version, go to your Desktop on the Virtual Machine, click the "Applications" menu in the bottom left of the screen, and find Emacs under the "Programming" section.

To get a command line for compiling and running your code, right click on the LXTerminal icon on the desktop."

*Note:*

Try to logout from the bottom left menu in the GUI rather than just closing the RDP window. This frees up system resources for others to use.

### 4.1.2 Basic Emacs Commands

In this section, we introduce a list of basic Emacs commands useful for your work in this course. There are many more commands available and you are encouraged to seek out more advanced Emacs usage as you gain familiarity with the software.

In the list, the character "C" represents the "Ctrl" key on your keyboard, and "M" is the "Alt" key on your keyboard. Also, <RET> is the return key, <SPACE> is the space key, <LEFT> and <RIGHT> are the "←" and "→" keys on your keyboard.

1. start Emacs: type "emacs" in your terminal
2. open a file: C-x C-f filename
3. save the file: C-x C-s
4. exit Emacs: C-x C-c
5. undo changes: C-x u
6. copy/cut and paste
  - select the region you want to operate in: At the beginning of the region, press C-<SPACE>, and then move your cursor to the end of the region;
  - copy the region you selected: M-w;
  - cut the region you selected: C-w;
  - paste the region you selected: C-y.
7. delete lines: C-k (C-<SPACE>, C-w)

- delete one line: C-k will delete all rest of the line after your cursor. If your cursor is at the beginning of the line, C-k will delete the whole line;
- delete multiple lines: Move your cursor directly before the region you want to delete, press C-<SPACE>. Then Move your cursor after the region you want to delete, press C-w.

8. switch buffers. Emacs allows you to open multiple files, and each file corresponds to one buffer in the tool. If you open more than one file in Emacs, you may find the following commands are useful for you to switch between different buffers.

- Select or create a buffer named buffer: C-x b buffer <RET>
- select buffer in another window: C-x 4 b buffer <RET>
- select buffer in a separate frame: C-x 5 b buffer <RET>
- Select the previous buffer in the buffer list: C-x <LEFT>
- Select the next buffer in the buffer list: C-x <RIGHT>

### 4.1.3 Customizing Emacs for C programming

This section is optional! To make writing C easy in Emacs, you should customize Emacs environment before programming. This is the one-time job, but is server-specific (if you make these changes to your virtual machine, they will not be present on `linuxremote1`). As mentioned in the class, you can configure Emacs by editing the “.emacs” file under your \$HOME directory. The following list sets several useful settings for C programming. The Emacs manual and judicious use of Google will reveal many more options if you desire.

1. Before you start to edit your “.emacs” file, please be sure to backup it. (Hint: Make a copy of the file with a different name)

2. Parenthesis auto matching:

Add the following to the end of the file:

```
(show-paren-mode 1) ; turn on paren match highlighting
```

3. Auto Indent:

Add the following to the end of the file:

```
;set auto indent for C programming
(add-hook 'c-mode-common-hook (lambda ()
  (local-set-key (kbd "RET") 'newline-and-indent)))
```

4. Show word count:

Add the following content to the end of the file:

```
;;;show work count: (use M-x wc)
(defun wc ()
  "Count the words in the current buffer, show the result in the minibuffer"
  (interactive))
```

```
(save-excursion
(save-restriction
(widen)
(goto-char (point-min))
(let ((count 0))
(while (forward-word 1)
(setq count(1+ count)))
(message "There are %d words in the buffer" count))))))
```

When you open Emacs again, you are able to use the command “M-x wc” to display the word count.

5. Disable welcome screen: Add the following content to the end of the file:

```
;;hide welcome scree
(setq inhibit-splash-screen t)
(setq inhibit-startup-message t)
```

6. Change colors:

- open Emacs, type the command “M-x customize-themes”, and then choose the theme you like and save the setting. Try the “manoj-dark” theme.
- Theme changes will automatically be written to the “.emacs” file by Emacs.

## 4.2 The GCC Compiler

To run a C program, you need to rely on a C compiler to generate the executable file on your operating system from your C source files. In Linux, GCC is the most popular compiler and will be used in this course. The simple programs you will be writing in this lab should only require one command to produce a binary; however, you should be aware of what happens behind the scenes for future purposes.

To compile C code, the following 4 stages are run by gcc:

1. **Preprocessing.** The compiler loads included files, but it does many other things at this stage which we will not introduce yet. In a basic “Hello World” program, the included file “stdio.h” will be loaded in this stage. You can add a -E flag to the gcc command to see the results of this stage.
2. **Compilation.** The C compiler generates assembly code for your C file. These are the actual instructions the CPU will run to execute your program. You can add a -S flag to the gcc command to see the result of this stage (stored in a .S file).
3. **Assembly.** An assembler is used to translate the assembly instructions into machine code, or object code. These are the 1s and 0s that correspond to the instructions generated in the previous stage. You can add a -c flag to the gcc command to see the result of this stage (stored in a .o file).

4. **Linking.** This stage generates an executable file. Linking allows your programs to share code without duplicating the same bits on your machine. We will make use of this later as our programs become more complex. For now, you can add a `-o <name>` flag to the `gcc` command to set the name of your executable.

For the simple, one file C programs in this lab, you can compile them by running `gcc -o <name> <file.c>` where `name` is the name you want the program to be called and `file.c` is the source file with your C code. To run your program, enter `./<name>` in your shell, which instructs it to run the program called `name` in the current directory.

## 4.3 Hello World!

Follow the tutorial in [Chapter 1 of C Programming Language, 2nd Edition](#). Pay special attention to the sections called out in the reference section for this lab. Take time to play around with the programs provided, making changes to the code and observing the results is the most effective learning technique and some time spent becoming familiar now will shorten headaches later.

### Exercise 1. (5 Points)

Begin your lab submission by:

1. Logging into your VM and opening a terminal
2. Running `git clone` followed by the URL you copied from the web site
3. Using `cd` to move into your new lab directory (lab-4-netid)
4. Follow the directions in the README.md file to checkout a branch named "develop"

Write a program that prints "Hello, world!" when run. Name the source `ex_1.c` in your lab repository.

Remember to commit your completed exercise.

## 4.4 Data Types

All the data you can use and manipulate in a C program has a *type*. There are types to store natural numbers, real numbers, or characters from the English alphabet. The basic data manipulation element in a C program is called a *variable*. The program can use and manipulate data stored inside a variable, and every variable has an associated data type. A data type in a C program tells the compiler how much storage is to be allocated to the associated variable in the computer's memory. For example, a variable that contains the number 10 needs less memory compared to a variable that contains 1 billion. Let's look at some of the basic data types used by C.

### 4.4.1 Integer Types

The basic integer types to store natural numbers are as follows. Note: 1 byte = 8 bits of memory.

Type	Storage Size	Value Range
char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

### Exercise 2. (6 Points)

Which data types will you use for the integers given below? Put these answers – one per line – in a file called `ex_2.txt` in your lab repo. There might be several possible types that can be used for an integer, write the one best suited for that number in terms of storage size, and value range (i.e. while the largest sizes can hold most numbers, they decrease the speed of the program and increase that space required to run – so when possible it is better to use the smallest reasonable type).

- |          |                               |
|----------|-------------------------------|
| 1. 1,854 | 4. $10^6$                     |
| 2. 23    | 5. $2^{100}$                  |
| 3. -34   | 6. Character 's' (not a typo) |

Remember that these exercise are graded by computer, so put one answer per line with no extraneous information as directed. Each line should be one of the above types or "none" if there isn't an appropriate answer. Commit your exercise when complete.

The wonderful thing about characters from the English alphabet is that there are not too many of them, and each letter can be stored as a number. No wonder that the data type `char` gets its name from its ability to store characters. The American Standard Code for Information Interchange (ASCII) standardizes how each character is represented using an integer. For example, the character 'a' can be represented using the number 97.

*Reference:*

**ASCII Table:** For different ASCII encodings, refer <https://ascii.cl/>.

How many ASCII characters are there in the ASCII code table? What is the correlation between the number of ASCII characters and the range of the `char` data type?

### 4.4.2 Floating-Point Types

The basic floating-point types to store real numbers are as follows. Note: 1 byte = 8 bits of memory.

Type	Storage Size	Value Range	Precision
float	4 byte	$1.2 \times 10^{-38}$ to $3.4 \times 10^{38}$	6 decimal places
double	8 byte	$2.3 \times 10^{-308}$ to $1.7 \times 10^{308}$	15 decimal places
long double	10 bytes	$3.4 \times 10^{-4932}$ to $1.1 \times 10^{4932}$	19 decimal places

### Exercise 3. (3 Points)

Which floating-point type will you use for the real numbers given below? Put these answers – one per line – in a file called `ex_3.txt` in your lab repo.

1. 2.34

2. 3.65728383

3. 1.5E+23

Remember that these exercise are graded by computer, so put one answer per line with no extraneous information as directed. Each line should be one of the above types or "none" if there isn't an appropriate answer. Commit your exercise when complete.

What is the 'E' in the third real number? (Hint: Scientists write very large floating numbers using this notation)

#### 4.4.3 Variable Usage in C

All variables used in a C program need to have a type. A variable is considered “declared” if it is specified in the C program before it is actually used. A typical variable declaration in C looks like

```
type variable_name; /*example variable declaration*/
```

Some examples of declared variables are

```
unsigned char letter;          /*declares a character called 'letter' */
int sign;                     /*declares an integer called 'sign' */
float acceleration;           /*declares a floating-point number called 'acceleration' */
unsigned long distanceToMoon; /*declares an unsigned long integer called 'distanceToMoon' */
```

A variable in C can be initialized to a value. A typical statement declaring and initializing a variable in C looks like

```
type variable_name = initial_value; /*both declares variable_name of type 'type' and assigns initial_value to it*/
```

It is important to make sure that the initial value assigned to the variable is in the range of the type associated with the variable. Bad things happen if that isn't the case. More on this later.

#### 4.4.4 Variable Arrays

*Arrays*, are a collection of **variables of the same type**. For example, if we want to use 20 integer variables in a program, instead of declaring 20 different variables individually, we can simply declare an array like this

```
int numbers[20]; /*declares an array of 20 integers*/
```

A typical array declaration is `type array_name[size]`. Array indexing starts from 0 to `size-1`. The first element in the example array is `numbers[0]` and the last element is `numbers[19]`. Elements in an array are assigned values the same way as normal variables. For example, to assign a value of 98 to element 11 of the `numbers` array we use `numbers[10] = 98`.

#### 4.4.5 Passing arguments to your program

Recall that in the “Hello World” example, there is the function “main” and the program exactly executes the instructions in this function. In fact, there is exactly one “main” function for each C program, and it is considered the “entry” of the C program. Only instructions in (or invoked by) this function will be executed by the program. Therefore, it is important for this function to interact with the environment outside the program. Arguments to the C program from the command line are passed into the main function through 2 special parameters.

The formal declaration of “main” is as follows:

```
int main (int argc, char **argv); /*main takes command-line arguments*/
int main (int argc, char *argv[]); /*main takes command-line arguments*/
```

The two declarations are equivalent, and they indicate that the return value of “main” is an integer (int). The argument “argc” stores the number of the arguments passed to your program, and “argv” stores all arguments passed to your program. To explain these two arguments clearly, we introduce a small example. Create a C file named “argv.c” and save the following code to the file.

```
#include <stdio.h>

int main (int argc, char **argv){
    int i = 0; /*loop variable*/
    for (i = 0; i < argc; i ++){ /*loop through all command-line arguments*/
        /*print each argument to the screen*/
        printf ("Argument %d: %s\n", i, argv[i]);
    } /*end for*/
    return 0; /*terminate without error*/
} /*end main*/
```

Compile and generate the executable file “argv”. What is the output when you just type “./argv”? And what is the difference when you type “./argv 1 2 3 4” and “./argv “1 2 3 4””? Try to run the program “argv” by providing more arguments. Can you finally figure out the meaning of two arguments of the function “main”? :)

### 4.5 printf and scanf

We have frequently used the C function “printf” in our previous examples. It is an important output function for C which provides a way to pass the output of the program to the external environment. Dually, there is an input function for C, namely “scanf”, which is used to accept the input from the external environment of the program. In the previous section, we saw how C passes arguments to the “main” function, which you can provide from the command line. While `argv` provides input from invocation (when the shell started the program), `scanf` provides input at runtime, while the program is executing. Try the following program to test “scanf”.

```
#include <stdio.h> /*both scanf and printf are included in this file*/
#include <string.h> /*string also needs to be included*/
```



```

int main (int argc, char **argv){
    char s[100]; /*declare a 100-character array*/
    while (1){ /*NOTE: Be VERY careful when writing infinite loops!!!*/
        printf ("Please input something (length not exceeding 100):\n");
        scanf ("%s", s); /*read in character 's' */
        printf ("Your input is: %s\n", s); /*print out s for debugging*/
        if (strcmp (s, "0") == 0){ /*an important test*/
            break; /*break out of the 'while' loop*/
        } /*end if*/
    } /*end while*/
    return 0; /*terminate normally*/
} /*end main*/

```

Save this code to a file named “scanf.c”. Compile and run it. Try to type anything whose length is less than 100. Does “scanf” correctly accept all your inputs? If not, do you know the reason why? Finally, how can you terminate the program?

“strcmp” is a function in C that is used to compare whether the given two strings (here, it’s your input and “0”) are the same. It returns 0 if the answer is positive and otherwise returns a non-zero value. To be honest, there are better input functions than “scanf” to accept inputs. For example, for the above program, what happen when you type just the <RET> key? what happens when you give the input containing <SPACE>?

As you can see, both the “printf” and “scanf” use the characters like “%d” and “%s”. They are called *format specifiers* which are used to replace the arguments passed to these functions. For example, in the program “scanf.c”, “%s” in both functions are replaced by your inputs. Table 4.1 lists some commonly used format specifiers.

Format Specifier	Type of value
%d	Integer
%f	Float
%lf	Double
%c	Single character
%s	String
%u	Unsigned int
%ld	Long int
%Lf	Long double

Table 4.1: Format Specifiers frequently used.

You may also use “\n” and “\t” in “printf”. These are *escapes* used in terminal output. “\n” says to print a new line (like pressing enter), and “\t” says to print a tab-stop (like pressing the <tab> key). We will use these to format our output.

## 4.6 Debugging with printf

It is endlessly lampooned on the internet, but programming is an error-prone process. Any code doing non-trivial work (and even some trivial programs) has a high probability of diverging from the expected behavior. This deviation from desired operation is called a "bug" and you have probably encountered this already when constructing piped command line expressions. The process of identifying and correcting bugs is "debugging" and is considered to be both more difficult than writing the original program, and more time consuming as well.

### *Warning:*

When planning your efforts for the remainder of this class, remember to include time for debugging. The program may seem simple (and probably is) but you never know if it will run the first time, or require many times over your original effort just to find a simple typo. You have been warned.

The root cause of a bug can be as obtuse as hardware radiation faults (highly unlikely) to simple typos (a mistake even experts commonly make). Tracking these defects is a particular skill that will be honed throughout this course and lab 6 will be dedicated to debugging methodologies, techniques and tools. Until then, printf can be used to provide insight into the inner workings of your program.

The simple programs of this first C lab can be debugged by adding "printf" statements into your code to display internal state, or identify when the execution has "hit" a particular line. The exact method of applying this technique is highly situation-dependent and seeped in personal preference but some points to consider:

- Visually distinguishing debug output from normal output is helpful. Try beginning all your debug strings with something to offset and identify them, like in this example:  

```
printf("\tDEBUG: i=%d", i);
```
- When debugging loops (introduced below), consider printing the loop condition or variable every cycle, to see if the bug is dependent on the loop number.
- When debugging branch points (introduced below) consider printing the variables involved in the decision before the condition and which branch was taken after – to compare your expected behavior with the actual execution.
- When they are no longer needed, comment out the statement. If you delete them entirely, you'll have to duplicate the work of adding them if you have need of it again, but leaving these extra prints on clutters the output and can slow down your program.

## 4.7 Mind Your Types!

Imagine fitting a 4 sq. inch cube in a 1 sq. inch sphere. Impossible right? How about fitting two 1 sq. inch cubes in the 4 sq. inch cube? Precisely. We can do that. If we are trying to move things and put them in new places, the dimension requirements need to be met. The same goes for data types in a C program as well. As an example, consider we have

100 `unsigned char` variables and we want to add them together and store them in a new variable. The operation we want to do is

```
unsigned char a1 = 34, a2 = 45, ..., a100 = 123;
type sum = a1 + a2 + ... + a100;
```

Can the data type of `sum` be `unsigned char`? Let's assume for the time being that it is possible. Recall that every `unsigned char` can take a number between 0 and 255. This means that the sum of all 100 `unsigned chars` can be up to  $100 * 255 = 25,500$ . Aha! This cannot be stored in a `unsigned char`! In this case, it is a better choice to use an `int` for this type of sum.

*Type Mismatch* errors are the most common errors in a program. These errors are often not reported by the compiler and are the most difficult to debug. However, execution of a program containing these can wreak havoc. Fire up the emacs editor and type in the following program.

```
#include <stdio.h>
int main(void) {
    unsigned char a1 = 125, a2 = 126, a3 = 127, a4 = 128;
    unsigned char sum;
    sum = a1 + a2 + a3 + a4;
    printf("Total: %d\n", sum);
    return 0; /*terminate normally*/
} /*end main*/
```

Save and compile the program using `gcc`. Does the compiler report an error? A simple calculation reveals that the output of the program should be  $125+126+127+128=506$ . Now execute the compiled binary and see the result. Does it match?

#### Exercise 4. (2 Points)

Assuming we are adding 100 `unsigned char` variables that can take the values 0,1, or 2. Which is the “most efficient” data type we can use to store their sum? Put this answer in a file called `ex_4.txt` in your lab repo.

Follow the same formatting instructions as the previous exercises and commit when complete.

#### Exercise 5. (9 Points)

For each of the following, compare the expected behavior listed with the output of the program. Does the compiler report any errors? If there is an error, what is the reason behind it? How would you correct them? Put these answers in a section of your report titled "Exercise 5" and use the "itemize" format to separate the three sub-problems.

1.  $123 + 567 = 690$

```
unsigned char a = 123; unsigned int b = 567;
unsigned char sum = a + b
printf("Result: %d\n", sum);
```

2.  $1/10 = 0.1$

```
float a = 1.0; unsigned int b = 10;
unsigned int div = a/b
printf("Result: %d\n", div);
```

3.  $3.6 = 3.60000000000000000000$

```
printf("%.20f\n", 3.6);
```

Your response for this exercise will appear in your report. Make sure that it is clearly identifiable for the TA.

## 4.8 Functions

So far, you have used the "main" function as well as the "printf" and "scanf" functions. Consider your program as a system, the functions are all modules to build your system. To keep code organized and useful, a function should do exactly one job and stay relatively small (under 100 lines). There are many more rules of style that will be introduced as we dive deeper into C. The general form of a function definition in C programming language is as follows:

```
return_type function_name( parameter_list ) {
    body
}
```

Where the parts are defined as:

- **Return Type.** A function may return a value. The `return_type` is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the `return_type` is the keyword `void`.
- **Function Name.** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters.** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. These are the function arguments. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body.** The function body contains a collection of statements that define what the function does.

### Exercise 6. (10 Points)

Expand your program "Hello World" to have a function that it calls. This function prints "Hello World" instead of main. Main now just calls the function. Name the source `ex_6.c` in your lab repository.

Your source code should have two function declarations and print "Hello World" when run. Commit your source code when complete.

## 4.9 Conditionals and Loops in C

You should already be familiar with the basics of conditionals and loops from the prerequisite courses. The syntax for these constructs in C are given in the reference.

### Exercise 7. (10 Points)

Write a C program that takes input at runtime, and reports if the input is an integer or not by printing "That is an integer" if it is, or "That is not an integer" if it is not. Your program should exit with a non-zero error-code if it was not an integer and with 0 as the return code if it was. Name the source `ex_7.c` in your lab repository.

Remember that your work is graded by a computer so ensure your program returns exactly as described. Commit your source when complete.

### Exercise 8. (15 Points)

Loop 5 times. Each time, ask the user for an integer, check that the input received is that type, and then either terminate with an error message or ask for another input until the user has given five correct inputs. Name it `ex_8.c` in your lab repository. The output of this program should look like this, assuming that the user gives five correct inputs of type int:

```
Hello! Please give me an integer: 0
Thanks! Please give me another integer: 1
Thanks! Please give me another integer: 1
Thanks! Please give me another integer: 2
Thanks! Please give me another integer: 3
Thanks! I am happy with five integers.
```

Test your code with a variety of inputs (what happens if the first input is a letter, what about the 3rd? What if you enter nothing, or a space?) and commit your source when you are satisfied.

### Exercise 9. (20 Points)

Ask the user for an integer 'n', loop n times. Each time, ask the user for a char, check that the input received is that type, and then either terminate with an error message or ask for another input until the user has given n correct inputs. Name the source `ex_9.c` in your lab repository. The output of this program should look like this, assuming that the user gives n = 2 correct inputs of type char:

```
Hello! How many chars should I take? 2
Please give me a char: a
Thanks! Please give me another char: b
Thanks! I am happy with 2 chars.
```

Half credit is awarded for programs that can handle the best case scenario (where a user follows instructions) and the other half is awarded to programs that can handle "off-nominal" conditions, such as the user entering full words, number, symbols, etc. Commit your source file when complete.

## 4.10 A Basic Introduction to Complexity

There can be multiple solutions to a problem, and a C program is essentially solving a problem. For any given problem, there may be a multitude of solutions – but some solutions are faster, cheaper or better than others.

For example, let's assume we want to solve a hypothetical puzzle using a program. To do this, we end up writing two C programs. The first program reads the puzzle and gives the correct answer in 1 second. The second program reads the same puzzle, and gives the same correct answer in 2 hours. Which program will you eventually use to solve the puzzle?

The **Big-O** notation is used to describe the performance or complexity of an algorithm implemented as a program. Big-O specifically describes the worst-case scenario, and can be used to describe the execution time required or computer memory used by an algorithm. In this lab, we focus on the former. The Big-O complexity of an algorithm is specified in terms of the size of the input. For the sake of explanation, we will use the size of the input for  $n$ . For example,  $n$  integers,  $n$  characters, etc.

### 4.10.1 $O(1)$ Complexity

$O(1)$  describes an algorithm that will always execute in the same time, regardless of the size of the input data set. For example, consider the function

```
int add(int a, int b) {  
    return a+b;  
}
```

The function `add()` runs in constant time irrespective of the values of  $a$  and  $b$ , and therefore the time complexity of `add()` is  $O(1)$ .

### 4.10.2 $O(n)$ Complexity

$O(n)$  describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data. For example, consider the function below that adds all numbers in an array (an array is simply a collection of elements of the same data type).

```
int addArray(int numbers[], int n) {  
    int sum = 0;  
    for(int i = 0 ; i < n ; i = i + 1) {  
        sum = sum + numbers[i];  
    }  
    return sum;  
}
```

If the value of  $n$  (size of numbers array) is 10, the `for` loop iterates 10 times. Similarly, if  $n = 10^{100}$ , the `for` loop iterates  $10^{100}$  times. To summarize, for a given value of  $n$ , the `for` loop iterates  $n$  times. Therefore, the time complexity of the `addArray` function is  $O(n)$ .

### 4.10.3 $O(n^2)$ Complexity

$O(n^2)$  represents an algorithm whose performance is directly proportional to the square of the size of the input data. For example, consider a function that finds the duplicate numbers in an array of integers of size  $n$ .

```
int findDuplicates(int numbers[], int n) {
    for(int i = 0 ; i < n ; i = i + 1) {
        for(int j = 0 ; j < n ; j = j + 1) {
            if(i != j) {
                if(numbers[i] == numbers[j])
                    return 1; /* found a duplicate */
            } /*end if*/
        } /*end for j*/
    } /*end for i*/
} /*end findDuplicates()*/
```

The function `findDuplicates` picks up a value of  $i$  in the outer loop. In the inner loop, `numbers[i]` is compared to all other numbers in the array, for which  $i \neq j$ : `numbers[i]` is compared to `numbers[0]`, `numbers[1]`, ..., `numbers[j]`. If at any point, `numbers[i]` is equal to `numbers[j]`, the function return a value of 1 and terminates. Let us consider a few scenarios assuming  $n = 10$ .

- If `numbers[0]` is equal to `numbers[1]`, the outer loop is executed only once, while the inner loop is executed twice.
- If `numbers[3]` is equal to `numbers[5]`, the outer loop is executed four times, while the inner loop is executed  $3 \cdot 10 + 6 = 36$  times. Can you see why?
- If `numbers[6]` is equal to `numbers[9]`, the outer loop is executed seven times, while the inner loop is executed  $7 \cdot 10 = 70$  times.

Now consider the case in which `numbers` doesn't contain duplicate numbers. In this scenario the outer loop is executed 10 times, and the inner loop is executed 100 times! Since Big-O complexity deals with the worst case, the worst case for `findDuplicates` happens when the `numbers` array, of size  $n$ , does not contain duplicates, and the inner loop is executed a total of  $n^2$  times. Therefore, the Big-O complexity for the above function is  $O(n^2)$ . In general, for  $m$  number of nested for loops, the worst case complexity is  $O(n^m)$ .

*Reference:* (Optional) For an anecdotal summary of several Big-O complexities we recommend Stuart Kuredjian's article on *Algorithm Time Complexity and Big-O Notation* available at: <https://medium.com/@StueyGK/algorithm-time-complexity-and-big-o-notation-51502e612b4d>

## 4.11 A First Computational Program

### Exercise 10. (40 Points)

For this exercise you will write **two** small C programs. When run, both programs will prompt the user for an **integer**  $n$  from the command line, read it in, check that it is indeed

an integer, and print the number from the `main` function, then call a function to compute the sum from  $1 \dots n$ . If  $n$  is not an integer the program should terminate with an appropriate message upon checking  $n$  (and therefore not proceed or call any functions.)

- The first program will be named `BruteForceAdder.c`. After printing  $n$ , the `main` function of `BruteForceAdder.c` will call a function named `BruteForceAdder` and pass this function  $n$ . The function `BruteForceAdder` will print a message to the screen telling the user that it has begun adding the numbers from 1 to  $n$ . This function will then proceed to add the numbers from 1 to  $n$  the brute force way: in a loop. The program will print the sum with a nice, descriptive message, and terminate.
- The second program will be named `GaussAdder.c`. After printing  $n$ , the `main` function of `GaussAdder.c` will call a function named `GaussAdder` and pass this function  $n$ . The function `GaussAdder` will print a message to the screen telling the user that it has begun adding the numbers from 1 to  $n$ . This function will then proceed to add the numbers from 1 to  $n$  the Gaussian way:

$$\sum_{i=1}^{i=n} i = \frac{n(n+1)}{2}$$

The program will print the sum with a nice, descriptive message, and terminate.

Run both of these programs with a large value of  $n$  from a bash shell, using the Linux `time` command. What does the time command tell you about the performance of both programs? Try increasingly larger value of  $n$ . How much does it matter to implement an intelligent algorithm versus a brute force one? **In your L<sup>A</sup>T<sub>E</sub>X report, describe the two algorithms – the Brute Force one, and the Gauss one – using the appropriate equations to describe how they compute the sum.**

**Also include your Big-O time classification for each method, and your analysis on the difference in performance as measured by the Unix time command.** In the report section there is a performance table to fill out that will help you answer these questions. The point distribution of this assignment is: 10 points for each working implementation (brute force and Gaussian), 10 points for the relative performance of the programs (roughly following their respective big-O complexity), and 10 points for your writeup and analysis.

*A note on testing:* (Optional) For an anecdotal summary of several Big-O complexities we recommend Stuart Kuredjian's article on *Algorithm Time Complexity and Big-O Notation* available at: <https://medium.com/@StueyGK/algorithm-time-complexity-and-big-o-notation-51502e612b4d>

## 4.12 Report Requirements

Your lab report should include similar three components to last week, each worth 10 points:

- a title page with your name, course name, assignment name, etc.



- a personal C “cheat sheet” with at least ten functions, compiler options, or patterns you used in the exercises or find useful
- any sources or resources you use while completing the exercises

Your report will also contain a section (use the "section" command seen in previous reports) containing your responses for exercise 5. Use the "itemize" structure to separate the 3 sub-components of the exercise (see the lab 2 template for an example of itemize).

Finally, your report will contain a section for exercise 10. Answering the questions listed in exercise 10 section are worth points as listed. Additionally 10 report points are awarded for completing a table comparing the performance of your two programs. LaTeX code of such a table is given below and should be filled out using the performance of your programs (use the 'time' command to test them).

$n$	Brute Force Time	Gauss Adder Time
1	?	?
10	?	?
100	?	?
1000	?	?
100000	?	?

The L<sup>A</sup>T<sub>E</sub>Xsource for this table is:

```
\begin{center} \begin{tabular}{|c|r|r|}
\hline
 $n$  & Brute Force Time & Gauss Adder Time \\
\hline
\hline
1 & ? & ? \\
\hline
10 & ? & ? \\
\hline
100 & ? & ? \\
\hline
1000 & ? & ? \\
\hline
100000 & ? & ? \\
\hline
\end{tabular} \end{center}
```

In total, your report this week is worth 40 points. You may use previous reports as a starting point or start fresh. Your report should be in a file named `main.tex` in a directory named `report` in your lab directory. While you are allowed to commit the pdf, it will not be used – your `.tex` source file will be graded. It is actually preferable not to include pdfs in git repos if the source to generate them is included as it dramatically inflates the size of the download.

Ensure your report builds without errors before submitting! If you have to interact with `pdflatex` to build the pdf (i.e. hit the Enter key), that counts as an error.

## 4.13 Lab work submission

*Warning:* As stated in the syllabus, programs that do not compile and L<sup>A</sup>T<sub>E</sub>X reports that do not compile will be awarded zero points!

If you have issues with your lab work, utilize your resources (lecture notes, text books, lab manuals, personal internet research) and feel free to contact the TA email for assistance if you are still unable to resolve your issue.

When you are finished with your lab, follow this procedure to turn in your work for grading:

1. Ensure all 10 exercises have a correctly named file in your lab directory
2. Ensure each of the 7 res files contain only the requested information and nothing more
3. Ensure that your report is named correctly, is in the correct directory, and builds without errors
4. Use `git status` to verify all the work you want to be graded has been committed to the develop branch
5. Use `git push` to update GitHub with all your changes
6. Open a pull request on GitHub
7. **DO NOT** click the "close and merge" button - your PR should remain open for the autograder to find

Congratulations, you are 28.57142857% of the way though 361!

## 5 | More C ... and Complexity

*Reference — C Programming Language, 2nd Edition:*

[Chapter 7— Pointers and Arrays](#)

[Chapter 7 – Input and Oputput](#)

Purpose: To gain increasing familiarity with the basics of C programming for scientific computing, including syntax, control structures for intelligent data analysis, and the importance of considering the complexity of programs when initially designing the program.

### Objectives

- Opening and closing files; managing input, output, and error streams
- Arrays
- Memory: `malloc` and `free`
- Passing data: by reference versus by value
- Specializing data structures with `structs`

## 5.1 File Management

### 5.1.1 Redirection

One way to read input into a program or to output from a program is to use standard input and standard output, respectively. That means, to read in data, we use `scanf()` (or a few other functions) and to write out data, we use `printf()`. When we need to take input from a file (instead of having the user type data on the keyboard) we can use input redirection:

```
$ a.out < inputfile
```

Here “a.out” is the executable file compiled from C code. The same applies to the following. This allows us to use the same `scanf()` calls we use to read from the keyboard. With input redirection, the operating system causes input to come from the file (e.g., `inputfile` above) instead of the keyboard. Similarly, there is output redirection:

```
$ a.out > outputfile
```

The above command allows us to use `printf()` as before, but causes the output of the program to go to a file (e.g., `outputfile` above) instead of the screen.

Of course, the 2 types of redirection can be used at the same time...

```
$ a.out < inputfile > outputfile
```

### 5.1.2 C File I/O

While redirection is very useful, it is really part of the operating system (not C). However, C also has a general mechanism for reading and writing files, which is more flexible than redirection alone.

**Include file.** There are types and functions in the library **stdio.h** that are used for file I/O. Make sure you always include that header when you use files.

**File Type.** For files you want to read or write, you need a file pointer, e.g. `FILE *fp`; What is this type "FILE \*"? Right now, you don't need to know. Just think of it as some abstract data structure, whose details are hidden from you. In other words, the only way you can use a `FILE *` is via the functions that C gives you.

Note: In reality, `FILE` is some kind of structure that holds information about the file. We must use a `FILE *` because certain functions will need to change that information, i.e., we need to pass the information around.

**Functions.** Reading from or writing to a file in C requires 3 basic steps:

- Open the file.
- Do all the reading or writing.
- Close the file.

Below are the functions needed to accomplish each step.

### 5.1.3 Open a file

Mode	Description
r	Opens an existing text file for reading purposes.
w	Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content to the existing file content.
r+	Opens a text file for both reading and writing.
w+	Opens a text file for both reading and writing. It first resets the file to the zeroth location if it exists, otherwise creates a file if it does not exist.
a+	Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

Table 5.1: File open modes in C.

You can use the `fopen()` function to create a new file or open an existing file. This call will initialize an object of the type `FILE`, which contains all information necessary to control the stream. The prototype of this function call is as follows:

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

Here, “filename” is a string literal, which you will use to name your file, and access “mode” can have one of the values shown in Table 5.1.

If you are going to handle binary files, then you will use the following access modes instead of those mentioned above:

```
"rb", "wb", "ab", "rb+/r+b", "wb+/w+b", "ab+/a+b"
```

#### 5.1.4 Read from or write to a file

**fscanf/fprintf.** Once a file has been successfully opened, you can read from it using `fscanf()` or write to it using `fprintf()`. These functions work just like `scanf()` and `printf()`, except they require an extra first parameter, a `FILE *` for the file to be read/written. An example to use `fscanf` and `fprintf` is shown as below.

```
/* fscanf example */
#include <stdio.h>

int main (void) {
    char str [80]; /*str is an 80-character string*/
    float f;       /*f is our example floating-point number*/
    FILE * pFile;  /*pFile is the file stream where we will write out*/
    int return_error = -1; /*this is the indicator of a NULL pointer*/

    pFile = fopen ("myfile.txt","w+"); /*open the file with read&write permission*/
    if (pFile == NULL) { /*always check for errors before accessing the file*/
        fprintf(stderr, "ERROR opening OUT file; Ending with return %d\n", return_error);
        return return_error; /*indicate an eerror and end execution */
    } /*end if*/
    fprintf (pFile, "%f %s", 3.1416, "PI");
    rewind (pFile); /*set the FILE pointer to the beginning*/
    fscanf (pFile, "%f", &f);
    fscanf (pFile, "%s", str);
    fclose (pFile);
    printf ("I have read: %f and %s \n",f,str);
    return 0; /*terminate normally*/
} /*end main*/
```

This sample code creates a file called `myfile.txt` and writes a float number and a string to it. Then, the stream is rewound and both values are read with `fscanf`. It finally produces an output similar to:

```
I have read: 3.141600 and PI
```

**fgets/fputs.** You can also use `fgets/fputs` to read and write a buffer (string) to a file. This is shown below:

```
/* using fgets() and fputs() */
#include <stdio.h>
```

```

#define MAXLINE 20 /*MAXLINE is our constant, hard-coded in by the C pre-processor*/
int main(void) {
    char line[MAXLINE]; /*a line is an array of characters MAXLINE long*/
    while (fgets(line, MAXLINE, stdin) != NULL &&
           line[0] != '\n') {
        fputs(line, stdout);
    } /*end while*/

    return 0; /*terminate normally*/
} /*end main*/

```

In the above code, “stdin” are “stdout” are two special FILE pointers corresponding to the standard input and output. We will introduce them more in the following section. Compile and run this program. Try to give the program two inputs: "hello world" and "hello world hello world hello world". Can you find anything strange?

For the second input, its length is larger than 20, which exceeds the maximum length of the buffer “line” in the program. In your testing, does this program works fine? Why?

Function	Description
fscanf	Read (raw) data from a file.
fprintf	Write (raw) data to a file.
fgets	Read a buffer (string) from a file.
fputs	Write a buffer (string) to a file.
fread	Read a data structure from a file.
fwrite	Write a data structure to a file.

Table 5.2: File read/write APIs in C (most frequently used).

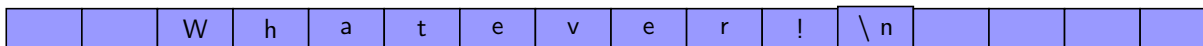
**fread/fwrite.** The third option to read and write a file is to use fread/fwrite. These two APIs are most often used to read/write binary files. Examples for these two APIs are omitted. Table 5.2 shows a summary of the introduced FILE read/write APIs and their differences.

### 5.1.5 Close a file

When done reading/writing to a file, it must be closed using the function fclose(). Closing a file is very important, especially with output files. The reason is that output is often buffered. This means that when you tell C to write something out, e.g.,

```
fprintf(ofp, "Whatever!\n");
```

it doesn’t necessarily get written to the disk right away, but may end up in a buffer in memory. This output buffer would hold the text temporarily:



When the buffer is full, the file is closed, or explicitly asked (with a "flush" command) and

the data gets written to the disk. So, if you forget to close an output file then whatever is still in the buffer may not be written out, causing data loss!

### 5.1.6 Special file pointers

There are 3 special FILE \* pointers that are always defined for a program. They are “stdin” (standard input), “stdout” (standard output) and “stderr” (standard error).

**Standard input** is where things come from when you use scanf(). In other words,

```
scanf("%d", &val);
```

is equivalent to the following fscanf():

```
fscanf(stdin, "%d", &val);
```

**Standard Output** Similarly, standard output is exactly where things go when you use printf(). In other words,

```
printf("Value = %d\n", val);
```

is equivalent to the following fprintf():

```
fprintf(stdout, "Value = %d\n", val);
```

Remember that standard input is normally associated with the keyboard and standard output with the screen, unless redirection is used.

**Standard error** is where you should display error messages. For example,

```
fprintf(stderr, "Can't open input file in.list!\n");
```

Standard error is normally associated with the same place as standard output; however, redirecting standard output does not redirect standard error.

For example,

```
$ a.out > outfile
```

only redirects stuff going to standard output to the file “outfile”. Anything written to standard error goes to the screen.

## 5.2 Runtime Memory

It isn't always possible to know how much memory you will need when it is time to compile, but without knowledge of size of the variables, the compiler cannot generate code. How do programs written in C handle cases where they don't know the length or form of input until execution? *Dynamic Memory Allocation*

### 5.2.1 Manual Memory Management

This was hinted at in the last lab and will be covered in greater depth during the next lab. But for now, we should cover some basics.

Included in "stdlib.h" are 4 functions for managing memory at runtime:

**malloc** : Request memory from the operating system

**free** : Return memory to the OS after it is no longer needed

**calloc** : Request memory and set all values to 0

**realloc** : Request the sized of a previously allocated memory

We will only focus on malloc and free right now.

### 5.2.2 Pointer Basics

Since our program doesn't know the amount of memory it needs at compile time, we can request more during execution. This leads to a chicken & egg problem, how can we address this memory at compile time if we don't know how big it is yet? To solve this, we use *pointers*.

You had to deal with pointers when using scanf because they are used in many string-related functions. They will be covered in further depth next week, but in a nutshell, a pointer is a variable that stores the memory address of another value. In this case, at compile time we reserve a variable that will point to the memory that is requested at run time. Pointers are denoted with an "\*" as part of their type. For example

```
int* p;
```

declares a variable "p" that stores the memory address of an integer. Note that the pointer has to know what type of data it is pointing to!

Just as we used the reference operator "&" to read the address of a variable, we can use the dereference operator "\*" to tell the program "read the value stored at the memory address held by this pointer" like in this example (in your lab repo as ptr\_adr.c):

```
#include <stdio.h>
int main(void){
    /* Create a pointer to an int (currently points at a random memory location, why?) */
    int* pc;

    /* Create a normal int variable */
    int c;

    /* Assign the value 22 to variable c */
    c=22;
    /* Print the memory address of the variable */
    printf("Address of c:%u\n",&c);
    /* Print the int stored at that memory address */
    printf("Value of c:%d\n\n",c);

    /* Make the value of the pointer the address of c */
    pc=&c;
    /* Print the location in memory of pc */
    printf("Address of pointer pc:%u\n",&pc);
```



```

/* Print the value of pointer pc (where it is pointing) */
printf("Value of pointer pc:%u\n",pc);
/* Print the value stored in the memory address stored in pc (dereference) */
printf("Content of pointer pc:%d\n\n",*pc);

/* Reassign the value of C */
c=11;
/* Print the value of pointer pc (where it is pointing) */
printf("Value of pointer pc:%u\n",pc);
/* Print the value stored in the memory address stored in pc */
printf("Content of pointer pc:%d\n\n",*pc);

/* Set the value of the variable pointed to by pc to 2 */
*pc=2;
/* Print the address in memory of variable c */
printf("Address of c:%u\n",&c);
/* Print the value of variable c */
printf("Value of c:%d\n\n",c);

return 0; /*terminate normally*/
} /*end main*/

```

Run this program and observe it's behavior.

*Warning:*

Beware of these common mistakes:

**pc = c;** This tries to set the address pc points at to the value of c (address  $\neq$  value)

**\*pc = &c;** This tries to set the value of the memory pc points to to the address of c (value  $\neq$  address)

These are the most likely corrections to the above errors:

**\*pc = c;** Sets the value of the memory pointed at by pc to the value of c (value = value)

**pc = &c;** Sets the pointer pc to point at variable c (address = address)

Pointers and arrays are very tightly related in C, for more information refer to chapter 7 in The C Programming Language and the following website:

<https://www.programiz.com/c-programming/c-pointers-arrays>

### 5.2.3 malloc

Per the man page (run "man malloc"), the function malloc take the number of bytes to allocate and returns a pointer to the start of the memory if successful. If the OS can't provide the required memory, a pointer to NULL is returned instead.

For example, to allocate space for 15 characters:

```
char* str;
str = (char *) malloc(15 * sizeof(char));
```

We declared a pointer to chars called "str" then set the value of str to the address returned by malloc.

The "sizeof(char)" pattern multiplies the number of items we want space for by the number of bytes required for each item (1 in this case, but for types that take more space this is necessary).

The "(char\*)" component is called a cast and explicitly hints that we want the returned pointer to be a char pointer. This is controversial and some people will tell you it is an unnecessary repetition of information, others will point out several benefits and standards that require it. In this course, we will expect these casts at malloc as they are used in your C book and force you to think about what you are doing.

#### 5.2.4 free

When you are done using memory you should "free" it (give it back to the operating system). Forgetting to free memory, or freeing it improperly can cause many hard to find bugs with symptoms ranging from incorrect results to memory leaks (slowly using up the memory of a system, often until it crashes or slows to a crawl). This is especially important as aerospace engineers as you will frequently be working with large datasets that can easily crash machines or deliver hard-to-verify incorrect results.

To free memory, call the "free" function on the pointer to that memory.

```
char* str;
str = (char *) malloc(15 * sizeof(char));
/* Do things... */
free(str);
```

There will be much more on this in the next lab, but for now remember that malloc and free are pairs and you should free everything, and only things, you have malloc'd when finished with them.

#### 5.2.5 Example

Find the file `max.c` in your lab 5 repository. Read the source. What does it do? What behavior do you expect? This code doesn't contain input validation, what inputs could break it?

*NOTE:*

You normally won't do pointer arithmetic like is shown in this example directly. We will instead be using arrays and indices in this course but `*(ptr + i)` is the same as `arr[i]` in this case (but not always!) more on this next week.

### 5.2.6 Seg-Fault

By now, you may have received a "segmentation fault" error in some of your programs. This error occurs when your program attempts to read or write to memory that is not assigned to it. This is a vital function of the operating system to prevent a buggy program from breaking other programs on your system and to prevent malicious programs from stealing your data.

If you encounter this error, it will always be an issue with pointers and usually with arrays. We will cover more advanced debugging techniques next lab but for now it will suffice to check that:

- Every function argument is correctly referenced or dereferenced
- All array bounds are used correctly (see next section)

### 5.2.7 Foot-Guns

Feet-guns? Feets-gun?

Dynamic memory can be vary dangerous to your own program, modern OSs make it increasingly difficult (but not impossible) to cause problems with other parts of the system. But be prepared to not only experience seg-faults (the result of trying to access memory not allocated to your program) but also more subtle errors like wrong answers. Always test your programs throughly.

## 5.3 Arrays

We briefly introduced arrays in the last lab. In this lab, we focus on array indexing and how to read and fill data in to an array. An array is a series of elements of the same types, and the elements are stored in contiguous memory locations (more on this later). As explained before, to initialize an array for five `int` values, we use `int foo[5]`. In general, to initialize an array for  $N$  values of a given type, we use `type array[N]`. The first element in the array is at index 0, and the last element is at index  $N - 1$ .

### 5.3.1 Initializing Arrays

When we initialize the array for the first time, each element is randomly a value (value is in the range of the data type). For example, try running the code below.

```
#include<stdio.h>
int main(void) {
    int foo[100]; /*foo is an array of 100 integers*/
    for(int i = 0 ; i < 100 ; ++i) { /*loop through all of foo: array indices 0..99 */
        printf("%d\n", foo[i]);
    } /*end for*/
    return 0; /*terminate normally*/
} /*end main*/
```

The C compiler assigns a memory location to each element in the array, and since we haven't assigned the element a value to store, the `printf` displays any random value that is stored in the location for that element. To do something useful with the arrays, there are multiple ways to assign values to its elements.

- **Explicit Initialization:** of the array can be done by supplying the values to the elements when declaring the array. For example, to create an array of 5 elements containing integers 5, 10, 15, 20, and 25, we use

```
int foo[5] = {5, 10, 15, 20, 25};
```

To access the third element in the `foo` array, we use `foo[2]`.

- **Empty Initialization:** of the array is useful when we don't know what values to put in the array, or our C program fills an array to store information computed during execution. For example, to create an empty integer array of five elements, we use

```
int foo[5] = { };
```

This creates an array of five `int` elements, and each element value is initialized to zero.

- **Implicit Initialization:** allows the possibility of leaving the the size  $N$  of the array undefined. However, if no size is specified, element values need to to be given in curly braces. For example, to create an array containing the integers 10, 20, 30, 40, 50, 60, and 70, we use

```
int foo[] = {10, 20, 30, 40, 50, 60, 70};
```

In this case, the compiler automatically assumes a size of the array that matches the number of values included between the curly braces.

- **Dynamic Initialization:** is useful when the size of the array, or the elements to be stored in it are not known before actually running the program. For example, a program that asks an user to enter an integer  $n$ , then asks the user to enter  $n$  integers, and finally outputs all integer values entered. Notice that in this scenario, we do not know the size of the array to use for storing the integers before we actually run the program. The program to do this is given below.

```
#include<stdio.h>
#include<stdlib.h>
int main(void) {
    int n;
    int *foo; /* pointer */
    printf("Enter a number: ");
    scanf("%d", &n);

    /* create array of size n */
    foo = malloc(sizeof(int) * n);

    /* ask user to enter n numbers */
    for(int i = 0 ; i < n ; ++i) {
        printf("Number %d: ", i+1);
        scanf("%d", &foo[i]);
    }
}
```

```

    } /*end for*/

    /* print the n numbers read */
    for(int i = 0 ; i < n ; ++i) {
        printf("%d\n", foo[i]);
    } /*end for*/

    return 0; /*terminate normally*/
} /*end main*/

```

In the above program, `int *foo` declares a *pointer* (more on this in a later lab). The `malloc` statement (`stdlib.h`) in the program creates an array `foo[n]` when the program is executing. For the time being, use the above program as a reference implementation of creating dynamic arrays. We will go into more detail about pointers and dynamic memory allocation (`malloc`) in a later lab.

### 5.3.2 Multi-Dimensional Arrays

The type of arrays we have seen in the previous section are also called *one-dimensional* arrays, or 1-D arrays. In general, it is possible to have N-dimensional arrays. To declare a 2-dimensional array (also called a 2D-matrix), consider the declaration below

```
int bar[5][6]
```

The above statement declares a 2D array with 5 rows and 6 columns. We can store  $5 \times 6 = 30$  elements in this matrix. To access the element in the second row and fourth column, we use `bar[1][3]` (note that indexing starts from 0). In general, to declare a 2D matrix, we use `type name[rows][columns]`. What about initialization? A N-dimensional array can be initialized in any of the ways described in the last section. For example, to explicitly initialize a 2D matrix with two rows and 3 columns, we use

```
int bar[2][3] = { {12, 32, 54},
                  {67, 87, 32}};
```

Did you notice how a 2D array compares to a 1D array? A 2D array is an array of several 1D arrays. In general, a N-dimensional array is an array of several N-1 dimensional arrays. Dynamic initialization of a 2D array is similar as well. For example, to dynamically create an array of dimensions  $m$  (row)  $\times$   $n$  (columns), we use

```
int **bar; /* double pointer */
bar = malloc(sizeof(int *) * m)
for(int i = 0 ; i < m ; ++i) {
    bar[i] = malloc(sizeof(int) * n);
} /*end for*/

```

In the code above, `int **bar` is a *double pointer*, i.e., a pointer to pointers :-). Use the sample code here as reference to create dynamic 2D arrays in the exercise.

*Reference:*

Row-Major and Column-Major are two ways to store  $n$ -dimensional arrays in the linear computer memory. The choice of storage impacts performance. For details, refer to [https://en.wikipedia.org/wiki/Row-\\_and\\_column-major\\_order](https://en.wikipedia.org/wiki/Row-_and_column-major_order).

### 5.3.3 Array Bounds

When using arrays, it is vital to ensure you do not violate its bounds. As explained above, the array variable is just a pointer to the start of the array, and the index (the number or variable in the square brackets) acts as an offset. If you request an offset greater than the size of the array, your program will read and write to memory outside the array. Depending on the exact memory assigned, you might crash your program, read garbage data, or even silently corrupt another variable — an incredibly difficult situation to debug.

You should always verify that for an array of  $n$  items, you never use an index greater than  $n - 1$ . Be especially careful if your index is based off of user input since you cannot verify at compile time that your indexing is correct and your program must verify the bounds of the array before attempting to access it. There will be an exercise to practice this.

## 5.4 Exercises

### Exercise 1. File I/O — (30 points):

To practice working with files in C, write the following three simple file utilities. Each is worth 10 points, 5 for a properly compiling program and 5 for correctly completing the assigned task.

1. In a file named `head.c` write a program that prompts the user for a file name and tries to open the file. If the file cannot be opened, print an error message and return a non-zero exit code. If it can be opened, read the first 3 lines and print them to the screen before exiting with a zero exit code.
2. In a file named `tail.c` write a program that prompts the user for a file name and tries to open the file. If the file cannot be opened, print an error message and return a non-zero exit code. If it can be opened, print the *last 3* lines to the screen before exiting with a zero exit code.
3. In a file named `out.c` write a program that prompts the user for a file name. If that file already exists, exit with an error message and a non-zero exit code. If the file does not exist, create it and write the numbers 1 through 100, one number per line, to the file.

### Exercise 2. Arrays — (30 points):

Write a program in the file `bounds.c` in your lab repository that does the following:

1. Ask the user for the size of the array  $n$ , verifying that the number they supply is valid
2. Dynamically allocate an array to hold  $n$  floats
3. Set the value of each element of the array in a loop according to the formula:

$$1 + i^2 + \frac{i^3}{3}$$

where  $i$  is the index of the array element

4. Ask the user for an element to retrieve, verifying that it is a valid number (an integer between 1 and  $n$ , inclusive)
5. Retrieve the corresponding element (remember that indices start at zero and go to  $n-1$ )
6. Print the value retrieved and exit with a zero exit code

The maximum allowed value is  $n = 250$  and the minimum value is  $n = 1$ . If at any point an invalid input is received, print a descriptive error message and exit with a non-zero return code. Be extra careful with checking the array bounds (don't try to read the 12th element of a 10 element array) and that an index counts from zero.

*Note:*

Many useful mathematics functions are included in the C programming language, but are not loaded by default. For this exercise you will want the "pow" exponential power functions. Make sure to use the man page (man pow) to select the correct version of the function for your program. To use these math functions you must do two things:

- Include the math header file at the beginning of your code: `#include <math.h>`
- Tell the compiler that you want the math function linked in by adding the flag `-lm` to your gcc invocation

We will explore header files and compiler/linker options more in future labs.

**Exercise 3.** A First Foray Into Matrices — (30 points):

In the file `shell.c` implement the following algorithms for creating and printing a spiral matrix.

**Program Input:** An integer  $n$ , where  $1 \leq n \leq 100$

**Program Output:** For a given integer  $n$ , your program will output to a file named `ans.out` a  $n \times n$  matrix such that all the integer numbers between 1 to  $n^2$  are stored in the matrix following the pattern in the example output below:

Input: 1

Output:

1

Input: 2

Output:

1 2

4 3

Input: 3

Output:

7 8 9

6 1 2

5 4 3

Input: 4

Output:

7 8 9 10

6 1 2 11

5 4 3 12

16 15 14 13

Input: 5

Output:

21 22 23 24 25



```

20  7  8  9 10
19  6  1  2 11
18  5  4  3 12
17 16 15 14 13

```

The 4 main stages of the program runtime are listed below with hints and algorithms:

1. Accept and validate input:

This is very similar to what you have been doing in lab 4 and the previous exercises. Make sure only integers within the valid range ( $1 - -100$ ) are accepted.

2. Allocate the array:

Follow the example in section [5.3.2](#) to dynamically allocate an array that is  $n$  by  $n$  in size.

3. Fill the array:

To fill this array we will follow a procedure of three nested for-loops, one to track the numbers we are filling,  $1 - -n^2$  and two to track the 2D position in the matrix. Use the following algorithm to complete this portion of the exercise

Variables:

x: current x position

y: current y position

d: current direction

0: east

1: south

2: west

3: North

c: counter

s: Direction Length

Origin point:

```
x = floor(n/2) - 1
```

```
y = floor(n/2) - 1
```

```

for (k from 1 to n - 1) {
  for (j = 0; j<(k<(n-1)?2:3); j++){ <- Can you figure out what this does?
    for (i from 0 to s){
      save "c" to the matrix position x, y
      c++;
      switch (d)
      {
        case 0: y = y + 1; break;
        case 1: x = x + 1; break;
        case 2: y = y - 1; break;
        case 3: x = x - 1; break;
      }
    }
    d = (d+1)%4 Note this is "modulo division"
  }
}

```

```

    }
    s = s + 1
}
}

```

#### 4. Format output:

An important component of the output format is the column alignment. Notice in the sample outputs that every column is right-aligned with one space between each column. The `printf` format includes a 'fixed width' parameter to ensure each number uses the same number of spaces regardless of the number of digits. To ensure this is consistent we must first determine the maximum number of digits required. Since the maximum number will always be  $n^2$ , you can simply set a "number length" variable based on the value of  $n$ . To actually print the fixed width format based on the variable, use the format `printf("%*d", width, value);`, where the `*` indicates that the first variable passed will indicate the number of columns to use. Finally, to print the entire matrix, use nested for-loops. The outer for-loop is for the rows from  $1 - n$  while the inner for loop prints the columns. The inner for-loop will print 1 number at a time, using the index variable of the for loops to access the matrix. After that inner loop has printed every column in the row it will end, you will print a single newline, then the outer loop will end and advance to the next row. Try printing to the screen first but remember to switch your program to output to the `ans.out` file before submission.

### 5.4.1 Report

Your report this week is worth 30 points and should contain the following sections, each worth 10 points (use the `\LaTeX` section command):

**Problem:** State the problem you are trying to solve (good engineering practice)

**Design:** How did you structure the program to solve the problem? Use the verbatim environment (i.e. `\begin{verbatim}` and `\end{verbatim}`) to present your pseudo code for your program. Do not repeat algorithms given in the exercise, but instead note where those building blocks are placed and describe the logic that binds them together.

**Complexity:** What is the time complexity of your algorithm? How is the computation time related to the size of your input matrices? (Look at the previous lab regarding Big O Notation if you get stuck.)

Remember that your report must be in a file called `main.tex` in a directory called `report` and must build without errors.

### 5.4.2 Lab Submission

*Warning:* As stated in the syllabus, programs and reports that do not compile will be awarded zero points!

If you having issues with your lab work, utilize your resources (lecture notes, text books, lab manuals, personal internet research) and contact a TA for assistance if you are still unable to resolve your issue.

When you are finished with your lab, follow this procedure to turn in your work for grading:

1. Ensure all 3 exercises have correctly named files in your lab directory
2. Ensure each of the 5 source files compile without errors
3. Ensure that your report is named correctly, is in the correct directory, and builds without errors
4. Use `git status` to verify all the work you want to be graded has been committed to the develop branch
5. Use `git push` to update GitHub with all your changes
6. Open a pull request on GitHub
7. **DO NOT** click the "close and merge" button - your PR should remain open for the autograder to find

Congratulations, you are 35.71428571% of the way though 361!

## 6 | Debugging with gdb

*Reference:*

**Introduction to Scientific and Technical Computing:**

Chapter 5: Debugging with gdb

Chapter 10: Prototyping

**C Programming Language, 2nd Edition:**

[Chapter 7– Pointers and Arrays](#)

Purpose: To better understand how memory is managed during runtime and provide tools for debugging code that can be difficult or impossible to test with simple print statements.

### Objectives

- Debugging with GDB
- Memory: malloc and free

### 6.1 Automatic Memory

When compiling, gcc determines memory requirements for each section of the program. When the program is run, instructions placed by the compiler request memory for your variables from the operating system.

For example, when you declare a simple int variable, where is it stored? When you run the program, that variable is given an address by the operating system. We can request that address be printed out using the "&" operator, called the reference operator. We could interpret the line of code:

```
printf("Address: %u", &var);
```

as "print the address of var" where "&var" is the address of var, instead of the value.

You've seen this before when using scanf. Scanf requires a memory address to save it's results, so we have been using the & syntax to pass it the address of the memory we had reserved.

Working directly with memory addresses like this opens up several entire classes of bugs. One of the most common is the "seg-fault" that you may have seen by now. A seg-fault is caused by your program trying to read or write to memory your program isn't allowed to touch - so it is killed by the operating system.

Catching these bugs can be hard or even impossible to find with simple print statements - in this lab we introduce tools and techniques for more sophisticated debugging.

## 6.2 Compile-Time Debugging

In this section, an example is shown to illustrate that the compiler does not always inform the errors raised in the program. The example is shown in the figure below. Although it is a python-related error, the mechanisms behind it should work for all kinds of programming languages such as C.

Here are the common rules for Compiler Debugging:

1. Read the last line of the error first. This may give hints.
2. Scroll upwards through the bug report until you find the first error listed.
3. Read the entire bug report top to bottom.
4. **Only try to fix one error.** All of the other errors may stem from that one! So the best strategy is to only ever fix a single error, then re-compile to see if any of the others are still there.

### 6.2.1 More Warnings!

When debugging, the compiler can help by adding more warnings and errors to help you spot code that might be causing undesired behavior. The following flags, when added to your compiler, will make it emit many more warnings. Remember that a warning doesn't mean something is bad, just suspicious and you should take a deeper look at the code to ensure it really is doing what you intended. To compile with more warnings when running `gcc` try using:

```
-std=c99 -pedantic -Wall -Werror
```

And there are even more where that came from. See the GCC manual, google, or the TA for more.

## 6.3 GDB Basics

This section introduces the basic and most commonly used GDB commands via small examples. A good GDB tutorial is also pointed to <https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf>, and you may find more useful information by googling.

**Running example.** We use the following example to explain how to use GDB in a common way. The C program listed below is one kind of implementation to compute the factorial

number for a given integer number. (Recall that for an integer  $n$ , the corresponding factorial number is  $n! = n * (n - 1) * \dots * 1$ .) This source is also in your lab repo as `fac_1.c`:

```
/*
 * Compute the factorial number for a given integer
 */

#include<stdio.h>

long factorial(int n); /*Declare factorial function so we can define it below main*/

int main(void) {
    int n = 0;           /*n is the input integer*/
    printf ("Please input a integer:\n");
    scanf ("%d", &n);    /*read in n from the command line*/
    long val=factorial(n); /*declare 'val' and assign it to the factorial of n*/
    printf ("%ld\n", val); /*print 'val' to stdout*/
    return 0;           /*terminate without an error*/
} /*end main*/

/* function factorial:
   Goal: compute the factorial of a given input integer n
   Output: a long integer containing the factorial
*/
long factorial(int n) {
    long result = 1; /*declare result and initialize to 1*/
    while(n-->0) {   /*loop from n down to 0*/
        result *= n; /*accumulate the result in n*/
    } /*end while*/
    return result;   /*result should contain the factorial when the loop terminates*/
} /*end factorial*/
```

Unfortunately, the program above returns the result incorrectly. Compile the following code, and run it with an input, e.g. 1, 2 or 3. You will receive the error information as follows:

```
[~]$ ./a.out
Please input a integer:
1
Segmentation fault (core dumped)
[~]$ ./a.out
Please input a integer:
2
Segmentation fault (core dumped)
[~]$ ./a.out
Please input a integer:
3
Segmentation fault (core dumped)
```

Examining this error, a first instinct may be to check the code line by line to locate the statements which cause the error. This is a valid approach, and sometimes the best, for simple and VERY small programs. In fact, if you already have fair experience programming, you can easily find the bug in the code above. However, this is not a scalable way to fix the bugs in your program. Imagine trying to fix the bug for a program with more than one thousand lines. Are you still expecting to check the codes line by line to locate the bug? This is why debugging tools such as GDB are so important for programming. Debugging is not a crutch: It is one of the most important steps in software development.

**Enable GDB debugging for your program.** When compiling the program with GCC, use the “-g” flag. In the example above (assume the C file name is “fab.c”), compiling for debugging GDB looks like:

```
gcc fab.c -g
```

**Start Debugging.** Type the command “gdb ./a.out” in your terminal, and a new prompt shows similar information as the following (tested on Remote Server 1):

```
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-100.el7_4.1
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/*****/a.out...done.
(gdb)
```

Now you can start to debug your “a.out” program. Notably, if you need to provided command line arguments to the “a.out” program, use: “gdb -args ./a.out [FLAGS]”.

**Set/delete the breakpoints.** Before running the program “a.out” in the debugging mode, we want to set a "breakpoint" which will cause our program to pause. Without this, our program will run as normal without the opportunity to debug. Recall that the “main” function is the entry to the program, so we can set the first breakpoint as follows.

```
(gdb) break main
Breakpoint 1 at 0x4005f5: file fab.c, line 7.
```

Another two options to set up the breakpoints in GDB are via line number or file name:line number. Examples are as follows:

```
(gdb) break 8
Breakpoint 2 at 0x4005fc: file fab.c, line 8.
(gdb) break fab.c:9
Breakpoint 3 at 0x400606: file fab.c, line 9.
```

If there are more than one source file in your program, the last one should be the best option. To delete a breakpoint in GDB, the command “delete” is used. For example,

```
(gdb) info break
Num      Type           Disp Enb Address           What
1        breakpoint     keep y   0x00000000004005f5 in main at fab.c:7
2        breakpoint     keep y   0x00000000004005fc in main at fab.c:8
3        breakpoint     keep y   0x0000000000400606 in main at fab.c:9
(gdb) delete 2
(gdb) delete 3
(gdb) info break
```

```
Num      Type           Disp Enb Address           What
1        breakpoint     keep y   0x00000000004005f5 in main at fab.c:7
```

The “info break” command is used to print out all breakpoints set in GDB, and the “delete N” command will delete the N-th breakpoint.

**Run the program.** The “run” command is used to run the program to be debugged in GDB. And it will stop when one of the breakpoints is hit. If there is no breakpoint set up, or none of the breakpoints are met during the execution of the program, the program is executed as normal.

```
(gdb) run
Starting program: /home/****/./a.out
```

```
Breakpoint 1, main () at fab.c:7
7      int n = 0;
```

**Run the program line by line.** Once the program is executed by “run” and stopped by a breakpoint, the “next” command is used to execute the program line by line. For example,

```
(gdb) next
8      printf ("Please input a integer:\n");
(gdb) next
Please input a integer:
9      scanf ("%d", n);
```

There is another command “step” which has similar functionality to “next”. It is suggested to google the difference between “next” and “step”.

**Identify the first error.** By executing one more “next” command in the code above, the first error of the program can be caught.

```
(gdb) next
8      printf ("Please input a integer:\n");
(gdb) next
Please input a integer:
9      scanf ("%d", n);
(gdb) next
1
```



Program received signal SIGSEGV, Segmentation fault.  
0x00007ffff7a72122 in \_\_GI\_\_IO\_vfscanf () from /lib64/libc.so.6

So we learn that it is the “scanf” function that causes the error. It seems that the function is not used correctly. After fixing this problem, we have the revised code as follows (fac\_2.c):

```
/*
 * Compute the factorial number for a given integer
 */

#include<stdio.h>

long factorial(int n); /*Declare factorial function so we can define it below main*/

int main(void) {
    int n = 0;          /*n is the input integer*/
    printf ("Please input a integer:\n");
    scanf ("%d", &n);    /*read in n from the command line*/
    long val=factorial(n); /*declare 'val' and assign it to the factorial of n*/
    printf ("%ld\n", val); /*print 'val' to stdout*/
    return 0;           /*terminate without an error*/
}

/* function factorial:
   Goal: compute the factorial of a given input integer n
   Output: a long integer containing the factorial
*/
long factorial(int n) {
    long result = 1; /*declare result and initialize to 1*/
    while(n-->0) {    /*loop from n down to 0*/
        result *= n; /*accumulate the result in n*/
    } /*end while*/
    return result;    /*result should contain the factorial when the loop terminates*/
} /*end function factorial*/
```

Now re-compile the codes again and test the program with the input 1, 2 or 3 again.

```
[ ~]$ ./a.out
Please input a integer:
1
0
[~]$ ./a.out
Please input a integer:
2
0
[~]$ ./a.out
Please input a integer:
3
```

0

The outputted factorial numbers turn out to be incorrect. So we need to debug the program again to locate the problem.

**Watch a variable in the program.** In the source code, the factorial number is computed by the function “factorial”, in which the value of  $n$  relates to the final result. As a result, we may want to see how the value of  $n$  updates in the function. First we set the breakpoint to the beginning of the function “factorial” then ask GDB to print the value of the variable “n”:

```
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-100.el7_4.1
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/*****/a.out...done.
(gdb) break factorial
Breakpoint 1 at 0x40064e: file fab.c, line 17.
(gdb) run
Starting program: /home/*****/./a.out
Please input a integer:
1
```

```
Breakpoint 1, factorial (n=1) at fab.c:17
17      long result = 1;
(gdb) print n
$1 = 1
```

Another way is to use the “watch” command to monitor the value changes of  $n$ :

```
(gdb) watch n
Hardware watchpoint 2: n
(gdb) continue
Continuing.
Hardware watchpoint 2: n

Old value = 1
New value = 0
0x000000000400672 in factorial (n=0) at fab.c:18
18      while(n--)
(gdb) continue
Continuing.
Hardware watchpoint 2: n
```

```
Old value = 0
New value = -1
```

In the lines above, we first set up watch on the variable  $n$ , whose id is 2. Then the “continue” command is used to execute the program until the value of the watched variable  $n$  changes: from 1 to 0. Continue again and the value of  $n$  changes again from 0 to -1. But recall that the computation of a factorial number,  $n$  must be greater than 0. As a result, we find that the problem is caused by line 18, where `while(n --)` allows the value of  $n$  to be 0 inside the while loop. Now we revise the codes as follows (`fac_3.c`):

```
/*
 * Compute the factorial number for a given integer
 */

#include<stdio.h>

long factorial(int n); /*Declare factorial function so we can define it below main*/

int main(void) {
    int n = 0;          /*n is the input integer*/
    printf ("Please input a integer:\n");
    scanf ("%d", &n);    /*read in n from the command line*/
    long val=factorial(n); /*declare 'val' and assign it to the factorial of n*/
    printf ("%ld\n", val); /*print 'val' to stdout*/
    return 0;           /*terminate without an error*/
} /*end main*/

/* function factorial:
   Goal: compute the factorial of a given input integer n
   Output: a long integer containing the factorial
*/
long factorial(int n) {
    long result = 1; /*declare result and initialize to 1*/
    while(n>0) {     /*loop from n down to 0*/
        result *= n; /*accumulate the result in n*/
        n--;        /*make sure n is decremented so that our 'while' loop terminates*/
    } /*end while*/
    return result;   /*result should contain the factorial when the loop terminates*/
} /*end function factorial*/
```

Re-compile and test the program again. Is the program running properly now?

**Summary.** In the example above, we introduced commonly used GDB commands such as “run, break, next, step, watch” and “continue”, all of which are powerful and important commands for debugging. Only part of the usage of these commands is shown in the example, and we suggest to learn more about their usage via google. In fact, you will find detailed commands and their usage by googling “GDB cheat sheet”. We omit the introduction to the

“GDB cheat sheet” here but show an explicit example to illustrate some of the frequently used GDB commands instead, hoping to make you learn GDB more easily in a practical way. In Lab 5, you were required to write a program to generate a matrix “properly.” One goal of this section is to make you able to use GDB to help you write the correct program from Lab 5.

## 6.4 Exercises

### Exercise 1.

GDB — (10 points):

The code `broke.c` in your lab repo segfaults. Use the debugging skills introduced above to correct the code.

### Exercise 2.

Report — (30 points):

Your report this week is worth 30 points and should be written before completing exercises 3 and 4. In the following exercises you will be programming a Maclaurin series evaluator, but following the principle that "a day of coding will save you half an hour of design work" you should take this opportunity to plan your attack (and receive points for your effort). (use the `LATEX` section command):

**Algorithm & Design (15 points):** How will your program solve the problem? Use the example algorithm block given below to print your pseudo code. Think before you start to code!)

**Labeled Loops (10 points):** What does each loop do? What changes and what is constant?

**Complexity (5 points):** Is the algorithm efficient? How will the time taken by the computer increases as the input increases?

An example pseudo code and the `LATEX` to generate it:

---

**Algorithm 1** Algorithm for finding the factorial

---

```
n = ? number to factorial, passed as argument(int)
fact=1 answer and working memory (long)
if n not >= 0 then
    Invalid n, exit with error
end if
```

▷ Required variables:

```
while n > 0 do
    fact = fact * n
    n = n - 1
end while
Return n
```

---

▷ Loop over decreasing n from initial n to 1

```

\usepackage{algpseudocode}
\usepackage{algorithm}

\begin{algorithm}
  \caption{Algorithm for finding the factorial}

  \begin{algorithmic}
    \Statex \Comment {Required variables:}
    \State n = ? number to factorial, passed as argument(int)
    \State fact=1 answer and working memory (long)

    \If{ n not >= 0}
      \State Invalid n, exit with error
    \EndIf

    \Statex \Comment {Loop over decreasing n from initial n to 1}
    \While{ n > 0 }
      \State fact = fact * n
      \State n = n - 1
    \EndWhile
    \State Return n
  \end{algorithmic}
\end{algorithm}

```

Now review the requirements for exercises 3 and 4, design your program and write-up your plan as described above. Remember that your report must be in a file called `main.tex` in a directory called `report` and must build without errors.

### Exercise 3.

Code Structure — (10 points):

In a file `mac_exp.c` you will write a program that evaluates the exponential function ( $e^x$ ) using the Maclaurin series. Exercise 3 is to write the skeleton of the code (everything but the math) while exercise 4 will be to finish the implementation. To receive full credit for exercise 3 your code should:

- Be in a correctly named file
- Compile without errors
- Prompt the user for the  $x$  to evaluate
- Prompt the user for a relative error
- Correctly reject invalid values of  $x$
- Print the correct output format with placeholder values as demonstrated below

Output format:

After `itr` terms in the series, `exp(x)` is approx. `ans` with an error of `err`  
 The values `itr`, `x`, `ans`, and `err` will be replaced with values in exercise 4.

#### Exercise 4.

Maclaurin Series — (50 points):

The Maclaurin series for this function is

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

your code should ask for a real value of  $x$  and an error as a convergence criterion. The termination criterion of this method is determined by the change introduced by the addition of another term in the series. For example, if the addition of the third term in the series changes the value by 0.001 and the user-specified error was 0.05 then the code should terminate.

Use doubles for all float variables ( $x$ ,  $err$ , the answer).

Make sure to fill in the 4 specified variables in the output above.

#### 6.4.1 Lab Submission

*Warning:* As stated in the syllabus, programs and reports that do not compile will be awarded zero points!

If you having issues with your lab work, utilize your resources (lecture notes, text books, lab manuals, personal internet research) and contact a TA for assistance if you are still unable to resolve your issue.

When you are finished with your lab, follow this procedure to turn in your work for grading:

1. Ensure all exercises have a correctly named files in your lab directory
2. Ensure each of the source files compile without errors
3. Ensure that your report is named correctly, is in the correct directory, and builds without errors
4. Use `git status` to verify all the work you want to be graded has been committed to the develop branch
5. Use `git push` to update GitHub with all your changes
6. Open a pull request on GitHub
7. **DO NOT** click the "close and merge" button - your PR should remain open for the autograder to find

Congratulations, you are 42.85714286% of the way though 361!

# 7 | All About Pointers

*Reference:*

**Introduction to Scientific and Technical Computing:**

Chapter 6: Makefiles, Libraries, and Linking

Chapter 17: Testing and Verification

**C Programming Language, 2nd Edition:**

Chapter 4 – Functions and Program Structure

Chapter 5 – Pointers and Arrays

Purpose: Provide tools and practice with pointers

## Objectives

- Use tooling to verify safe memory operation
- Utilize pointers to functions
- Forward declare a function in another source file
- Take a first shot at programming numerical methods

## 7.1 Memory Layout of C Programs

So far we have seen how to declare basic data types such as `int`, `double`, etc. in our C program. The way we have been declaring them (with a syntax like other languages such as MATLAB), puts these variables in a special memory area called the *stack*. We also did not have to worry about how and where these variables were stored in memory. A typical memory layout of a C program is shown in Figure 7.1.

### 7.1.1 The Text Segment

The text segment contains the machine code of the compiled C program. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. The text segment of an executable object file is often a read-only segment that prevents a program from being accidentally modified.

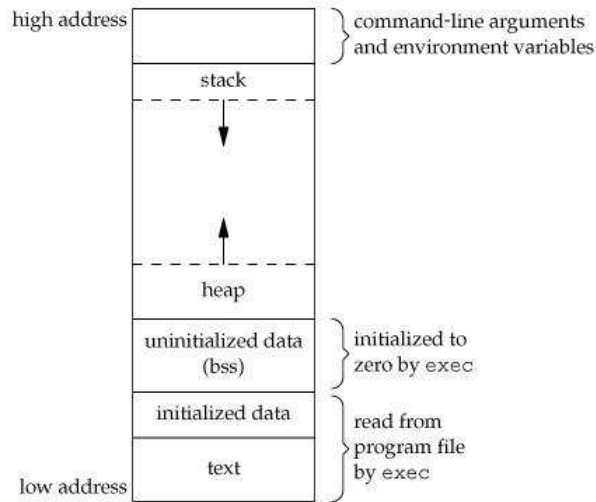


Figure 7.1: Typical memory layout of a C program.

### 7.1.2 The Data Segment

The **global**, **static**, **const** variables used within a program are stored in the data segment. Depending on whether these variables are initialized, the data segment has two parts: initialized data segment, and uninitialized data segment (bss). Each of these segments can be further classified into **read-only** or **read-write**. Let's see how this works using an example.

```
#include <stdio.h>

char c[] = "Aerospace Engineering"; /* global variable stored in initialized
                                     data segment in read-write area */

const char s[] = "C Programming";   /* global variable stored in initialized
                                     data segment in read-only area */

char x; /* uninitialized global variable stored in bss */

int main(void) {
    static int i = 11; /* static variable stored in
                       initialized data segment */

    static int j;      /* uninitialized static variable
                       stored in bss */

    return 0; /*end without error*/
} /*end main*/
```

### 7.1.3 The Stack

The stack is a special region of your computer's memory that stores temporary variables created by each function (including the `main()` function). The stack is a "LIFO" (last in,



first out) data structure managed by the CPU. Every time we declare a new variable in a function, it is "pushed" onto the stack. Then every time a function exits, all of the variables pushed onto the stack by that function, are freed (that is to say, they are deleted). Once a stack variable is freed, that region of memory becomes available for other stack variables in other functions.

The advantage of using the stack to store variables, is that memory is managed for you. For example, you don't have to worry about allocating 4 bytes for `int` variable, or free the memory used by a `double` variable in a function you call. Because the CPU organizes stack memory so efficiently, reading from and writing to stack variables is very fast.

A key to understanding the stack is the notion that when a function exits, all of its variables are popped off of the stack (and hence lost forever). Thus stack variables are **local** in nature.

A common bug in C programming is attempting to access a variable that was created on the stack inside some function, from a place in your program outside of that function (i.e. after that function has exited).

Another "feature" of the stack to keep in mind, is that there is a limit (varies with OS) on the size of variables that can be stored on the stack. Any guesses why this is a feature?

To summarize the stack:

1. The stack grows and shrinks as functions push and pop local variables
2. There is no need to manage the memory yourself, variables are allocated and freed automatically
3. The stack has size limits
4. Stack variables only exist while the function that created them is running

#### 7.1.4 The Heap

The heap is a region of your computer's memory that is NOT managed automatically for you, and is not (as tightly) managed by the CPU. It is a more free-floating region of memory, and is much larger than the stack. To allocate memory on the heap for a variable, you must use `malloc()` or `calloc()` (next section), which are built-in C functions. Once you have allocated memory on the heap, you are responsible for deallocating that memory using `free()` once you don't need it any more. If you fail to do this, your program will have what is known as a *memory leak*. A memory leak is a programming error (not reported by the compiler) in which memory is set aside on the heap, and if not freed, is not available to other processes. As we will see in the debugging section, there is a tool called `valgrind` that can help you detect memory leaks.

Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer). Heap memory is slightly slower to be read from and written to, because one has to use pointers to access memory on the heap. We will talk about pointers shortly. Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables can be accessed by any function

with a pointer to that variable, letting you work with the same memory from multiple functions.

When should you use the heap, and when should you use the stack? If you need to allocate a large block of memory (e.g. a large array), and you need to keep that variable around a long time (like a global), then you should allocate it on the heap. If you are dealing with relatively small variables that only need to persist as long as the function using them is alive, then you should use the stack, it's easier and faster. If you need variables like arrays that can change size dynamically (e.g. arrays that can grow or shrink as needed) then you will likely need to allocate them on the heap, and use dynamic memory allocation functions like `malloc()`, `calloc()`, `realloc()` and `free()` to manage that memory manually.

### 7.1.5 The `size` command

A useful Linux command to monitor memory usage for text and data (initialized and uninitialized) is `size`. Consider the following C program:

```
#include <stdio.h>

int main(void) {
    return 0; /*end without error*/
} /*end main*/
```

Save the program as `sample.c` and compile using `gcc sample.c`. To view the memory usage, run `size a.out`. A sample output is

text	data	bss	dec	hex filename
1127	540	4	1671	687 a.out

Add a global uninitialized integer variable to the above program (`int global;`) just before the main function. `size` now returns

text	data	bss	dec	hex filename
1127	540	12	1679	68f a.out

The size of the `bss` increases from 4 bytes to 12 bytes, meaning that an `int` variable on the linux remote servers occupies 8 bytes. This matches up with the output of the `"sizeof"` function we've used in the past. Refer to the `man` page of the `size` command to learn more about what each of the fields mean in the output.

## 7.2 Allocating and Managing Memory: beyond `malloc` and `free`

### 7.2.1 `calloc`

In the prior labs, `malloc` was used to request space on the heap and return a pointer to that memory. Another function for accomplishing this is `calloc`. The format of `calloc` usage is: `ptr = (cast-type*)calloc(n, element-size);`

Notice that this is similar to the usage of `malloc` but instead of providing one argument, the number of bytes, we provide 2 – the number of "things" we need to store (`n` in the above format) and the element size in bytes. Using `calloc` can make it harder to create

subtle bugs relating to array size. The tradeoff is using `calloc` can be slower for large data structures because the memory is zeroed-out (has a value of all 0 written to it) when it is reserved, instead of `malloc` that just returns a pointer to reserved memory without assigning any value. For our purposes, the following `malloc` and `calloc` are roughly equivalent.

```
ptr = (int*) malloc(100 * sizeof(int));
ptr = (int*) calloc(100, sizeof(int));
```

### 7.2.2 `realloc`

`Realloc` allows a portion of heap memory to be resized – if the new requested size is larger, memory will be added and if it is smaller, the unneeded space will be freed. When necessary, this will be explained further – for now you should simply be aware of its existence. Check the man page for more information.

## 7.3 Introduction to Valgrind

In this section, we introduce a very useful tool suite for debugging and profiling Linux programs, Valgrind, to facilitate C programming in terms of pointers. The tool suits are already installed on the college Linux servers, and more information can be found on its official website: <http://valgrind.org/info/>. If the department servers claim the command is not found, try a college of engineering server.

The Valgrind tool suite provides a number of debugging and profiling tools that help you make your programs faster and more correct. The most popular of these tools is called Memcheck. It can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behavior. Memcheck is extremely useful to learn the pointers in C. The rest of this guide gives the minimum information you need to start detecting memory errors in your program with Memcheck. For full documentation of Memcheck and the other tools, please read the Valgrind user manual at its website.

### 7.3.1 Running example

We use the following source code as the example in this section. This is contained in the lab repository as `val_test.c`

```
#include <stdlib.h>

/* function f
   What is the purpose of this function?
*/
void f(void) {
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;          /* problem 1: heap block overrun */
} /*end f*/           /* problem 2: memory leak -- x not freed*/

int main(void) {
```

```

    f();          /*call function f to allocate our memory*/
    return 0; /*return without error*/
} /*end main*/

```

### 7.3.2 Prepare the program

Like using GDB to debug your program, the “-g” option is also needed if you want to use Valgrind to detect your program. For example, assume the file name of the source codes above is “test.c”, then the compiling command to enable Valgrind is as follows:

```
gcc test.c -g
```

The output program is “a.out”.

### 7.3.3 Running your program under Memcheck

For the example above, the following command is suggested to run the program with Valgrind:

```
valgrind --leak-check=yes ./a.out
```

By default, Valgrind displays the output through the terminal. If you want to store the output in a file, e.g. “valgrind.out”, use the following command:

```
valgrind --leak-check=yes ./a.out 2>valgrind.out
```

A test run in Linux server 1 outputs the information as follows:

```

==5893== Memcheck, a memory error detector
==5893== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==5893== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==5893== Command: ./a.out
==5893==
==5893== Invalid write of size 4
==5893==    at 0x40054B: f (test.c:6)
==5893==    by 0x40055B: main (test.c:11)
==5893== Address 0x51f9068 is 0 bytes after a block of size 40 alloc'd
==5893==    at 0x4C29BE3: malloc (vg_replace_malloc.c:299)
==5893==    by 0x40053E: f (test.c:5)
==5893==    by 0x40055B: main (test.c:11)
==5893==
==5893==
==5893== HEAP SUMMARY:
==5893==    in use at exit: 40 bytes in 1 blocks
==5893== total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==5893==
==5893== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==5893==    at 0x4C29BE3: malloc (vg_replace_malloc.c:299)
==5893==    by 0x40053E: f (test.c:5)
==5893==    by 0x40055B: main (test.c:11)
==5893==

```

```

==5893== LEAK SUMMARY:
==5893==    definitely lost: 40 bytes in 1 blocks
==5893==    indirectly lost: 0 bytes in 0 blocks
==5893==    possibly lost: 0 bytes in 0 blocks
==5893==    still reachable: 0 bytes in 0 blocks
==5893==    suppressed: 0 bytes in 0 blocks
==5893==
==5893== For counts of detected and suppressed errors, rerun with: -v
==5893== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)

```

In the Valgrind output, most error messages look like the following, which describes problem 1, the heap block overrun:

```

==5893== Invalid write of size 4
==5893==    at 0x40054B: f (test.c:6)
==5893==    by 0x40055B: main (test.c:11)
==5893== Address 0x51f9068 is 0 bytes after a block of size 40 alloc'd
==5893==    at 0x4C29BE3: malloc (vg_replace_malloc.c:299)
==5893==    by 0x40053E: f (test.c:5)
==5893==    by 0x40055B: main (test.c:11)

```

Here are some noteworthy things about the message.

- The 5893 is the process ID; it's usually unimportant.
- The first line ("Invalid write...") tells you what kind of error it is. Here, the program wrote to some memory it should not have due to a heap block overrun.
- Below the first line is a stack trace telling you where the problem occurred. Reading them from the bottom up can help.
- The code addresses (eg. 0x40054B) are usually unimportant, but occasionally useful for tracking down weirder bugs.
- Some error messages have a second component which describes the memory address involved. This one shows that the written memory is just past the end of a block allocated with malloc() on line 5 of test.c.

It's worth fixing errors in the order they are reported, as later errors can be caused by earlier errors. Failing to do this is a common cause of difficulty with Memcheck. Memory leak messages look like this:

```

==5893== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==5893==    at 0x4C29BE3: malloc (vg_replace_malloc.c:299)
==5893==    by 0x40053E: f (test.c:5)
==5893==    by 0x40055B: main (test.c:11)

```

The stack trace tells you where the leaked memory was allocated. Memcheck cannot tell you why the memory leaked, unfortunately. (Ignore the "vg\_replace\_malloc.c", that's an implementation detail.) There are several kinds of leaks; the two most important categories are:

- “definitely lost”: your program is leaking memory – Try to fix it!

- “probably lost”: your program is leaking memory, unless you’re doing funny things with pointers (such as moving them to point to the middle of a heap block).

## 7.4 A Summary of Errors Valgrind can report

Despite considerable sophistication under the hood, Memcheck can only really detect two kinds of errors: use of illegal addresses, and use of undefined values. Nevertheless, this is enough to help you discover all sorts of memory-management nasties in your code. This section presents a quick summary of what error messages mean.

### 7.4.1 Illegal read / write errors

For example:

```
Invalid read of size 4
  at 0x40F6BBCC: (within /usr/lib/libpng.so.2.1.0.9)
  by 0x40F6B804: (within /usr/lib/libpng.so.2.1.0.9)
  by 0x40B07FF4: read_png_image__FP8QImageIO (kernel/qpngio.cpp:326)
  by 0x40AC751B: QImageIO::read() (kernel/qimage.cpp:3621)
  Address 0xBFFFFFF0E0 is not stack'd, malloc'd or free'd
```

This happens when your program reads or writes memory at a place which Memcheck reckons it shouldn’t. In this example, the program did a 4-byte read at address 0xBFFFFFF0E0, somewhere within the system-supplied library `libpng.so.2.1.0.9`, which was called from somewhere else in the same library, called from line 326 of `qpngio.cpp`, and so on.

Memcheck tries to establish what the illegal address might relate to, since that’s often useful. So, if it points into a block of memory which has already been freed, you’ll be informed of this, and also where the block was free’d at. Likewise, if it should turn out to be just off the end of a malloc’d block, a common result of off-by-one-errors in array subscripting, you’ll be informed of this fact, and also where the block was malloc’d.

In this example, Memcheck can’t identify the address. Actually the address is on the stack, but, for some reason, this is not a valid stack address – it is below the stack pointer.

Note that Memcheck only tells you that your program is about to access memory at an illegal address. It can’t stop the access from happening. So, if your program makes an access which normally would result in a segmentation fault, your program will still suffer the same fate – but you will get a message from Memcheck immediately prior to this. In this particular example, reading junk on the stack is non-fatal, and the program stays alive.

### 7.4.2 Use of uninitialized values

For example:

```
Conditional jump or move depends on uninitialised value(s)
  at 0x402DFA94: _IO_vfprintf (_itoa.h:49)
  by 0x402E8476: _IO_printf (printf.c:36)
  by 0x8048472: main (tests/manuel1.c:8)
  by 0x402A6E5E: __libc_start_main (libc-start.c:129)
```

An uninitialised-value use error is reported when your program uses a value which hasn't been initialised – in other words, is undefined. Here, the undefined value is used somewhere inside the `printf()` machinery of the C library. This error was reported when running the following small program:

```
int main(void) {
    int x;                /*Declare integer x*/
    printf ("x = %d\n", x); /*print our int*/
    return 0;             /*terminate normally*/
} /*end main*/
```

It is important to understand that your program can copy around junk (uninitialised) data to its heart's content. Memcheck observes this and keeps track of the data, but does not complain. A complaint is issued only when your program attempts to make use of uninitialised data. In this example, `x` is uninitialised. Memcheck observes the value being passed to `_IO_printf` and thence to `_IO_vfprintf`, but makes no comment. However, `_IO_vfprintf` has to examine the value of `x` so it can turn it into the corresponding ASCII string, and it is at this point that Memcheck complains.

Sources of uninitialised data tend to be:

- Local variables in procedures which have not been initialised, as in the example above.
- The contents of malloc'd blocks, before you write something there.

### 7.4.3 Illegal frees

For example:

```
Invalid free()
at 0x4004FFDF: free (vg_clientmalloc.c:577)
by 0x80484C7: main (tests/doublefree.c:10)
by 0x402A6E5E: __libc_start_main (libc-start.c:129)
by 0x80483B1: (within tests/doublefree)
Address 0x3807F7B4 is 0 bytes inside a block of size 177 free'd
at 0x4004FFDF: free (vg_clientmalloc.c:577)
by 0x80484C7: main (tests/doublefree.c:10)
by 0x402A6E5E: __libc_start_main (libc-start.c:129)
by 0x80483B1: (within tests/doublefree)
```

Memcheck keeps track of the blocks allocated by your program with malloc, so it can know exactly whether or not the argument to free is legitimate or not. Here, this test program has freed the same block twice. As with the illegal read/write errors, Memcheck attempts to make sense of the address free'd. If, as here, the address is one which has previously been freed, you will be told that – making duplicate frees of the same block easy to spot.

### 7.4.4 Passing system call parameters with inadequate read/write permissions

Memcheck checks all parameters to system calls. If a system call needs to read from a buffer provided by your program, Memcheck checks that the entire buffer is addressible and has valid data, ie, it is readable. And if the system call needs to write to a user-supplied buffer,

Memcheck checks that the buffer is addressable. After the system call, Memcheck updates its administrative information to precisely reflect any changes in memory permissions caused by the system call. Here's an example of a system call with an invalid parameter:

```
#include <stdlib.h>
#include <unistd.h>
int main(void) {
    char* arr = malloc(10); /*declare a 10-character array*/
    (void) write( 1 /* stdout */, arr, 10 );
    return 0; /*terminate normally*/
} /*end main*/
```

You will get the following complaint:

```
Syscall param write(buf) contains uninitialised or unaddressable byte(s)
  at 0x4035E072: __libc_write
  by 0x402A6E5E: __libc_start_main (libc-start.c:129)
  by 0x80483B1: (within tests/badwrite)
  by <bogus frame pointer> ???
Address 0x3807E6D0 is 0 bytes inside a block of size 10 alloc'd
  at 0x4004FEE6: malloc (ut_clientmalloc.c:539)
  by 0x80484A0: main (tests/badwrite.c:6)
  by 0x402A6E5E: __libc_start_main (libc-start.c:129)
  by 0x80483B1: (within tests/badwrite)
```

The reason is because the program has tried to write uninitialised junk from the malloc'd block to the standard output.

#### 7.4.5 Overlapping source and destination blocks

The following C library functions copy some data from one memory block to another (or something similar): `memcpy()`, `strcpy()`, `strncpy()`, `strcat()`, `strncat()`. The blocks pointed to by their `src` and `dst` pointers aren't allowed to overlap. Memcheck checks for this. For example:

```
==27492== Source and destination overlap in memcpy(0xbffff294, 0xbffff280, 21)
==27492==    at 0x40026CDC: memcpy (mc_replace_strmem.c:71)
==27492==    by 0x804865A: main (overlap.c:40)
==27492==    by 0x40246335: __libc_start_main (../sysdeps/generic/libc-start.c:129)
==27492==    by 0x8048470: (within /auto/homes/njn25/grind/head6/memcheck/tests/overlap)
==27492==
```

You don't want the two blocks to overlap because one of them could get partially trashed by the copying.

### 7.5 Use Valgrind in your programming

We have introduced the minimum information for you to use Valgrind to debug your C program pointers. Please try to use this powerful tool to help you finish the current and



future labs. More information on additional checks valgrind can make, as well as help understanding its output can be found in the manual or online.

## 7.6 Function Pointers

Pointers can reference more than arrays. One example is the *function pointer*. A function pointer references a function with a specific signature (return and input types) and allows us to write code that uses a function that isn't known at compile time. A great example of this is the following function (in the lab repo as `funcp.c`):

```
#include <stdio.h>

double sum_to_100(double (*funcp)(double)) {
    double sum = 0.0;

    int i; /*declare an iterator for use in our loop*/
    /*for loop with variants i in 1..100 inclusive
       and sum, which accumulates with each iteration*/
    for(i = 1; i <=100; i++) {
        sum += funcp((float) i);
    } /*end for*/

    return sum;
} /*end sum_to_100*/

double square(double x) {
    return x * x;
} /*end square*/

int main(void) {
    double sum; /*declare a variable to accumulate our sum*/

    sum = sum_to_100(square);
    printf("Sum of x^2 from 1 to 100: %g\n", sum);

    return 0; /*terminate normally*/
} /*end main*/
```

In the `sum_to_100` function, we declared an argument of type `double`, in the form of a pointer to a function that took a `double` as an input. There is nothing special about the name "funcp", as long as it follows normal naming rules, and is preceded with an asterisk and enclosed in parentheses. Note the format in general is `return_type (ptr_name)(arg_types)`

By asserting in the definition of the `sum_to_100` function that we would provide a function that takes a `double` and returns a `double`, we can use the pointer like the function (as in `sum += funcp((float) i)`). To call `sum` we provide a function that meets these criteria (takes a `double` and returns a `double`). This is incredibly powerful for use in numerical methods and

technical computing as you will see in your lab exercises this week.

## 7.7 Project – Numerical Integration

When doing technical computing for aerospace application, integration is frequently required. Computing the analytical solution to a definite integral can be easy for a human but significantly more difficult for a computer. To get the analytical solution we must first teach the machine to see the form of the equation and all the rules you learned in calculus. This isn't impossible (I'm sure many have used Wolfram Alpha for this exact feature) but it is computationally expensive, and what about integrals without a known analytical form?

Enter numerical methods, these methods approximate the analytical result – but have two advantages over analytical methods:

- While difficult for humans, computers are very well suited for them
- They can solve integrals without known forms

In general, the computer can find an approximation arbitrarily close to the actual solution significantly faster than any other solution method, and for engineering where we can define our tolerance (i.e. we only need so many decimal places before it becomes irrelevant) this is a fantastically advantageous trade-off.

### Receiving your integrand

To allow your code to take arbitrary integrands that follow the *type signature* (pattern of input types and output type) your integration functions will take a function pointer. In the lab repository under the `example` sub-dir, there is a version of the `funcp.c` code split between two files. The file `main.c` has the square function removed. Instead, on the third line is `double ext_func(double);` This is called a *forward declaration*, it tells the compiler that "there exists some function 'ext\_func' that takes a double and returns a double".

If you look in the other file, `ext_func.c` you will find the definition of this function (Note: the name of the C file doesn't have to match the name of the function – this is just a coincidence, but the name of the function prototype in main does have to match the name of the actual function).

Compile the program by providing both files to the compiler: `gcc main.c ext_func.c`  
Run the resulting binary, it should work identically to the single file version.

For your project, the autograder will supply its own main and integrand functions and use your integration functions. All 4 of your integration functions should be in a file called `integration.c` in the base of your lab repo. Make sure you name your functions *exactly* as named in the `template/main.c` file and follow the argument order described in the template integration file.

You may test your code by copying the `template/main.c` file to the base of your lab repository, adding the forward declarations of the functions, and compiling the `main.c` and `integration.c` files together.

### 7.7.1 Exercises

Each method will be worth 40 points with the following breakdown:

**10 Points:** Correctly compiling code

**20 Points:** Correct implementation of the listed integration method

**10 Points:** Complexity analysis in your report (see the report section below)

Each should take any univariate integrand as well as the bounds of the definite integral with all numbers being of type `double`. In addition, the Gauss Quad function must take an integer specifying the order between 1 and 10 inclusive.

**Exercise 1.** Midpoint Rule (also known as the Rectangle Rule):

$$\int_a^b f(x)dx \approx (b-a)f\left(\frac{a+b}{2}\right)$$

**Exercise 2.** Simpson's  $\frac{1}{3}$ :

$$\int_a^b f(x)dx \approx \frac{(b-a)}{6} \left[ f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right]$$

**Exercise 3.** Simpson's  $\frac{3}{8}$ :

$$\int_a^b f(x)dx \approx \frac{(b-a)}{8} \left[ f(a) + 3f\left(\frac{2a+b}{3}\right) + 3f\left(\frac{a+2b}{3}\right) + f(b) \right]$$

**Exercise 4.** Gauss Quad:

$$\int_a^b f(x)dx \approx m \sum_{i=1}^n w_i f(c + mt_i)$$

Where:

$$x = c + mt \quad c = \frac{1}{2}(b+a) \quad m = \frac{1}{2}(b-a)$$

And the values of  $t$  and  $w$  can be found at [http://www.public.iastate.edu/~akmitra/aero361/design\\_web/quad.html](http://www.public.iastate.edu/~akmitra/aero361/design_web/quad.html) and  $n$  is the order. Note that the provided constants are indexed starting at 1, but C arrays begin at 0.

### 7.7.2 Report

In your report file, have one section per method above that explains the time complexity of the method. For this exploration, ignore the complexity of the integrand (assume it to be constant time) and instead focus on the question "As the bounds of integration increase, how many more lines of code are executed". Develop this expression mathematically as "lines of code" then fit the pattern to one of the complexity classes from lab 4 (i.e. does it grow linearly, exponentially, etc.). Use Big O notation to describe your programs. Don't forget to include a title page and a sources page listing the resources you accessed for the exercises. Remember that your report must be in a file called `main.tex` in a directory called `report` and must build without errors.

### 7.7.3 Lab Submission

*Warning:* As stated in the syllabus, programs and reports that do not compile will be awarded zero points!

If you have issues with your lab work, utilize your resources (lecture notes, text books, lab manuals, personal internet research) and contact a TA for assistance if you are still unable to resolve your issue.

When you are finished with your lab, follow this procedure to turn in your work for grading:

1. Ensure all 4 exercises are in the correctly named file in your lab directory
2. Ensure the source file compiles with the sample main function without errors
3. Ensure that your report is named correctly, is in the correct directory, and builds without errors
4. Use `git status` to verify all the work you want to be graded has been committed to the develop branch
5. Use `git add` to track the files you intend to push
6. Use `git commit -m "message"` to commit your changes
7. Use `git push origin develop` to update GitHub with all your changes
8. Open a pull request on GitHub
9. **DO NOT** click the "close and merge" button - your PR should remain open for the autograder to find

Congratulations, you are 50.00000000% of the way though 361!

## 8 | Makefiles, Libraries, and Linking

*Reference:*

**Introduction to Scientific and Technical Computing:**

Chapter 6: Makefiles, Libraries, and Linking

Chapter 7: Linking and Interoperability

### Objectives

- Gain experience working with more complex data structures (`struct`).
- Think about how to write code that is general and re-useable, rather than hard-coded to a particular task.
- Learn to manage compilation of multiple types of projects with `make`.
- Write an external library that is imported and used by your main program. Practice compiling these together.

### 8.1 Libraries

In lab 7, we put some functionality in a separate file and used it from main by declaring the signature of the function. This is a good first approximation of a library in C. Just as cloud computing is just a fancy way of saying somebody else's computer, libraries are just fancy names for somebody else's code. The special part about libraries is that we can use this other code, even without the source files! All that is needed is a description of the functions and variables that we want to access, like the function prototypes you used in lab 7.

#### 8.1.1 The gold standard (library)

You have already been using libraries this entire class, the standard library is a set of libraries required to be included with every C compiler. Your basic `printf` usage demonstrates how libraries work, via header files. One thing that is simpler about the standard library is that you don't need to tell the compiler where it is - that's built-in. As we begin using other libraries and writing our own, this won't always be the case.

### 8.1.2 Include Files

When you use the `#include` directive, the filename you pass as the argument is copy-pasted into the code before it is compiled. This is used to make using libraries simple. All the function prototypes and any other declarations that are needed to use the library are written into a file called a header file. We can then "include" that header file to accomplish the same thing as the function prototypes you used in lab 7 to tell main that the function exists somewhere else.

*Note:*

When including the standard library headers like `stdio.h`, you use angle brackets:

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

When including header files written by you or someone else, you use quotes: `#include "foobad.h"`

### 8.1.3 Compiling Libraries

The header file contains the forward declarations of what is available in the library, but the code itself can come in many forms. In lab 7, we just provided the source file to the compiler and let it take care of the rest. For our library usage, we will use shared objects. We will describe what this means in lab but that knowledge isn't necessary to begin using them.

To compile a library as a shared object we pass two flags to gcc: `-shared -c` which instructs the compiler to make a shared object library and to only compile it and not link it. You should name the output with the format `lib<libname>.so`, for example `libfoo.so` for a library called `foo`.

### 8.1.4 Compiling with Libraries

To compile code that uses a shared object library, we need to tell the compiler three things: where to find the headers, where to find the library and what library to include.

When you `#include "foo.h"` the compiler looks through a list of directories for a file called `foo.h`. If our header isn't on this "path" (and it probably isn't) we need to tell the compiler where to look. This is done with the `-I` flag. It takes a relative path with no spaces! So if you have all your header files in a directory called "include" in the directory you are running gcc from, then you would add: `-Iinclude` to the compiler command.

When you want to use a shared object library, you need to tell the linker where to find it! This works much like the include flag, but is instead `-L`. If you have a shared object in a sub-directory called `lib` you would add `-Llib` to the compiler command.

Finally, you need to instruct the linker to use the library to fill in any functions you use that aren't defined yet. This uses the lower-case `-l` flag with the name of the library. You may have seen this already where you have to add `-lm` to use the math libraries. So if you have

a library with the filename `libfoo.so` you would link to it with the flag `-lfoo` added to the compiler.

Putting this all together, if we have the library `foo` with the headerfile in the `include` directory and the library shared object in the `lib` directory, we'd need to type `gcc -Iinclude main.c -Llib -lfoo -o mycoolprogram` just to get the executable, and that is only one library without any debugging or warnings! That's waaaaay too much typing. Thankfully the next section introduces a tool that takes care of all of this for you.

## 8.2 make

In this section, we introduce the basics of *GNU make*, which is a powerful tool to manage the generation of executables or non-source files (such as libraries) from source code. As the tools introduces before, the official website is the authority on GNU make: <https://www.gnu.org/software/make/>. You can probably find whatever you want to know about GNU make on its official website. Driven by a small example below, we aim to introduce the basic usage and mechanism behind make.

### 8.2.1 Why use make?

Large projects can contain thousands of lines of code, distributed in multiple source files, written by many developers and arranged in several subdirectories. A project may contain several component divisions. These components may have complex inter-dependencies – for example, in order to compile component `X`, you have to first compile `Y`; in order to compile `Y`, you have to first compile `Z`; and so on. For a large project, when a few changes are made to the source, manually recompiling the entire project each time is tedious, error-prone and time-consuming.

Make is a solution to these problems. It can be used to specify dependencies between components, so that it will compile components in the order required to satisfy dependencies. An important feature is that when a project is recompiled after a few changes, it will recompile only the files that have changed, and any components that are dependent on it. This saves a lot of time. Make is, therefore, an essential tool for a large software project.

### 8.2.2 What makes a Makefile

Each project needs a Makefile – a script that describes the project structure, namely, the source code files, the dependencies between them, compiler arguments, and how to produce the target output (normally, one or more executables). Whenever the `make` command is executed, the Makefile in the current working directory is interpreted, and the instructions are executed to produce the target outputs. The Makefile contains a collection of rules, macros, variable assignments, etc. (“Makefile” or “makefile” are both acceptable.)

By default, make will seek the “makefile” or “Makefile” file under the current directory, unless a specific file is given by the parameter “-f”. If both “makefile” and “Makefile” are in the directory, “makefile” will be taken and “Makefile” is ignored. However in practice, “Makefile” is more preferable. To understand why, please refer to [https://www.gnu.org/software/make/manual/html\\_node/Makefile-Names.html](https://www.gnu.org/software/make/manual/html_node/Makefile-Names.html).

### 8.2.3 A sample project

To acquaint ourselves with the basics of make, let’s use a simple C “Hello world” project with a Makefile that handles building the target binary. We have three files (below): *module.h*, the header file that contains the declarations; *module.c*, which contains the definition of the function defined in *module.h*; and the main file, *main.c*, in which we call the *sample\_func()* defined in *module.c*. Since *module.h* includes the required header files like *stdio.h*, we don’t need to include *stdio.h* in every module; instead, we just include *module.h*. Here, *module.c* and *main.c* can be compiled as separate object modules, and can be linked by GCC to obtain the target binary.

```
/* module.h: */

#include <stdio.h>
void sample_func();

/* module.c: */

#include "module.h" /*include a local library*/
void sample_func(void) {
    printf("Hello world!");
} /*end sample_func*/

//main.c:

#include "module.h"
int main(void) {
    sample_func(); /*call our sample function*/
    return 0; /*terminate normally*/
} /*end main*/
```

The following are the manual steps to compile the project and produce the target binary:

```
~$ gcc -I . -c main.c           #comment: Obtain main.o
~$ gcc -I . -c module.c         #comment: Obtain module.o
~$ gcc main.o module.o -o target_bin  #comment: Obtain target binary
```

#Note: (-I is used to include the current directory (.) as a header file location.)



### 8.2.4 Writing a Makefile from scratch

By convention, all variable names used in a Makefile are in upper-case. A common variable assignment in a Makefile is `CC = gcc`, which can then be used later on as `$CC` or `$(CC)`. Makefiles use `#` as the comment-start marker, just like in shell scripts.

The general syntax of a Makefile rule is as follows:

```
target: dependency1 dependency2 ...
[TAB] action1
[TAB] action2
...
```

**Warning:**

Make sure the whitespace at the start of the action lines are TABS not spaces. This can prevent an otherwise correct Makefile from functioning.

Let's take a look at a simple Makefile for our sample project:

```
all: main.o module.o
    gcc main.o module.o -o target_bin
main.o: main.c module.h
    gcc -I . -c main.c
module.o: module.c module.h
    gcc -I . -c module.c
clean:
    rm -rf *.o
    rm target_bin
```

We have four targets in the Makefile:

- `all` is a special target that depends on `main.o` and `module.o`, and has the command (from the manual steps earlier) to make GCC link the two object files into the final executable binary.
- `main.o` is a filename target that depends on `main.c` and `module.h`, and has the command to compile `main.c` to produce `main.o`.
- `module.o` is a filename target that depends on `module.c` and `module.h`; it calls GCC to compile the `module.c` file to produce `module.o`.
- `clean` is a special target that has no dependencies, but specifies the commands to remove generated files from the project.

You may be wondering why the order of the make targets and commands in the Makefile are not the same as that of the manual compilation commands we ran earlier. The reason is so that the easiest invocation, by just calling the `make` command, will result in the most commonly desired output – the final executable. How does this work?

The `make` command accepts a target parameter (one of those defined in the Makefile), so the generic command line syntax is `make <target>`. However, `make` also works if you do

not specify any target on the command line, saving you a little typing; in such a case, it defaults to the first target defined in the Makefile. In our Makefile, that is the target `all`, which results in the creation of the desired executable binary target `_bin!`

### 8.2.5 Makefile processing

When the `make` command is executed, it looks for a file named `makefile` or `Makefile` in the current directory. It parses the found Makefile, and constructs a dependency tree. Based on the desired make target specified (or implied) on the command-line, `make` checks if the dependency files of that target exist. And (for filename targets, explained below) if they exist – whether they are newer than the target itself, by comparing file timestamps.

Before executing the action (commands) corresponding to the desired target, its dependencies must be met; when they are not met, the targets corresponding to the unmet dependencies are executed before the given make target, to supply the missing dependencies.

When a target is a filename, `make` compares the timestamps of the target file and its dependency files. If the dependency filename is another target in the Makefile, `make` then checks the timestamps of that target’s dependencies. It thus winds up recursively checking all the way down the dependency tree, to the source code files, to see if any of the files in the dependency tree are newer than their target filenames. (Of course, if the dependency files don’t exist, then `make` knows it must start executing the make targets from the “lowest” point in the dependency tree, to create them.)

If `make` finds that files in the dependency tree are newer than their target, then all the targets in the affected branch of the tree are executed, starting from the “lowest”, to update the dependency files. When `make` finally returns from its recursive checking of the tree, it completes the final comparison for the desired make target. If the dependency files are newer than the target (which is usually the case), it runs the command(s) for the desired make target.

This process is how `make` saves time, by executing only commands that need to be executed, based on which of the source files (listed as dependencies) have been updated, and have a newer timestamp than their target.

Now, when a target is not a filename (like `all` and `clean` in our Makefile, which we called “special targets”), `make` obviously cannot compare timestamps to check whether the target’s dependencies are newer. Therefore, such a target is always executed, if specified (or implied) on the command line.

For the execution of each target, `make` prints the actions while executing them. Note that each of the actions (shell commands written on a line) are executed in a separate sub-shell. If an action changes the shell environment, such a change is restricted to the sub-shell for that action line only. For example, if one action line contains a command like “`cd newdir`”,

the current directory will be changed only for that line/action; for the next line/action, the current directory will be unchanged.

### 8.2.6 Working with Makefiles

After understanding how make processes Makefiles, let's run make on our own Makefile, and see how it is processed to illustrate how it works. In the project directory, we run the following command:

```
~$ make
gcc -I . -c main.c
gcc -I . -c module.c
gcc main.o module.o -o target_bin
```

What has happened here?

When we ran make without specifying a target on the command line, it defaulted to the first target in our Makefile – that is, the target "all". This target's dependencies are module.o and main.o. Since these files do not exist on our first run of make for this project, make notes that it must execute the targets main.o and module.o. These targets, in turn, produce the main.o and module.o files by executing the corresponding actions/commands. Finally, make executes the command for the target "all". Thus, we obtain our desired output, target\_bin.

If we immediately run make again, without changing any of the source files, we will see that only the command for the target all is executed:

```
~$ make
gcc main.o module.o -o target_bin
```

Though make checked the dependency tree, neither of the dependency targets (module.o and main.o) had their own dependency files bearing a later timestamp than the dependency target filename. Therefore, make rightly did not execute the commands for the dependency targets. As we mentioned earlier, since the target all is not a filename, make cannot compare file timestamps, and thus executes the action/command for this target.

Now, we update module.c by adding a statement "printf("n first update");" inside the "sample\_func()" function. We then run make again:

```
~$ make
gcc -I . -c module.c
gcc main.o module.o -o target_bin
```

Since module.c in the dependency tree has changed (it now has a later timestamp than its target, module.o), make runs the action for the module.o target, which recompiles the changed source file. It then runs the action for the all target.

We can explicitly invoke the clean target to clean up all the generated .o files and target\_bin:

```
$ make clean
rm -rf *.o
```

```
rm target_bin
```

### 8.2.7 make is for automating all kinds of compilation

You can use make for L<sup>A</sup>T<sub>E</sub>X compilation too. For example, if you are using the university's L<sup>A</sup>T<sub>E</sub>X template for an honors thesis and you want to compile the document, compile the bibliography, and then switch out the blank signature page on the front with the copy of the one your committee signed, here is a makefile for that:

```
NAME = thesis_main
LATEX = pdflatex
BIBTEX = bibtex

bib:
$(LATEX) $(NAME).tex
$(LATEX) $(NAME).tex
$(LATEX) $(NAME).tex
$(BIBTEX) $(NAME)
$(LATEX) $(NAME).tex
$(LATEX) $(NAME).tex

chapters:
$(LATEX) $(NAME).tex
$(LATEX) $(NAME).tex

all: chapters
pdftk thesis_main.pdf cat 2-end output temp.pdf
pdftk SignaturePage.pdf temp.pdf cat output MySignedThesis.pdf

clean:
rm -f *.blg *.bbl *.aux *.log *.dvi *.out *.idx *.log *.lof *.lot
```

In this example, your thesis is in thesis\_main.tex and you choose to use `pdftk` to switch out the signature page. Notice how variables are used with a shell-like syntax and that we explicitly request the file be run several times to ensure all the links resolve.

### 8.2.8 Using make

You can now use makfiles to:

- Reduce compile times on multi-file projects with small changes
- Give your binary a descriptive name without typing `-o useful_name` every time
- Add many useful compiler flags without memorizing and retyping (try setting as many warnings as you can!)
- Use header files and linker commands to organize and reuse your code
- Never forget to add `-lm` again!

*Warning:*

«««< HEAD Now that we have a consistent method to control compilation, we will begin taking advantage of compiler options. From this point on the autograder will always compile your code with the option `-std=c99` to tell the compiler more explicitly which version of C we are using. You are responsible for ensuring your code compiles in this mode. All C we have presented in this course is C99 compliant.

===== Now that we have a consistent manor to control compilation, we will begin taking more advantage of compiler options. From this point on the autograder will always compile your code with the option `-std=c99` to tell the compiler more explicitly which version of C we are using. You are responsible for ensuring your code compiles in this mode, including with the flags `-Wall -Werror`. All C we have presented in this course is C99 compliant.

»»»> b688173d9fd1456c557949f0cfb944b59b67b916

You are encouraged to use this to enable more warnings as mentioned in the debugging lab to have the compiler assist in your programming.

For ideas on how to write "good" make files, the GNU style guide has great guidelines with reasons and examples: [https://www.gnu.org/prep/standards/html\\_node/Makefile-Conventions.html#Makefile-Conventions](https://www.gnu.org/prep/standards/html_node/Makefile-Conventions.html#Makefile-Conventions)

## 8.3 Advanced Data Structures: struct

### 8.3.1 What is struct

A structure is a user-defined data type in C. A structure creates a data type that can be used to group variables together. The key word in C for a structure is “struct”.

### 8.3.2 How to create a structure

The “struct” keyword is used to create a structure. Below is an example of how it is used:

```
struct address {
    char name[50];
    char street[100];
    char city[50];
    char state[20]
    int pin;
}; /*end struct address*/
```

Notice that a structure is simply a set of variable declarations grouped together. Each variable within a structure is called a member.

### 8.3.3 How to declare structure variables?

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

```
/* A variable declaration with structure declaration. */
struct Point {
```

```

    int x, y;
} p1; /* The variable p1 is declared with 'Point' */

/* A variable declaration like basic data types */
struct Point {
    int x, y;
}; /*end struct Point*/

int main(void) {
    struct Point p1; // The variable p1 is declared like a normal variable
    struct Point points[10]; // An array of 10 Point structs
    return 0; /*terminate normally*/
} /*end main*/

```

### 8.3.4 How to initialize structure members?

Structure members cannot be initialized with declaration. For example, the following C program fails during compilation.

```

struct Point {
    int x = 0; /* COMPILER ERROR: cannot initialize members here */
    int y = 0; /* COMPILER ERROR: cannot initialize members here */
}; /*end definition of struct Point*/

```

The reason for the above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Structure members can be initialized using curly braces “{ }”. The following is an example of a valid struct initialization.

```

struct Point {
    int x, y;
}; /*end definition of struct Point*/

int main(void) {
    /* A valid initialization. member x gets value 0 and y
       gets value 1. The order of declaration is followed. */
    struct Point p1 = {0, 1};
} /*end main*/

```

### 8.3.5 How to access structure elements?

Structure members are accessed using the dot (.) operator.

```

struct Point {
    int x, y;
}; /*end definition of struct Point*/

```

```

int main(void) {
    struct Point p1 = {0, 1};

    /* Accesing members of point p1 */
    p1.x = 20;
    printf ("x = %d, y = %d", p1.x, p1.y);

    return 0; /*terminate without error*/
} /*end main*/

```

output:  
x = 20, y = 1

### 8.3.6 What is a structure pointer?

Like primitive types (the basic built-in types we have been using), we can have pointers to a structure. If we have a pointer to a structure, members are accessed using the arrow ( -> ) operator.

```

struct Point {
    int x, y;
}; /*end definition of struct Point*/

int main(void) {
    struct Point p1 = {1, 2};

    /* p2 is a pointer to structure p1 */
    struct Point *p2 = &p1;

    /* Accessing structure members using structure pointer */
    printf("%d %d", p2->x, p2->y);
    return 0; /*terminate normally*/
} /*end main*/

```

output:  
1 2

The -> operator is "syntactic sugar" for dereferencing the pointer to a structure and accessing a member. Syntactic sugar is just a nice way to write something that can be expressed other ways. In this case the expression `p2->x` from the example above is equivalent to `(*p2).x` which very explicitly shows what is happening but can be annoying and error-prone to type out.

### 8.3.7 Allocating Structures

Dynamically allocating a struct is similar to allocating an array. Declare a pointer to the right kind of struct (if we were using the `Point` struct from the above example, the pointer declaration would be `struct Point *ptr;`) then allocate enough space for as many structs as

you need: `ptr = (struct Point*) malloc(num * sizeof(struct Point));` where `num` is an integer number of the number of Points you need to store.

Similar to an array, you can now iterate through the structs and access them by an offset. To print each point in our example you could make a for loop like `for (i=0; i<num; i++)` and access each point as `(ptr+i)->x`.

## 8.4 Project – Library Usage

As we move into numerical methods, you will want to reuse code you have already created, and make use of code written by the community of open-source developers. By leveraging work that has already been done and proven, you can reduce the opportunity for errors while eliminating the need to re-invent the wheel.

*Note:*

For this course, you will be told when you can use outside libraries. Code you have already written can always be included but must follow the external library guidelines of the assignment being completed.

### 8.4.1 Building a library

Convert the 4 integration functions you wrote in the last lab into your own integration library.

In the base of your lab repository, make a directory named `integration` and copy your `integration.c` file from lab 7 into this new folder. Complete the following 3 exercises in that location.

#### Exercise 1.

Integration Header — (5 points):

Make a header file for your integration library. It must be named `integration.h` and contain:

- 1 point** All `#include` statements needed by your library
- 3 points** Forward declarations for all your library functions
- 1 point** Include guards (see this helpful [Wikipedia article](#))

#### Exercise 2.

Integration Makefile — (5 points):

Write a Makefile for your library. It must be named `Makefile` and should:

- 2 points** Compile your library with all required flags, including `-std=c99 -Wall -Werror`
- 2 point** Produce a shared object (`.so`) file name `libintegration.so`
- 1 point** Contain a clean target such that running `make clean` inside the integration directory removes files generated by the build



### Exercise 3.

Integration Struct — (5 points):  
Standardize

**2 points** Define a struct called `integral` that contains (in this order):

1. A function pointer called `integrand` that takes a double and returns a double
2. A double called `low` to store one bound of integration
3. A double called `high` to store the other bound of integration

**2 point** Modify your integration functions and their headerfile prototypes to take a single `integral` struct as an argument  
(Gauss Quad will take the struct plus an integer for the order)

**1 point** Put the struct definition in your headerfile so any code that includes the header has access to your struct definition

At this point you should have a directory in the base of your lab repository named `integration` which contains:

**integration.h** A header-file defining the interface to your library (i.e. the function prototypes)

**Makefile** A makefile that builds the `libintegration.so` shared object from your source code and can clean up after the build

**integration.c** The source-file with the function definitions

The autograder will run from the `integration` sub-directory, first running `make` then compiling the test program with the following flags: `-I. -L. -lcsv` to tell the compiler and linker that it should expect to find `integration.h` and `libintegration.so` in the current directory.

#### *A note on grading:*

You will not be put in double jeopardy on your integration code – we will not be testing the mathematical validity of the answers, but rather the correctness of your conversion to a library function. That being said, incorrect code requires manual grading (and therefore more scrutiny) so if you do not believe that your lab 7 implementations are correct, we invite you to work with a TA or classmate this week to improve those functions.

### 8.4.2 Using a library

To practice using other libraries, and to prepare for writing programs that ingest large amounts of data, you will write a program that uses the libcsv library to read a CSV file and construct a 2D array to hold it. If you need a refresher on CSV files, [Wikipedia](#) has a good reference.

In your lab repository, the `CSV_parse` directory contains a very standard layout for a C library. The following directories are included:

**bin** Short for binary, all runnable executables produced by your Makefile should be placed here

**doc** Short for Documentation, includes the documentation for the libcsv functions.

**include** Traditional directory for all header files, contains the `csv.h` header for using libcsv

**lib** Short for Library, contains the `libcsv.so` shared object

**src** Short for Source, your code should go here

**examples** A collection of small examples written by the libcsv authors

#### Exercise 4. CSV Makefile — (10 points):

Write a Makefile named `Makefile` in the `csv_parse` directory. When run without a target (i.e. `make`) it should build both programs in the `examples` sub directory into binaries with the same name as the source file without an extension in the `bin` directory. For example, the file `usage_example.c` should make a program named `usage_example` in `bin`. Your Makefile should also contain a `clean` target that removes the generated programs and any build artifacts like `.o` files. See section 8.1.4 for assistance.

**2 points each** Compile each tool program

**1 point** Running `make clean` removes all generated files

**5 points** From the build and clean requirements for exercise 5, detailed below

#### Exercise 5. CSV Parsing — (15 points):

Write a program that takes the name of a CSV file on the command line, reads and validates the file, and construct a 2D matrix of doubles with the values if valid. If a valid matrix is constructed, it prints the matrix with the same format from lab 5. Below is pseudo-code to follow for this exercise.

---

**Algorithm 2** Algorithm for CSV Matrix parse Code

---

```
Take filename from argv
Check that file can be opened
if File cannot be opened then
    Exit Code 1
end if
Create CsvParser
if CSV Parser Error then
    Exit Code 2
end if
Check size of first row
while Not end of file do
    Count each row
    Check each row length
    if If row length not match first row length then
        Exit code 3
    end if
end while
Allocate matrix of doubles (use lab 5 code)
Reset CSV Parser
for for each field do
    check if valid double (optional plus or minus, 1 or more digits, optional one decimal, optional digits)
    if invalid double then
        Exit Code 4
    end if
    store max length of value string
    Fill in matrix
end for
Use lab 5 matrix code with stored max string length for output
Exit Code 0
```

---

There are points available for each potential program end-state. Make sure that your program exits with the correct code for each situation presented.

**1 point** File Error (code 1)

**2 points** CSV Parser Error (code 2)

**1 point** Dimension Error (code 3)

**2 points** Value Error (code 4)

**10 points** Valid Execution (code 0)

### 8.4.3 Report

Your report this week is worth 10 points. It should contain the following two sections, each worth 5 points:

- Complexity Analysis: What is the complexity of your CSV parsing code?
- Loop Analysis: What are the major loops contributing to that complexity? What are the invariants of these loops? Which loops (if any) do not contribute to complexity?

### 8.4.4 Lab Submission

*Warning:* As stated in the syllabus, programs and reports that do not compile will be awarded zero points!

If you having issues with your lab work, utilize your resources (lecture notes, text books, lab manuals, personal internet research) and contact a TA for assistance if you are still unable to resolve your issue.

When you are finished with your lab, follow this procedure to turn in your work for grading:

1. Ensure all exercises are in the correctly named files in your lab directory
2. Ensure the source file compiles without errors
3. Ensure that your report is named correctly, is in the correct directory, and builds without errors
4. Use `git status` to verify all the work you want to be graded has been committed to the develop branch
5. Use `git push` to update GitHub with all your changes
6. Open a pull request on GitHub
7. **DO NOT** click the "close and merge" button - your PR should remain open for the autograder to find

Congratulations, you are 57.14285714% of the way though 361!

## 9 | Machine Numbers and the IEEE 754 Floating-Point Standard

*Reference:*

**Introduction to Scientific and Technical Computing:**

Chapter 2: Machine Numbers and the IEEE 754 Floating-Point Standard

### Objectives

- Use loop invariants to help catch floating-point errors.
- Understand underflow and overflow, why they happen, and how to prevent them in industrial code.
- Understand the limits of machine representation of numbers, including machine epsilon, and ensure robust computation within the constraints of machine representation.
- Be able to calculate approximation errors of machine computations and account for them.
- Learn how to test for cases when operations like addition might be influenced by significant differences in the sizes or types of the operands.

### 9.1 Number Representation

In this lab, we deal with the machine representation of decimal numbers, i.e., numbers with base 10. For example, 42, 100, 200, 456, etc., are all base-10 numbers. Similarly, 101.43, 56.7353, 0.00000234 are all fractional base-10 numbers. Humans use the base-10 number representation system. However, computer don't understand decimal; they work with the binary number system which is base-2. Let's see how decimal numbers are converted to binary, and vice-versa. We represent a base-10 number  $x$  using the notation  $x_{10}$ , while a base-2 number  $y$  is represented using the notation  $y_2$ .

#### 9.1.1 Decimal-to-Binary Conversion

**Integers.**

To convert a decimal integer to binary, start with the integer and divide it by 2 keeping track of the quotient and the remainder. Continue dividing the quotient by 2 until you get

a quotient of zero. To get the binary representation of the integer, write out the remainder in the reverse order. As an example, let's see how to convert  $194_{10}$  to binary.

Operation	Quotient	Remainder
$194 \div 2$	97	0
$97 \div 2$	48	1
$48 \div 2$	24	0
$24 \div 2$	12	0
$12 \div 2$	6	0
$6 \div 2$	3	0
$3 \div 2$	1	1
$1 \div 2$	0	1

Therefore,  $194_{10}$  is equivalent to  $11000010_2$  (by writing out the remainders in reverse order). Similarly,  $12_{10}$  is equivalent to  $1100_2$ .

### Fractions.

To convert fractions (real numbers between 0 and 1), start with the fraction in question and multiply it by 2 keeping track of the resulting integer and fractional part. Continue multiplying by 2 until you get a resulting fractional part of equal to zero. To get the binary representation of the fraction, write out the integer parts from the results of each multiplication. As an example, let's see how to convert  $0.375_{10}$  to binary.

Operation	Integer	Fraction
$0.375 \times 2$	0	0.75
$0.75 \times 2$	1	0.5
$0.5 \times 2$	1	0.0

Therefore,  $0.375_{10}$  is equivalent to  $0.011_2$  (by writing out the integer results in each step). Similarly,  $0.625_{10}$  is equivalent to  $0.101_2$ . How about converting  $0.545_{10}$  to binary?

Operation	Integer	Fraction
$0.545 \times 2$	1	0.09
$0.09 \times 2$	0	0.18
$0.18 \times 2$	0	0.36
$0.36 \times 2$	0	0.72
$0.72 \times 2$	1	0.44
$0.44 \times 2$	0	0.88
$0.88 \times 2$	1	0.76
...	...	...

As it turns out, there is no exact binary representation of  $0.545_{10}$ . However, a close approximation of  $0.545_{10}$  is  $0.1000101_2$ . More about this in later sections. Therefore, **not all fractional decimal numbers can be represented exactly in the binary system!**

### 9.1.2 Binary-to-Decimal Conversion

#### Integers.

To convert a binary integer to a decimal, we start from the left. Starting with a total of zero, we take the current total, multiply it by two and add the current binary digit. Continue until there are no more digits left. As an example, let's see how to convert  $1011_2$  to decimal.

$$\begin{aligned}2 \times 0 + 1 &= 1 \\2 \times 1 + 0 &= 2 \\2 \times 2 + 1 &= 5 \\2 \times 5 + 1 &= 11\end{aligned}$$

Therefore,  $1011_2$  is equivalent to  $11_{10}$ . Similarly,  $101001_2$  is equivalent to  $41_{10}$ .

#### Fractions.

To convert a binary fraction to a decimal, we start from the right. Starting with a total of zero, we take the current total, add the current digit, and divide the result by 2. Continue until there are no digits left. As an example, let's see how to convert  $0.1101_2$  to decimal.

$$\begin{aligned}(1 + 0) \div 2 &= 0.5 \\(0 + 0.5) \div 2 &= 0.25 \\(1 + 0.25) \div 2 &= 0.625 \\(1 + 0.625) \div 2 &= 0.8125\end{aligned}$$

Therefore,  $0.1101_2$  is equivalent to  $0.8125_{10}$ . Let's try converting  $0.1000101_2$  to decimal (from the last section, we know that  $0.1000101_2$  is a binary approximation of  $0.545_{10}$ ).

$$\begin{aligned}(1 + 0) \div 2 &= 0.5 \\(0 + 0.5) \div 2 &= 0.25 \\(1 + 0.25) \div 2 &= 0.625 \\(0 + 0.625) \div 2 &= 0.3125 \\(0 + 0.3125) \div 2 &= 0.15625 \\(0 + 0.15625) \div 2 &= 0.078125 \\(1 + 0.078125) \div 2 &= 0.5390625\end{aligned}$$

As expected,  $0.1000101_2$  is an approximation of  $0.545_{10}$  and is equivalent to  $0.5390625_{10}$ . The approximation can be made more exact by using more binary digits.

## 9.2 Integer Representation

Integers are whole numbers or fixed-point numbers with the radix point fixed after the least-significant bit. They contrast to real numbers or floating-point numbers, where the position of the radix point varies. It is important to take note that integers and floating-point numbers are treated differently in computers. They have different representations and are processed differently (e.g., floating-point numbers are processed in a so-called floating-point processor).

Computers use a fixed number of bits to represent an integer. The commonly-used bit-lengths for integers are 8-bit, 16-bit, 32-bit or 64-bit. Besides bit-lengths, there are two representation schemes for integers:

- Unsigned Integers: can represent zero and positive integers.
- Signed Integers: can represent zero, positive and negative integers. Three representation schemes had been proposed for signed integers:
  1. Sign-Magnitude representation
  2. 1's Complement representation
  3. 2's Complement representation

You, as the programmer, need to decide on the bit-length and representation scheme for your integers, depending on your application's requirements. Suppose that you need a counter for counting a small quantity from 0 up to 200, you might choose the 8-bit unsigned integer scheme as there are no negative numbers involved.

### 9.2.1 n-bit Unsigned Integers

Unsigned integers can represent zero and positive integers, but not negative integers. The value of an unsigned integer is interpreted as "the magnitude of its underlying binary pattern". For example,

- Suppose that  $n=8$  and the binary pattern is  $0100\ 0001_2$ . The value of this unsigned integer is  $65_{10}$ .
- Suppose that  $n=16$  and the binary pattern is  $0001\ 0000\ 0000\ 1000_2$ . The value of this unsigned integer is  $4104_{10}$ .
- Suppose that  $n=16$  and the binary pattern is  $0000\ 0000\ 0000\ 0000_2$ . The value of this unsigned integer is  $0_{10}$ .

An  $n$ -bit pattern can represent  $2^n$  distinct integers. An  $n$ -bit unsigned integer can represent integers from 0 to  $(2^n) - 1$ , as tabulated below:

$n$	Minimum	Maximum
8	0	$(2^8) - 1 = 255$
16	0	$(2^{16}) - 1 = 65,535$
32	0	$(2^{32}) - 1 = 4,294,967,295$
64	0	$(2^{64}) - 1 = 18,446,744,073,709,551,615$



### 9.2.2 Signed Integers

Signed integers can represent zero, positive integers, and negative integers. Three representation schemes are available for signed integers:

1. Sign-Magnitude representation
2. 1's Complement representation
3. 2's Complement representation

In all three schemes, the most-significant bit (msb) is called the sign bit. The sign bit is used to represent the sign of the integer – with 0 for positive integers and 1 for negative integers. The magnitude of the integer, however, is interpreted differently in different schemes.

### 9.2.3 n-bit Sign Integers in Sign-Magnitude Representation

In sign-magnitude representation:

- The most-significant bit (msb) is the sign bit, with value of 0 representing positive integer and 1 representing negative integer.
- The remaining n-1 bits represents the magnitude (absolute value) of the integer. The absolute value of the integer is interpreted as "the magnitude of the (n-1)-bit binary pattern".

For example,

- Suppose that n=8 and the binary representation is 0 100 0001<sub>2</sub>.

Sign bit is 0  $\Rightarrow$  positive

Absolute value is 100 0001<sub>2</sub> = 65<sub>10</sub>

Hence, the integer is +65<sub>10</sub>

- Suppose that n=8 and the binary representation is 1 000 0001<sub>2</sub>.

Sign bit is 1  $\Rightarrow$  negative

Absolute value is 000 0001B = 1D

Hence, the integer is -1D

- Suppose that n=8 and the binary representation is 0 000 0000B.

Sign bit is 0  $\Rightarrow$  positive

Absolute value is 000 0000B = 0D

Hence, the integer is +0D

- Suppose that n=8 and the binary representation is 1 000 0000B.

Sign bit is 1  $\Rightarrow$  negative

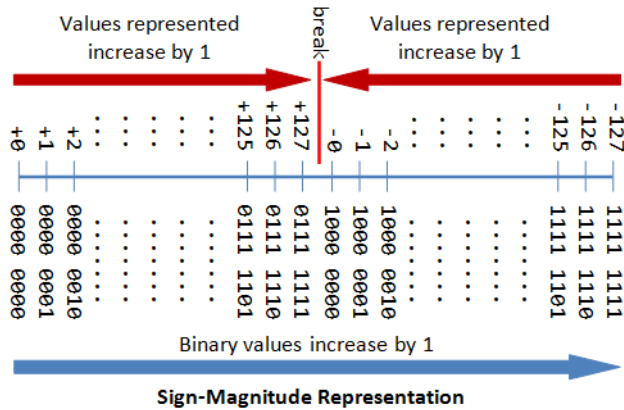
Absolute value is 000 0000B = 0D

Hence, the integer is -0D

The Sign-Magnitude Representation schema is shown in the figure below.

The drawbacks of sign-magnitude representation are:

1. There are two representations (0000 0000B and 1000 0000B) for the number zero, which could lead to inefficiency and confusion.



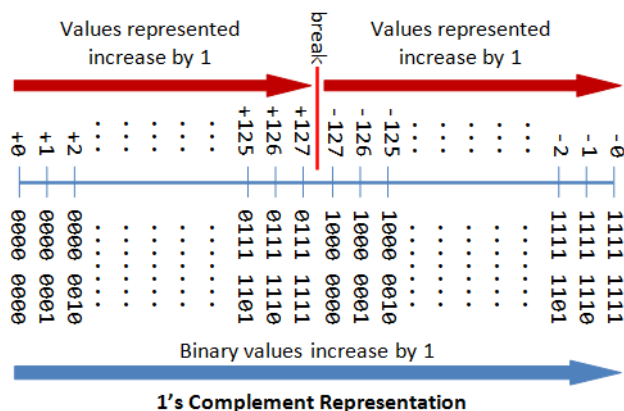
2. Positive and negative integers need to be processed separately.

### 9.2.4 n-bit Sign Integers in 1's Complement Representation

In 1's complement representation:

- Again, the most significant bit (msb) is the sign bit, with a value of 0 representing positive integers and 1 representing negative integers.
- The remaining n-1 bits represent the magnitude of the integer, as follows:
  - for positive integers, the absolute value of the integer is equal to "the magnitude of the (n-1)-bit binary pattern".
  - for negative integers, the absolute value of the integer is equal to "the magnitude of the complement (inverse) of the (n-1)-bit binary pattern" (hence why it's called 1's complement).

The 1's Complement Representation schema is shown in the figure below.



For example,

- Suppose that n=8 and the binary representation 0 100 0001B.

Sign bit is 0  $\Rightarrow$  positive  
Absolute value is 100 0001B = 65D  
Hence, the integer is +65D

- Suppose that n=8 and the binary representation 1 000 0001B.

Sign bit is 1  $\Rightarrow$  negative  
Absolute value is the complement of 000 0001B, i.e., 111 1110B = 126D  
Hence, the integer is -126D

- Suppose that n=8 and the binary representation 0 000 0000B.

Sign bit is 0  $\Rightarrow$  positive  
Absolute value is 000 0000B = 0D  
Hence, the integer is +0D

- Suppose that n=8 and the binary representation 1 111 1111B.

Sign bit is 1  $\Rightarrow$  negative  
Absolute value is the complement of 111 1111B, i.e., 000 0000B = 0D  
Hence, the integer is -0D

Again, the drawbacks are:

- There are two representations (0000 0000B and 1111 1111B) for zero.
- The positive and negative integers need to be processed separately.

### 9.2.5 n-bit Sign Integers in 2's Complement Representation

In 2's complement representation:

- Again, the most significant bit (msb) is the sign bit, with a value of 0 representing positive integers and 1 representing negative integers.
- The remaining n-1 bits represent the magnitude of the integer, as follows:
  - for positive integers, the absolute value of the integer is equal to "the magnitude of the (n-1)-bit binary pattern".
  - for negative integers, the absolute value of the integer is equal to "the magnitude of the complement of the (n-1)-bit binary pattern plus one" (hence why it's called 2's complement).

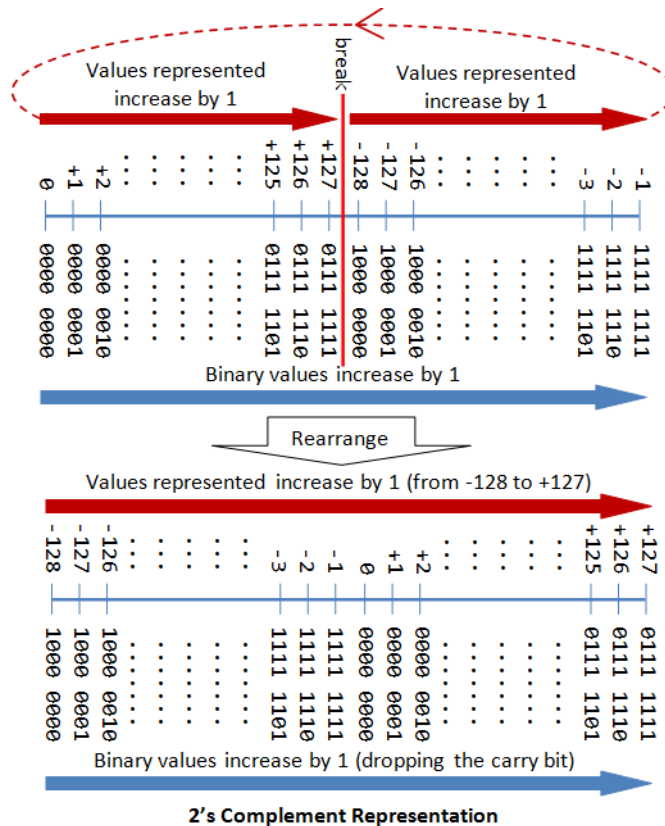
The 2's Complement Representation schema is shown in the figure below.

For example,

- Suppose that n=8 and the binary representation 0 100 0001B.

Sign bit is 0  $\Rightarrow$  positive  
Absolute value is 100 0001B = 65D  
Hence, the integer is +65D

- Suppose that n=8 and the binary representation 1 000 0001B.



Sign bit is 1  $\Rightarrow$  negative

Absolute value is the complement of 000 0001B plus 1, i.e., 111 1110B + 1B = 127D

Hence, the integer is -127D

- Suppose that  $n=8$  and the binary representation 0 000 0000B.

Sign bit is 0  $\Rightarrow$  positive

Absolute value is 000 0000B = 0D

Hence, the integer is +0D

- Suppose that  $n=8$  and the binary representation 1 111 1111B.

Sign bit is 1  $\Rightarrow$  negative

Absolute value is the complement of 111 1111B plus 1, i.e., 000 0000B + 1B = 1D

Hence, the integer is -1D

We have discussed three representations for signed integers: signed-magnitude, 1's complement and 2's complement. Computers use 2's complement in representing signed integers. This is because:

1. There is only one representation for the number zero in 2's complement, instead of two representations as is the case with sign-magnitude and 1's complement.
2. Positive and negative integers can be treated together in addition and subtraction. Subtraction can be carried out using "addition logic".

Examples of addition and subtraction under the 2's Complement Representation:

- Addition of Two Positive Integers: Suppose that  $n=8$ ,  $65D + 5D = 70D$

$$\begin{array}{rcl} 65D & \rightarrow & 0100\ 0001B \\ 5D & \rightarrow & 0000\ 0101B(+ \\ & & 0100\ 0110B \rightarrow 70D\ (OK) \end{array}$$

- Subtraction is treated as the Addition of a Positive and a Negative Integer: Suppose that  $n=8$ ,  $65D - 5D = 65D + (-5D) = 60D$

$$\begin{array}{rcl} 65D & \rightarrow & 0100\ 0001B \\ -5D & \rightarrow & 1111\ 1011B(+ \\ & & 0011\ 1100B \rightarrow 60D\ (\text{discard carry} - OK) \end{array}$$

- Addition of Two Negative Integers: Suppose that  $n=8$ ,  $-65D - 5D = (-65D) + (-5D) = -70D$

$$\begin{array}{rcl} -65D & \rightarrow & 1011\ 1111B \\ -5D & \rightarrow & 1111\ 1011B(+ \\ & & 1011\ 1010B \rightarrow -70D\ (\text{discard carry} - OK) \end{array}$$

Because of the fixed precision (i.e., fixed number of bits), an  $n$ -bit 2's complement signed integer has a certain range. For example, for  $n=8$ , the range of 2's complement signed integers is  $-128$  to  $+127$ . During addition (and subtraction), it is important to check whether the result exceeds this range, in other words, whether overflow or underflow has occurred.

- Overflow: Suppose that  $n=8$ ,  $127D + 2D = 129D$  (overflow - beyond the range)

$$\begin{array}{rcl} 127D & \rightarrow & 0111\ 1111B \\ 2D & \rightarrow & 0000\ 0010B(+ \\ & & 1000\ 0001B \rightarrow -127D\ (\text{wrong}) \end{array}$$

- Underflow: Suppose that  $n=8$ ,  $-125D - 5D = -130D$  (underflow - below the range)

$$\begin{array}{rcl} -125D & \rightarrow & 1000\ 0011B \\ -5D & \rightarrow & 1111\ 1011B(+ \\ & & 0111\ 1110B \rightarrow +126D\ (\text{wrong}) \end{array}$$

An  $n$ -bit 2's complement signed integer can represent integers from  $-2^{(n-1)}$  to  $+2^{(n-1)} - 1$ , as tabulated. Take note that the scheme can represent all the integers within the range, without any gap. In other words, there are no missing integers within the supported range.

$n$	minimum	maximum
8	$-(2^7)$ ( $=-128$ )	$+(2^7) - 1$ ( $=+127$ )
16	$-(2^{15})$ ( $=-32,768$ )	$+(2^{15}) - 1$ ( $=+32,767$ )
32	$-(2^{31})$ ( $=-2,147,483,648$ )	$+(2^{31}) - 1$ ( $=+2,147,483,647$ )(9+ digits)
64	$-(2^{63})$ ( $=-9,223,372,036,854,775,808$ )	$+(2^{63}) - 1$ ( $=+9,223,372,036,854,775,807$ )(18+ digits)

### 9.3 Floating-Point Numbers

There are several ways to represent real numbers on computers.

Fixed point places a radix point somewhere in the middle of the digits, and is equivalent to using integers that represent portions of some unit. (The radix is the separator between the whole and fractional components of the number.) For example, one might represent 1/100ths of a unit; if you have four decimal digits, you could represent 10.82, or 00.01. Another approach is to use rationals, and represent every number as the ratio of two integers.

Floating-point representation - the most common solution - uses *scientific notation* to encode numbers, with a base number and an exponent. For example, 123.456 could be represented as  $1.23456 \times 10^2$ . In binary, the number 10100.110 could be represented as  $1.0100110 \times 2^4$ .

Floating-point solves a number of representation problems. Fixed-point has a fixed window of representation, which limits it from representing both very large and very small numbers. Also, fixed-point is prone to a loss of precision when two large numbers get divided. Floating-point, on the other hand, employs a sort of "sliding window" of precision appropriate to the scale of the number. This allows it to represent numbers from 1,000,000,000,000 to 0.0000000000000001 with ease, while maximizing precision (the number of digits) at both ends of the scale.

Formally, a floating-point number system is characterized by four integers:

$\beta$	Base or radix
$t$	Precision
$L, U$	Exponent range (L = lower; U = upper)

Any number  $x$  in the floating-point system is represented as follows:

$$x = \pm(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \dots + \frac{d_{t-1}}{\beta^{t-1}})\beta^e,$$

where

$$0 \leq d_i \leq \beta - 1, \quad i = 0, \dots, t-1, \quad L \leq e \leq U$$

Here,  $\beta$  is the number base (so  $\beta = 2$  for binary,  $\beta = 8$  for octal, and  $\beta = 16$  for hexadecimal), the string of base- $\beta$  digits  $(d_0, \frac{d_1}{\beta}, \dots, \frac{d_{t-1}}{\beta^{t-1}})$  is the *mantissa* or *significand*, and  $e$  is the *exponent* or *characteristic*.

In order to maximize precision, we *normalize* floating-point numbers to eliminate leading zeros. A floating-point system is normalized if the leading digit  $d_0$  is always nonzero unless the number represented is zero. Note that the representation of each number is unique.

For binary systems ( $\beta = 2$ ), the leading bit is then always 1 so we don't need to store it - therefore we get an extra bit of precision for every field! The number of normalized floating-point numbers is

$$2(\beta - 1)\beta^{t-1}(U - L + 1) + 1$$

because there are two choices of sign,  $\beta - 1$  choices for the leading digit of the mantissa,  $\beta$  choices for each of the remaining  $t - 1$  digits of the mantissa, and  $U - L + 1$  possible values for the exponent. We add 1 because the number could be zero, which is always uniquely represented by all zeros in both the mantissa and exponent.

We can calculate the smallest positive normalized floating-point number,

$$Underflowlevel = UFL = \beta^L,$$

which has a mantissa of all 0s except for the leading digit 1 and the smallest possible exponent.

We can also calculate the largest positive normalized floating-point number,

$$Overflowlevel = OFL = \beta^{U+1}(1 - \beta^{-t})$$

which has a mantissa of all  $\beta - 1$  digits and the largest possible exponent.

**Not all real numbers are exactly representable in a floating-point system!** For every real number  $x$  that is not exactly representable, we approximate  $x$  by rounding to the representation  $fl(x)$  in some way, such as rounding toward zero, rounding to the nearest representable number, or in the case of a tie, rounding to the nearest even representable number. **Note that decimal-to-binary and binary-to-decimal conversions introduce data errors through rounding.**

The IEEE floating-point standard provides some additional special values as well, including:

- **subnormal numbers**, which can have leading zeros in their representations in order to fill in the gaps on either side of 0; otherwise we could not represent any numbers between 0 and  $\beta^L$ .
- **Inf** or infinity, which results from dividing a finite number by zero.
- **NaN** or not a number, which results from undefined operations including  $0/0$ ,  $0 * Inf$ ,  $Inf/Inf$ .

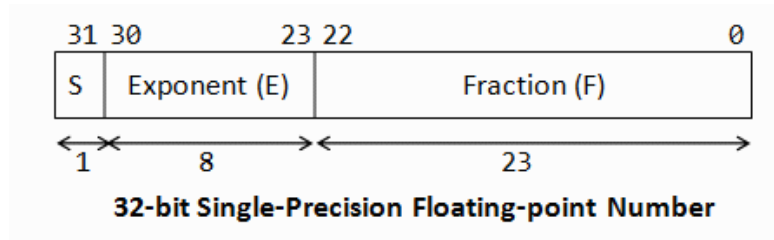
*Required Reading:*

*IEEE Standard 754 Floating Point Numbers* <http://steve.hollasch.net/cgindex/coding/ieeefloat.html>.

### 9.3.1 IEEE-754 32-bit Single-Precision Floating-Point Numbers

In 32-bit single-precision floating-point representation:

- The most significant bit is the sign bit (S), with 0 for positive numbers and 1 for negative numbers.
- The following 8 bits represent exponent (E).
- The remaining 23 bits represents fraction (F).



### Normalized Form

Let's look at this further with an example. Suppose that the 32-bit pattern is 1 1000 0001 011 0000 0000 0000 0000 0000, with:

- $S = 1$
- $E = 1000\ 0001$
- $F = 011\ 0000\ 0000\ 0000\ 0000\ 0000$

In the normalized form, the actual fraction is normalized with an implicit leading 1 in the form of 1.F. In this example, the actual fraction is  $1.011\ 0000\ 0000\ 0000\ 0000\ 0000 = 1 + 1 \times 2^{-2} + 1 \times 2^{-3} = 1.375D$ .

The sign bit represents the sign of the number, with  $S=0$  for positive and  $S=1$  for a negative number. In this example with  $S=1$ , this is a negative number, i.e.,  $-1.375D$ .

In normalized form, the actual exponent is  $E-127$  (so-called excess-127 or bias-127). This is because we need to be able to represent both positive and negative exponents. With an 8-bit  $E$ , ranging from 0 to 255, the excess-127 scheme could provide actual exponent of -127 to 128. In this example,  $E-127=129-127=2D$ .

Hence, the number represented is  $-1.3752^2=-5.5D$ .

The minimum and maximum normalized floating-point numbers are:

Precision	Normalized N(min)	Normalized N(max)
Single	0080 0000H	7F7F FFFFH
	0 00000001 000000000000000000000000B	0 11111110 000000000000000000000000B
	$E = 1, F = 0$	$E = 254, F = 0$
	$N(\min) = 1.0B \times 2^{-126}$ ( $\approx 1.17549435 \times 10^{-38}$ )	$N(\max) = 1.1...1B \times 2^{127} = (2 - 2^{-23}) \times 2^{127}$ ( $\approx 3.4028235 \times 10^{38}$ )
Double	0010 0000 0000 0000H	7FEF FFFF FFFF FFFFH
	$N(\min) = 1.0B \times 2^{-1022}$ ( $\approx 2.2250738585072014 \times 10^{-308}$ )	$N(\max) = 1.1...1B \times 2^{1023} = (2 - 2^{-52}) \times 2^{1023}$ ( $\approx 1.7976931348623157 \times 10^{308}$ )

### De-Normalized Form

Normalized form has a serious problem, with an implicit leading 1 for the fraction, it cannot represent the number zero! Convince yourself on this!

De-normalized form was devised to represent zero and other numbers.



For  $E=0$ , the numbers are in the de-normalized form. An implicit leading 0 (instead of 1) is used for the fraction; and the actual exponent is always -126. Hence, the number zero can be represented with  $E=0$  and  $F=0$  (because  $0.02^{-126} = 0$ ).

We can also represent very small positive and negative numbers in de-normalized form with  $E=0$ . For example, if  $S=1$ ,  $E=0$ , and  $F=011\ 0000\ 0000\ 0000\ 0000\ 0000$ . The actual fraction is  $0.011 = 1 \times 2^{-2} + 1 \times 2^{-3} = 0.375D$ . Since  $S=1$ , it is a negative number. With  $E=0$ , the actual exponent is -126. Hence the number is  $-0.375 \times 2^{-126} = -4.4 \times 10^{-39}$ , which is an extremely small negative number (close to zero).

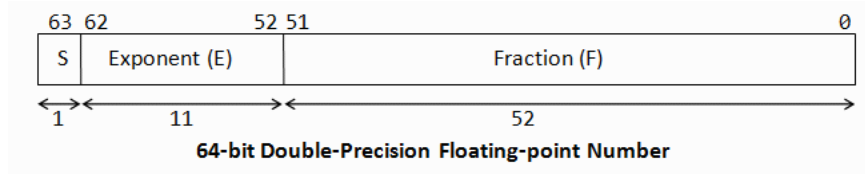
The minimum and maximum of denormalized floating-point numbers are:

Precision	Denormalized D(min)	Denormalized D(max)
Single	0000 0001H 0 00000000 000000000000000000000001B E = 0, F = 00000000000000000000000001B D(min) = $0.0...1 \times 2^{-126} = 1 \times 2^{-23} \times 2^{-126} = 2^{-149}$ ( $\approx 1.4 \times 10^{-45}$ )	007F FFFFH 0 00000000 1111111111111111111111111111B E = 0, F = 1111111111111111111111111111B D(max) = $0.1...1 \times 2^{-126} = (1 - 2^{-23}) \times 2^{-126}$ ( $\approx 1.1754942 \times 10^{-38}$ )
	0000 0000 0000 0001H D(min) = $0.0...1 \times 2^{-1022} = 1 \times 2^{-52} \times 2^{-1022} = 2^{-1074}$ ( $\approx 4.9 \times 10^{-324}$ )	001F FFFF FFFF FFFFH D(max) = $0.1...1 \times 2^{-1022} = (1 - 2^{-52}) \times 2^{-1022}$ ( $\approx 4.4501477170144023 \times 10^{-308}$ )

### 9.3.2 IEEE-754 64-bit Double-Precision Floating-Point Numbers

The representation scheme for 64-bit double-precision is similar to 32-bit single-precision:

- The most significant bit is the sign bit (S), with 0 for positive numbers and 1 for negative numbers.
- The following 11 bits represent exponent (E).
- The remaining 52 bits represents fraction (F).



The value (N) is calculated as follows:

- Normalized form: For  $1 \leq E \leq 2046$ ,  $N = (-1)^S \times 1.F \times 2^{(E-1023)}$ .
- Denormalized form: For  $E = 0$ ,  $N = (-1)^S \times 0.F \times 2^{(-1022)}$ .
- For  $E = 2047$ , N represents special values, such as  $\pm\text{INF}$  (infinity), NaN (not a number).

### 9.3.3 Summary

In summary, the value (N) is calculated as follows:

- For  $1 \leq E \leq 254$ ,  $N = (-1)^S \times 1.F \times 2^{(E-127)}$ . These numbers are in the so-called normalized form. The sign-bit represents the sign of the number. Fractional part (1.F) are normalized with an implicit leading 1. The exponent is biased (or in excess) of 127, so as to represent both positive and negative exponents. The range of exponents is -126 to +127.
- For  $E = 0$ ,  $N = (-1)^S \times 0.F \times 2^{(-126)}$ . These numbers are in the so-called denormalized form. The exponent of  $2^{(-126)}$  evaluates to a very small number. Denormalized form is needed to represent zero (with F=0 and E=0). It can also represent very small positive and negative numbers close to zero.
- For  $E = 255$ , it represents special values, such as  $\pm\text{INF}$  (positive and negative infinity) and NaN (not a number).

For example,

1. Suppose that an IEEE-754 32-bit floating-point representation pattern is 0 10000000 110 0000 0000 0000 0000.

Sign bit  $S = 0 \Rightarrow$  positive number

$E = 1000\ 0000\text{B} = 128\text{D}$  (in normalized form)

Fraction is 1.11B (with an implicit leading 1)  $= 1 + 1 \times 2^{-1} + 1 \times 2^{-2} = 1.75\text{D}$

The number is  $+1.75 \times 2^{(128-127)} = +3.5\text{D}$

2. Suppose that an IEEE-754 32-bit floating-point representation pattern is 1 01111110 100 0000 0000 0000 0000.

Sign bit  $S = 1 \Rightarrow$  negative number

$E = 0111\ 1110\text{B} = 126\text{D}$  (in normalized form)

Fraction is 1.1B (with an implicit leading 1)  $= 1 + 2^{-1} = 1.5\text{D}$

The number is  $-1.5 \times 2^{(126-127)} = -0.75\text{D}$

3. Suppose that an IEEE-754 32-bit floating-point representation pattern is 1 01111110 000 0000 0000 0000 0001.

Sign bit  $S = 1 \Rightarrow$  negative number

$E = 0111\ 1110\text{B} = 126\text{D}$  (in normalized form)

Fraction is 1.000 0000 0000 0000 0000 0001B (with an implicit leading 1)  $= 1 + 2^{-23}$

The number is  $-(1 + 2^{-23}) \times 2^{(126-127)} = -0.500000059604644775390625$

(may not be exact in decimal!)

4. (De-Normalized Form): Suppose that an IEEE-754 32-bit floating-point representation pattern is 1 00000000 000 0000 0000 0000 0001.

Sign bit  $S = 1 \Rightarrow$  negative number

$E = 0$  (in de-normalized form)

Fraction is 0.000 0000 0000 0000 0000 0001B (with an implicit leading 0)  $= 12^{-23}$

The number is  $-2^{-23} \times 2^{(-126)} = -2^{(-149)} \approx -1.410^{-45}$

## 9.4 Machine-Epsilon

Machine epsilon,  $\epsilon$ , is the smallest difference between two numbers that can be represented, it determines the maximum possible relative error in representing a nonzero real number  $x$  in a floating-point system:

$$\left| \frac{fl(x) - x}{x} \right| \leq \epsilon.$$

The unit roundoff  $\epsilon$  is determined by the number of digits in the mantissa whereas the underflow level UFL is determined by the number of digits in the exponent field such that:

$$0 \leq UFL \leq \epsilon \leq OFL.$$

One trick for estimating machine epsilon is to compute the following:

$$\epsilon = |3 * ((4/3) - 1) - 1|$$

Do you see why this trick works?

Now that you know that calculations and computations are very different animals that give you different answers, how do you program robust code? Using Invariants! We learned invariants for a reason: you can assert that invariants hold over your loop to guard against scientific computing errors. For example, it may seem trivial to assert that `a` is always positive in the following loop, but if a user gives you the value 2147483647 as an input, this assertion will fail (immediately)!

```
int a;

/* ... Code up here asks for user input for the value of a ... */

while (aircraft_is_flying == 1) {
    a += 1;
    printf("a is %f\n",a);
} /*end while*/
```

Try this code snippet with `a = 2147483647`. Now try adding an `assert` statement to check the loop invariant you expect: that `a` is always positive (and in fact always increasing).

## 9.5 Approximation Errors

Most problems involved in *numerical analysis*, or the computation of continuous mathematics, cannot be solved, even in theory, in a finite number of steps and therefore must be solved by a (possibly infinite) iterative process that ultimately converges to a solution; such problems include those over derivatives, integrals, and nonlinearities. Approximations are used in scientific computation to replace a difficult problem with an easier one that has the same solution or a sufficiently closely-related solution to enable the required analysis. *Error analysis*

is the study of the effects of these approximations on the accuracy and stability of the results.

Here are some useful equations for understanding sources of error in a computation (versus the exact calculation):

**total error = computation error + propagated data error** – Where propagated data error is not affected by the algorithm implemented in the code performing the computation. It is affected by other sources of input errors, such as the difference between the input mathematical model of a system and the actual system.

**computational error = truncation error + rounding error** – Where truncation error is the error due to truncating infinite calculations into finite computations, including integrals, derivatives, nonlinearities, and other problems that involve a theoretically infinite solution process. Rounding error is due to the inexactness in the representation of real numbers and the arithmetic operations performed on them.

### 9.5.1 Floating-Point Arithmetic

To add or subtract two floating-point numbers, their exponents must first match, which is accomplished by shifting the mantissa of one of the numbers. Therefore, some of the trailing digits of the mantissa of the smaller number will be shifted out of the field; if the difference is too great, the smaller number could be lost entirely! If the true sum of two  $t$ -digit numbers contains more than  $t$  digits, then the excess digits will be lost when the result is rounded to  $t$  digits.

To multiply two floating-point numbers, their exponents are summed and their mantissas are multiplied. The product of two  $t$ -digit mantissas can contain up to  $2t$  digits; it will be rounded down to  $t$  digits and some precision will be lost.

**Neither floating point-addition nor floating-point multiplication are associative!**

Let  $e$  be a positive floating-point number slightly smaller than  $\epsilon$ . Then  $(1 + e) + e = 1$ , but  $1 + (e + e) > 1$ . Techniques such as *backwards analysis* can be used to understand this error, but as aerospace engineers you simply need to be aware of this kind of pitfall and the patterns used to avoid it.

**Subtraction can also cause loss of precision.** If two nearly-equal numbers are accurate only to within rounding error, then taking their difference leaves only the rounding error as a result. Let  $e$  be a positive number slightly smaller than  $\epsilon$ . Then  $(1+e) - (1-e) = 1-1 = 0$  in floating-point arithmetic; the result of  $2e$  is completely lost. Because of this *cancellation* error, computing a small quantity as a difference of two large quantities is generally a bad idea.

**Quadratic Formula** Numerical errors occur for long series of computations and for simple ones, like the quadratic equation, where the two solutions of  $ax^2 + bx + c = 0$  are given by

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

For some values of the coefficients, naïve use of this formula in floating-point arithmetic can produce overflow, underflow, or catastrophic cancellation.

For example, if the coefficients are very large or very small then  $b^2$  or  $4ac$  may overflow or underflow. The possibility of overflow can be avoided by rescaling the coefficients, such as dividing all three coefficients by the coefficient of the largest magnitude. Such a rescaling does not change the roots of the quadratic equation, but now the largest coefficient is 1 and overflow cannot occur in computing  $b^2$  or  $4ac$ . Such rescaling does not eliminate the possibility of underflow, but it does prevent *needless* underflow, which could otherwise occur when all three coefficients are very small.

Cancellation between  $-b$  and the square root can be avoided by computing one of the roots using the alternative formula

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}},$$

which has the opposite sign pattern from that of the standard formula. But cancellation inside the square root cannot be easily avoided without using higher precision (if the discriminant is small relative to the coefficients, then the two roots are close to each other, and the problem is inherently ill-conditioned).

For example, we can use four-digit decimal arithmetic, with rounding to the nearest floating-point number, to compute the roots of the quadratic equation having coefficients  $a = 0.05010$ ,  $b = -98.78$ , and  $c = 5.015$ . For comparison, the correct roots, rounded to 10 significant digits, are 1971.605916 and 0.05077069387. Computing the discriminant in four-digit arithmetic produces

$$b^2 - 4ac = 9757 - 1.005 = 9756,$$

so that

$$\sqrt{b^2 - 4ac} = 98.77.$$

The standard quadratic formula then gives the roots

$$\frac{98.78 \pm 98.77}{0.1002} = 1972 \text{ and } 0.0998.$$

The first root is the correctly rounded four-digit result, but the other root is completely wrong, with an error of about 100 percent. The culprit is cancellation, not in the sense that the final subtraction is wrong (indeed it is exactly correct), but in the sense that cancellation of the leading digits has left nothing remaining but previous rounding errors. The alternative quadratic formula gives the roots

$$\frac{10.03}{9.78 \mp 98.77} \text{ and } 0.5077.$$

Once again, we have obtained one fully accurate root and one completely erroneous root, but in each case it is the opposite root from the one obtained previously. Cancellation is again the explanation, but the different sign pattern causes the opposite root to be contaminated. In general, for computing each root we should choose whichever formula avoids this cancellation, depending on the sign of  $b$ .

## 9.6 Type Conversion in C

A type cast is basically a conversion from one type to another. There are two kinds of type conversion:

### 1. Implicit Type Conversion – Also known as "automatic type conversion".

- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place in an expression where more than one data type is present. In such conditions, type conversion (type promotion) takes place to avoid the loss of data.
- All the data types of the variables are upgraded to the data type of the variable with the largest data type.

```
bool -> char -> short int -> int ->
unsigned int -> long -> unsigned ->
long long -> float -> double -> long double
```

- It is possible for implicit conversions to lose information, for signs to be lost (when signed is implicitly converted to unsigned), and for overflow to occur (when long long is implicitly converted to float).

Check a real case about overflow: <https://www.linkedin.com/pulse/12-year-dormant-error-found-just-1474-seconds-yogananda-jeppu/>

- Example of Implicit Type Conversion:

```
/* An example of implicit conversion */
#include<stdio.h>
int main(void) {
    int x = 10;    /* integer x */
    char y = 'a'; /* character c */

    /* y implicitly converted to int. ASCII
       value of 'a' is 97 */
    x = x + y;

    /* x is implicitly converted to float */
    float z = x + 1.0;

    printf("x = %d, z = %f", x, z);
    return 0; /*terminate normally*/
} /*end main*/
```

Output:

```
x = 107, z = 108.000000
```

2. **Explicit Type Conversion** – This process is also called type casting and is user-defined. Here, the user can type cast the result to make it a particular data type. The syntax in C is as follows:

`(type) expression`

Example of Explicit Type Conversion:

```
/* C program to demonstrate explicit type casting */
#include<stdio.h>

int main(void) {
    double x = 1.2;

    /* Explicit conversion from double to int */
    int sum = (int)x + 1;

    printf("sum = %d", sum);

    return 0; /*terminate without error*/
} /*end main*/
```

Output:

sum = 2

## 9.7 Exercises

When grading, the command `make` will be run in the directory, then the requested program name will be run. You are in control of your compiling process via the makefile and have discretion in naming and organizing your source code.

**Exercise 1. (50 Points)** A decimal real number is a number with both an integer and fractional part. The decimal real number is converted to a binary real number by converting both the integer and fractional parts to binary individually. For example,  $194.375_{10}$  can be converted to binary by first converting  $194_{10}$  to binary, and then converting  $0.375_{10}$  to binary. From our examples above, we know that  $194_{10}$  is equivalent to  $11000010_2$ , and  $0.375_{10}$  is equivalent to  $0.011_2$ . Therefore,  $194.375_{10}$  is equivalent to  $11000010.011_2$ .

Write a program named `bindec` that takes as input a decimal number and a flag either `-d` to indicate decimal, or `-b` to indicate binary. It should output the equivalent number in the other system. To make things easier, use at most 10 binary digits for decimal fractions that cannot be represented exactly in binary.

Example program invocations for the above example: `./bindec -d 194.375`

`./bindec -b 11000010.011`

Make sure to validate inputs (for examples, only ones, zeros and optionally a decimal point in binary input mode) and return with a zero exit code on success and non-zero exit code if there was an error.

**Exercise 2. (50 Points)** For your report this week, put the answers to the following questions in your report. Make sure the L<sup>A</sup>T<sub>E</sub>X source is named `main.tex` and is in a subdirectory of your lab repo named `report`.

1. What are the ranges of 8-bit, 16-bit, 32-bit and 64-bit integer, in unsigned and signed representation?

You can use the following L<sup>A</sup>T<sub>E</sub>X table:

```
% Please add the following required packages to your document preamble:
% \usepackage{multirow}
\begin{table}[]
\caption{My caption}
\label{my-label}
\begin{tabular}{llllll} %NOTE: These are "ell" for "left alignment"
    %You could also use "c" for "center" or "p" for "paragraph" --
    % see the manual for many other options!
\multirow{2}{*}{Size} & \multicolumn{2}{c}{Unsigned} & \multicolumn{2}{c}{Signed} & \\
& Min. Value & Max. Value & Min. Value & Max. Value & \\
8-bit & & & & & \\
16-bit & & & & & \\
32-bit & & & & & \\
64-bit & & & & & \\
\end{tabular}
\end{table}
```

2. Give the value of 88, 0, 1, 127, and 255 in 8-bit unsigned representation.
3. Give the value of +88, -44, -1, 0, +1, -128, and +127 in 8-bit 2's complement signed representation.
4. Compute the largest and smallest positive numbers that can be represented in the 32-bit normalized form.
5. Compute the largest and smallest negative numbers that can be represented in the 32-bit normalized form.
6. Repeat (4) for the 32-bit denormalized form.
7. Repeat (5) for the 32-bit denormalized form.
8. Compute the largest and smallest positive numbers that can be represented in the 64-bit normalized form.
9. Compute the largest and smallest negative numbers can be represented in the 64-bit normalized form.
10. Repeat (8) for the 64-bit denormalized form.
11. Repeat (9) for the 64-bit denormalized form.

**Exercise 3. (100 Points)** Write a program to solve the quadratic equation  $ax^2 + bx + c = 0$



using the standard quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

or the alternative formula

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}.$$

Your program should accept values for the coefficients  $a$ ,  $b$ , and  $c$  as command-line input and produce the two roots of the equation as output.

Your program should detect when the roots are imaginary, but need not use complex arithmetic explicitly. You should guard against unnecessary overflow, underflow, and cancellation. When should you use each of the two formulas? Try to make your program robust when given unusual input values. Any root that is within the range of the floating-point system should be computed accurately, even if the other is out of range. Test your program using the following values for the coefficients:

a	b	c
6	5	-4
$6 \times 10^{30}$	$5 \times 10^{30}$	$-4 \times 10^{30}$
0	1	1
1	$-10^5$	1
1	-4	3.999999
$10^{-30}$	$-10^{30}$	$10^{30}$

Your binary should be named `quad`.

**Exercise 4. (100 Points)** Write a program to generate the first  $n$  terms in the sequence given by the difference equation

$$x_{k+1} = 111 - (1130 - 3000/x_{k-1})/x_k,$$

with starting values

$$x_1 = \frac{11}{2} \text{ and } x_2 = \frac{61}{11}.$$

Use  $n = 10$  if you are working in single precision,  $n = 20$  if you are working in double precision. The exact solution is a monotonically-increasing sequence converging to 6. Can you explain your results?

Your program should be called `seq`, accept a command-line flag `-d` for double precision, otherwise use single precision. Print out the  $n$  terms, one per line when run.

### 9.7.1 Lab Submission

*Warning:* As stated in the syllabus, programs and reports that do not compile will be awarded zero points!

If you have issues with your lab work, utilize your resources (lecture notes, text books, lab manuals, personal internet research) and contact a TA for assistance if you are still unable to resolve your issue.

When you are finished with your lab, follow this procedure to turn in your work for grading:

1. Ensure all exercises are in the correctly named files in your lab directory
2. Ensure the source file compiles without errors
3. Ensure that your report is named correctly, is in the correct directory, and builds without errors
4. Use `git status` to verify all the work you want to be graded has been committed to the develop branch
5. Use `git push` to update GitHub with all your changes
6. Open a pull request on GitHub
7. **DO NOT** click the "close and merge" button - your PR should remain open for the autograder to find

Congratulations, you are 64.28571429% of the way though 361!

## 9.8 Sources

- <https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/lab4/#fun-and-interesting-further-reading-on-floats>
- [https://docs.oracle.com/cd/E19957-01/806-3568/ncg\\_goldberg.html](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html)
- [http://horstmann.com/unblog/2011-07-29/Capture-ch02\\_jfe2.png](http://horstmann.com/unblog/2011-07-29/Capture-ch02_jfe2.png)
- <https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/>
- <http://steve.hollasch.net/cgindex/coding/ieeefloat.html>
- <https://stackoverflow.com/questions/10371857/is-floating-point-addition-and-multiplication-associative>
- <https://stackoverflow.com/questions/6360049/what-are-arithmetic-underflow-and-overflow-in-c>

# 10 | The Matrix

*Reference:*

**Introduction to Scientific and Technical Computing:**

Chapter 15: Libraries for Linear Algebra

**C Programming Language, 2nd Edition:**

Section 5.7 – Multi-dimensional Arrays

Matrices often appear in aerospace engineering applications, like in wind tunnel data vector fields, air traffic control radar sweeps, and structural simulations; we are often analyzing data that comes in the form of a matrix. However, software for accessing and reasoning about this data can perform dramatically differently depending on the *order* in which we read the matrices into program memory due to the structure of the computer memory hierarchy. Therefore, it is very important to understand how cache hits and cache misses occur when accessing all matrices, especially in the age of *big data* where the cost of repeated cache misses can exponentially increase the running time of matrix applications – to the point of limiting on-board software from upholding critical timing requirements.

## Objectives

- Develop a basic understanding of the computer memory hierarchy and how it affects data read in, analyzed, and stored by software.
- Learn how arrays of all dimensions are pre-loaded into the cache and how this affects the most pervasive data structure: the matrix.
- Write software that accesses and multiplies two matrices efficiently (and inefficiently).
- Implement basic performance analysis to compare the performance of two programs that *calculate* the same answer but *compute* it differently.

## 10.1 Essential Computer Architecture

With lower-level languages (like C) and real-time performance software that runs on-board aircraft, you need to be aware of the memory hierarchy of the system you are running on. Until now, we assumed we have infinite memory and all memory accesses took equal time. In reality, memory is divided in terms of its distance from the CPU and access times, *aka*,

the memory hierarchy. In modern computer design, memory is built in different layers. This is because the ultimate goals in memory design are to

1. have lots of it (terabytes, petabytes, etc.)
2. make it fast to access (nanoseconds access time)
3. make it cheap (affordable and not too expensive)

The three goals are difficult to achieve simultaneously since fast memory, such as *Static RAM* (SRAM), is very expensive, while cheaper memory, such as *Dynamic RAM* (DRAM), is slower, and the cheapest memory, such as hard drive storage, is extremely slow compared to SRAM or DRAM. A memory system based only on SRAM will be too expensive, while a memory system with only hard drive storage will be extremely slow. Building memory based only on DRAM would soften the price but slow down the overall performance significantly.

### 10.1.1 Memory Hierarchy

To achieve all three goals, hardware designers combine a small amount of expensive, fast memory and large amounts of inexpensive, slow memory in such a way that the combination of the two behaves as if large amounts of fast memory are available, and at an affordable price. To create this illusion of lots of fast memory, we create a hierarchical memory structure, with multiple levels. An example of a structure with 4 levels is shown in Figure 10.1.

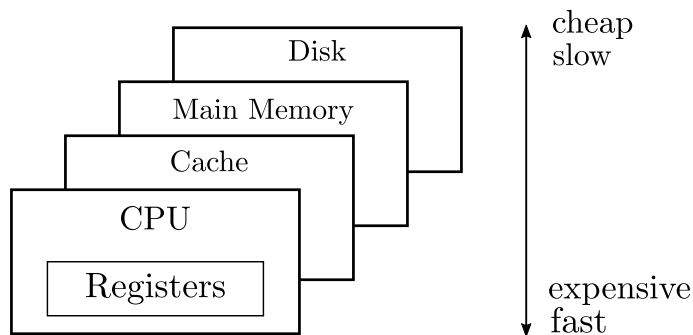


Figure 10.1: Memory Structure Hierarchy of a Modern Computer

Each level in the memory hierarchy contains a subset of the information that is stored in the level right below it:

$$\text{CPU} \subset \text{Cache} \subset \text{Main Memory} \subset \text{Disk}$$

**Registers** in the CPU allow to store information that can be accessed extremely fast. If information is not present in the registers, the CPU will request information from memory by providing the address of the location where the required information is stored. First, the **cache** will verify whether it has the requested information available or not. The cache is located close to the CPU and is composed of a relatively small amount of fast and expensive memory (SRAM). If the requested information is available in the cache, it can be retrieved quickly. If not, **main memory**, which is significantly larger and composed of slower and cheaper DRAM, is accessed. If the requested information is in the main memory, it is provided to the cache, which then provides it to the CPU. If not, the **hard drive**, which

contains all information that is stored in the machine, is accessed. The hard drive offers a vast amount of storage space, at an affordable price, however, accessing it is slow. So, fundamentally, the closer to the CPU a level in the memory hierarchy is located, the faster, smaller, and more expensive it is.

### 10.1.2 Locality

In order to create the illusion of having lots of fast memory available, it is crucial that, with high probability, the cache contains the data the CPU is looking for, such that the main memory and especially the hard drive get accessed only sporadically. Fortunately, not all data in the entire address space is equally likely to be accessed: usually, only a small portion of the entire address space is being accessed over any range of 10-100 lines of code. This is because of locality:

- **Temporal Locality:** recently-accessed memory content tends to get accessed again sooner than other memory. For example, programs with simple loops that cause instructions and data to be referenced repeatedly. In this line of code:

```
for (int i = 0 ; i < 100 ; ++i)
```

the memory location that stores the variable  $i$  will be referenced repeatedly, as well as the locations that contain the sequence of machine instructions that encode the loop.

- **Spatial Locality:** memory content that is located nearby recently-accessed memory content tends to be accessed as well, within the next 10-100 clock cycles. For example, program instructions are usually accessed sequentially, if no branches or jumps occur. Also, reading or writing arrays usually results in accessing memory sequentially.

### 10.1.3 Cache Hits/Misses

By keeping the relatively small amount of data which is most likely to be accessed in the cache (i.e., small, fast memory, close to the CPU), memory access will occur rapidly most of the time, i.e., when the requested information is available in the cache. This is called a **hit**. If the requested information is not present in the cache, called a **miss**, it is copied from the next level down in the hierarchy (in this case, the main memory), in so-called *blocks*. In general, for any two adjacent levels in memory hierarchy, a block is the minimum amount of information that is transferred between them, which can either be present or absent in the upper level (i.e., the level closest to the CPU). A **hit** occurs if the data required by the processor appears in some block in the upper level and a **miss** occurs if this is not the case and the lower level needs to be accessed to copy the block that contains the data requested by the CPU into the upper level (after finding the information at the lower level or an even lower level). Figure 10.2 shows this procedure with the cache as the upper level and the main memory as the lower level. Since the data contained in one block are likely to get referenced soon (again), resulting in several cache hits, the average memory access time is likely to be low. Block size is usually larger for lower levels in the memory hierarchy. We introduce the following definitions to assess memory performance:

- Hit Rate ( $h$ ) = number of hits / number of memory requests
- Miss Rate =  $1 - h$

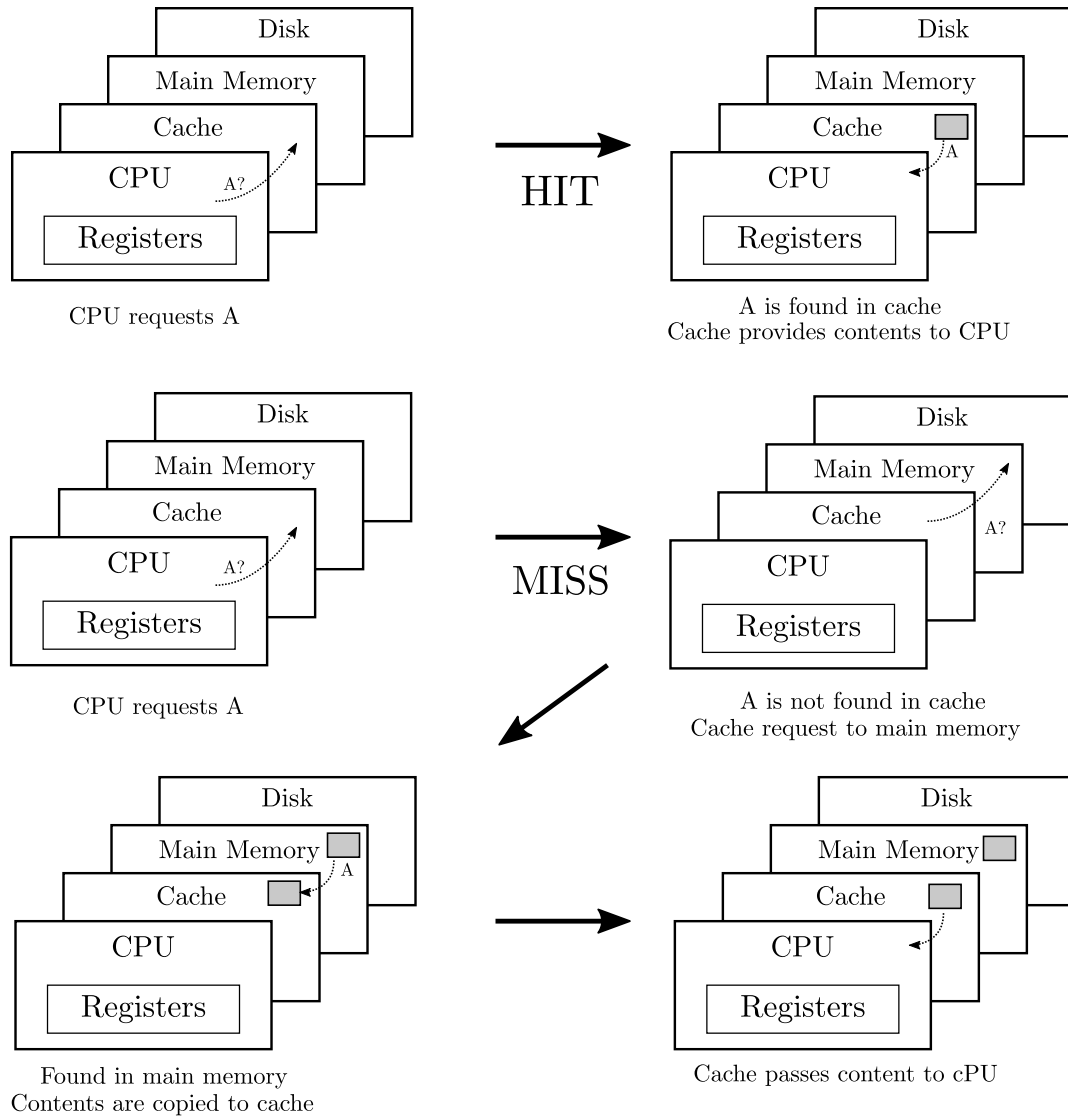


Figure 10.2: Example showing cache *hit* and cache *miss*.

- Hit Time ( $T_h$ ): time to access content in the upper level memory (including time to determine hit or miss)
- Miss Penalty ( $T_m$ ): time to replace a block in upper level memory with the relevant block, retrieved from lower level memory, plus the time the upper level takes to deliver the requested information to the CPU.

Usually,  $T_m$  is significantly larger than  $T_h$ . The average memory access time ( $T_{avg}$ ) can be calculated as

$$T_{avg} = h \times T_h + (1 - h) \times T_m$$

## 10.2 Accessing Matrices and Cache Efficiency

### 10.2.1 A Matrix as a 2-dimensional Array

In C, a Matrix is normally considered as a 2-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional array of size  $[x][y]$ , you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where the **type** is any valid C data type and **arrayName** is a valid C identifier. A two-dimensional array can be considered as a table with x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where “a” is the name of the array, and “i” and “j” are the subscripts that uniquely identify each element in “a”.

#### Initializing Two-Dimensional Arrays

2-dimensional arrays may be initialized by specifying bracketed values for each row. The following is an array with 3 rows and 4 columns. This method of initialization is called **Static Allocation**.

```
int a[3][4] = {
    {0, 1, 2, 3} ,    /* initializers for row indexed by 0 */
    {4, 5, 6, 7} ,    /* initializers for row indexed by 1 */
    {8, 9, 10, 11}    /* initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

In most instances, the C program may be required to create the matrix based on the numbers of row and column provided by the user. As a result, **Dynamic Allocation** is more demanding in practice. There are four different approaches to create the matrix in a dynamic way, and we introduce them via the examples below. In the following examples, we have considered **r** as number of rows, **c** as number of columns and we created a 2-dimensional array with  $r = 3$ ,  $c = 4$  and the following values:

```
1  2  3  4
5  6  7  8
9  10 11 12
```

1. **Using a single pointer:** A simple way is to allocate memory block of size  $r \times c$  and access elements using simple pointer arithmetic. Note that we are declaring a 1D array to the compiler. It is up to the programmer to map the 2D coordinate to the 1D index.

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int r = 3, c = 4; /*We have 3 rows and 4 columns*/
    int *arr = (int *)malloc(r * c * sizeof(int)); /*allocate matrix memory: r*c*/

    int i, j, count = 0; /*declare loop iterators for two loops*/
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) { /*put something in each array element*/
            *(arr + i*c + j) = ++count;
        } /*end for*/
    } /*end for*/

    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            printf("%d ", *(arr + i*c + j)); /*print all array elements*/
        } /*end for*/
    } /*end for*/

    return 0; /*terminate normally*/
} /*end main*/
```

Output:

1 2 3 4 5 6 7 8 9 10 11 12

2. **Using an array of pointers:** We can create an array of pointers of size  $r$ . After creating an array of pointers, we can dynamically allocate memory for every row. This requires knowing one dimension of the array at compile-time.

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int r = 3, c = 4, i, j, count; /*declare 3 rows, 4 columns, loop iterators, and a counter*/

    int *arr[r]; /*declare an array of row pointers*/
    for (i=0; i<r; i++) { /*loop through the row-pointer array*/
        arr[i] = (int *)malloc(c * sizeof(int)); /*allocate a row each time*/
    } /*end for*/

    /* Note that arr[i][j] is same as *((arr+i)+j) */
    count = 0; /*start our count at 0*/
    for (i = 0; i < r; i++) { /*loop through all rows*/
        for (j = 0; j < c; j++) { /*loop through all columns*/
```



```

        arr[i][j] = ++count; // Or (*(arr+i)+j) = ++count
    } /*end for*/
} /*end for*/

for (i = 0; i < r; i++) { /*loop through all rows*/
    for (j = 0; j < c; j++) { /*loop through all columns*/
        printf("%d ", arr[i][j]); /*print each array element*/
    } /*end for*/
} /*end for*/

return 0; /*terminate normally*/
} /*end main*/

```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12
```

3. **Using pointer to a pointer:** We can create an array of pointers dynamically using a double pointer. Once we have an array of pointers allocated dynamically, we can dynamically allocate memory and for every row like method 2.

```

#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int r = 3, c = 4, i, j, count; /*declare 3 rows, 4 columns,
                                    loop iterators, and a counter*/

    /*declare a 2D array of pointers; allocate memory for the rows*/
    int **arr = (int **)malloc(r * sizeof(int *));

    /*loop through the rows and allocate each one to be column-width*/
    for (i=0; i<r; i++) {
        arr[i] = (int *)malloc(c * sizeof(int)); /*allocate one row: c columns*/
    } /*end for*/

    /* Note that arr[i][j] is same as (*(arr+i)+j) */
    count = 0; /*start our count at 0*/
    for (i = 0; i < r; i++) { /*loop through all rows*/
        for (j = 0; j < c; j++) { /*loop through each column in a row*/
            arr[i][j] = ++count; /* OR (*(arr+i)+j) = ++count */
        } /*end for*/
    } /*end for*/

    /*For debugging, we may want to break out the following as a function...*/
    for (i = 0; i < r; i++) { /*loop through all rows*/
        for (j = 0; j < c; j++) { /*loop through all columns in one row*/
            printf("%d ", arr[i][j]); /*print each single array element*/

```

```

        } /*end for*/
    } /*end for*/
    return 0; /*terminate normally*/
} /*end main*/

```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12
```

#### 4. Using double pointer and one malloc call for all rows

```

#include<stdio.h>
#include<stdlib.h>
int main(void) {
    int r=3, c=4; /*declare 3 rows and 4 columns*/
    int **arr;    /*arr is our 2D pointer array*/
    int count = 0,i,j; /*initialize count to 0; declare loop iterators*/

    /*allocate memory for our outer array of pointers (array of rows) */
    arr = (int **)malloc(sizeof(int *) * r);
    /*allocate memory for the first row (row 0) */
    arr[0] = (int *)malloc(sizeof(int) * c * r);

    for(i = 0; i < r; i++) { /*loop through every row*/
        arr[i] = (*arr + c * i); /*each row pointer now points to a column-wide row*/
    } /*end for*/

    for (i = 0; i < r; i++) { /*loop through every row*/
        for (j = 0; j < c; j++) { /*loop through every column*/
            arr[i][j] = ++count; /* OR *((arr+i)+j) = ++count */
        } /*end for*/
    } /*end for*/

    for (i = 0; i < r; i++) { /*loop through every row*/
        for (j = 0; j < c; j++) { /*loop through every column*/
            printf("%d ", arr[i][j]); /*print each individual array element*/
        } /*end for*/
    } /*end for*/

    return 0; /*terminate without error*/
} /*end main*/

```

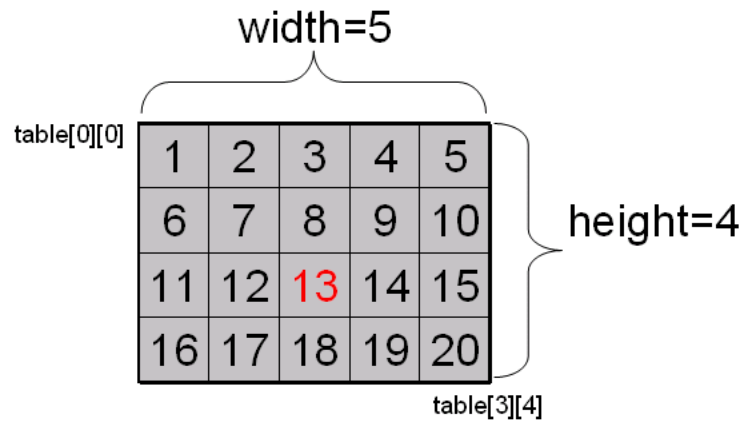
Output:

```
1 2 3 4 5 6 7 8 9 10 11 12
```

In the four examples above, you may ask the user to provide the values of r and c. The fourth approach is the most common way for dynamic allocation of arrays.

## Accessing Two-Dimensional Array Elements

In C, 2-dimensional arrays are essentially stored as 1-dimensional arrays in the computer. When we create a table of integers with 4 rows and 5 columns:



We can reach the elements by using one of the two ways as below:

1. `int element = table[row-1][column-1];`
2. `int element = (*(table+row-1)+column-1);`

In these examples **row** and **column** is counted from 1, that's the reason for the "-1". In the following code you can test that both techniques are correct. In this case, we count the rows and columns from 0.

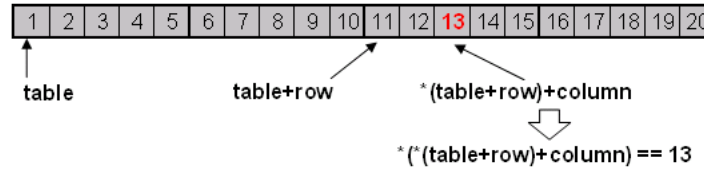
```
#include <stdio.h>
#include <stdlib.h>
#define HEIGHT 4
#define WIDTH 5

int main(void) {
    int table[HEIGHT][WIDTH] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
    int row = 2; /* declare row; initialize to 2*/
    int column = 2; /*declare column; initialize to 2*/
    int a = (*(table+row)+column);
    printf("%d\n",a); /* 13 */
    printf("%d\n",table[row][column]); /* 13 */
    return 0; /*terminate without error*/
} /*end main*/
```

This is double pointer arithmetic, so `table` points to the first row and `*table` points to the first element, if you deref it then `**table` will return the value of the first element. In the following example you can see that `*table` and `table` is pointing to the same memory address.

```
printf("%d\n",table); /* 2293476 */
printf("%d\n",*table); /* 2293476 */
```

```
printf("%d\n",**table); /* 1 */
```



In the memory, all the rows of the table are following each other. Because **table** points to the first row, if we add the row number where the needed element is in the table, we will get a pointer that points to that row. In this case **\*(table+row)** will contain an address to the first element of the given row. Now we just have to add the column number like **\*(table+row)+column**, and we get the address of the element in the given row and column. If we deref this, we get the exact value of this element. So if we count the rows and columns from zero, we can get elements from the table like this:

```
int element = (*(table+row)+column);
```

### 10.2.2 An Evaluation on Matrix Allocation

Please run the following C program (in your lab repository as `examples/matrix_test.c`) on your VM using the following input pairs (the value of `r` and `c`):

1. 3 and 4;
2. 300 and 400;
3. 3000 and 4000;
4. 30000 and 40000;
5. 300000 and 400000;

and test the execution times for different runs with the command "time". An example to run is

```
time ./matrix 3 4 > out
```

Here is the code, with comments explaining each portion of its operation.

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[]) {
    if (argc != 3) { /*check for correct number of arguments*/
        /*print correct usage message and exit*/
        printf ("usage: ./matrix row column\n");
        exit (0);
    } /*end if*/
    int r = atoi (argv[1]), c = atoi (argv[2]); /*ensure int types!*/
    int **arr;
    int count = 0,i,j;
```

```

/*malloc arr and check that the memory was allocated*/
arr = (int **)malloc(sizeof(int *) * r);
if (arr == NULL) {
    printf ("first malloc error\n");
    exit (0);
} /*end if*/
/*malloc arr[0] and make sure that the memory was allocated*/
arr[0] = (int *)malloc(sizeof(int) * c * r);
if (arr[0] == NULL) {
    printf ("second malloc error\n");
    exit (0);
} /*end if*/

for(i = 0; i < r; i++) { /*loop through all rows*/
    arr[i] = (*arr + c * i);
} /*end for*/

for (i = 0; i < r; i++) { /*loop through all rows*/
    for (j = 0; j < c; j++) { /*loop through all columns*/
        arr[i][j] = ++count; /* OR (*(arr+i)+j) = ++count */
    } /*end for*/
} /*end for*/

for (i = 0; i < r; i++) { /*loop through all rows*/
    for (j = 0; j < c; j++) { /*loop through all columns*/
        printf("%d ", arr[i][j]); /*print each array element*/
    } /*end for*/
} /*end for*/

return 0; /*exit without error*/
} /*end main*/

```

Observe the outputs for each set of inputs. What conclusion can you make in terms of the memory allocation introduced in the first section?

By default, the command “time” provides three different times with respect to the input program:

1. “real time” represents the elapsed real time between invocation and termination of the input program;
2. “user time” represents the user CPU time spent on running the input program;
3. “sys time” represents the the system CPU time spent on running the input program;

In this testing program, the bigger gap between real time and user time reflects that more time is spent on memory allocation.

## 10.3 Order matters

As mentioned in lecture, the physical layout of the matrix in memory can drastically increase the time taken by the computer to run your program.

The previous section gave an example of how an array is implemented in memory by the C compiler (note that this can change depending on the programming language) and discussed how *locality* of memory affects the probability of a cache hit. If you think about an algorithm that requires accessing each element in a matrix, going column by column, one row at a time moves the program along the matrix in memory in order – the next element is very local to the previous and highly likely to be in the cache.

Compare this to going row by row, one column at a time – this would cause the program to have to skip an entire row's worth of memory to get to the next element! For all but the smallest matrices this would cause expensive cache misses. To put this in perspective: if each instruction of the CPU took 1 second, checking the cache takes as long as a yawn. But if the data is not there, going to memory takes as long as brushing your teeth. Even worse, if the program has to go to the hard-drive - it would take a weekend if you have a solid-state drive, and a weekend if you have a spinning platter hard-disk!

To demonstrate the difference - look at the program `examples/order_test.c` in your lab repo:

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

#define DIM 1024

/* function col_test
   What does this function do?
*/
int col_test(void){

    size_t i,j;
    int matrix [DIM][DIM];

    for (i=0;i< DIM; i++){
        for (j=0;j <DIM;j++){
            matrix[j][i]= i*j;
        } /*end for*/
    } /*end for*/
    return 0; /*return a 0 to the calling function*/
}

/* function row_test
```

```

    What does this function do?
*/
int row_test(void){
    size_t i,j;
    int matrix [DIM][DIM];

    for (i=0;i< DIM; i++) {
        for (j=0;j <DIM;j++) {
            matrix[i][j]= i*j;
        } /*end for*/
    } /*end for*/

    return 0; /*return a 0 to the calling function*/
}

int main(void) {
    clock_t tic, toc;
    double col_time, row_time;

    tic = clock();
    col_test();
    toc = clock();
    col_time = (double)(toc - tic) / CLOCKS_PER_SEC;

    tic = clock();
    row_test();
    toc = clock();
    row_time = (double)(toc - tic) / CLOCKS_PER_SEC;

    printf("Column Order: %f seconds\n", col_time);
    printf("Row Order: %f seconds\n", row_time);
    printf("Speed change: %g percent\n", ((row_time - col_time)/col_time)*100.00);

    return 0; /*terminate normally*/
} /*end main*/

```

What does this program do? What are the differences between the 2 functions? Compile and run the program - what is the output? Try modifying the define statement to make the matrices smaller, how does this affect the speed difference?

### 10.3.1 Counting Cache Misses

The valgrind tool can be used to see the cache misses responsible for the dramatic slowdown with selection of the `cachegrind` tool like this:

```
valgrind -tool=cachegrind ./a
```

Try compiling and running the `examples/row_order.c` and `examples/col_order.c` programs with `cachegrind` to see how many hits and misses occur on these relatively small matrices.

## 10.4 Exercises

When grading, the command `make` will be run in the directory, then the requested program name will be run. You are in control of your compiling process via the makefile and have discretion in naming and organizing your source code.

As an example of technical computing with matrices in an aerospace application, your lab contains a datafile with *PIV* (Particle Image Velocimetry) results. The file, `PIV.dat`, represents the velocity vector field from a single frame of a wind tunnel test.

*Warning:* We are going to make some pretty rough assumptions – this is not an aerodynamics class, we just want to explore the process of working with this kind of higher-ordered data.

### Exercise 1.

Data Import — (20 points):

The datafile is tab separated (similar to comma separated) and after one header row each line (or record) contains the following fields: X position in mm, Y position in mm, X Velocity in  $\frac{m}{s}$ , Y velocity in  $\frac{m}{s}$ .

Your makefile should produce a program called `piv_vort` (notice the underscore in the name) which will be run with a filename of a datafile as a command line argument, such as `./piv_vort PIV.dat` and will do the following:

1. Read through the datafile, counting the number of data collections and storing the maximum and minimum X and Y positions
2. Use the range and number of points to compute the X and Y spacing
3. Use the number of points to allocate two separate 2D matrices, one for the X velocity and the other for the Y velocity
4. Read through the datafile, storing the velocity components in their corresponding matrix locations
5. Save these two matrices to text files named `Xvel.txt` and `Yvel.txt` respectively. Use your matrix printing code from before. You may want to keep track of the largest number you find as you are importing the velocity data for this.

All data should be treated with double precision and the position measurements should be changed from millimeters to meters during import. You may copy the CSV library from your previous lab or use `scanf` to read the file - whatever you are more comfortable with.



From now on, the X velocity matrix will be called the U matrix, and the Y velocity matrix will be the V matrix.

## Exercise 2.

Computing the Curl — (30 points):

Next we want to find the vorticity of the flow, which we can compute as the curl of the velocity field. Since we are looking at a 2D slice of the flow field, the vorticity equation simplifies to a scalar field:

$$\omega = \left( \frac{\partial V}{\partial x} - \frac{\partial U}{\partial y} \right)$$

To compute this, we will need the spatial derivatives of the velocity fields. Since we don't have a function for these fields, we can approximate the partial derivatives using finite difference. In particular, we will use a 5-point stencil for the numerical differentiation of the field (see [here](#) for derivation if interested). The 5-point stencil approximation of the first derivative in one dimension is:

$$f'(x) \approx \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12 \cdot h}$$

Since we found the spacing between measurements in both directions (here denoted as  $h_x$  and  $h_y$ ) in the last exercise, we can compute the magnitude of the vorticity at any position of the matrix  $(i, j)$  as:

$$\omega(i, j) = \frac{-V(i+2, j) + 8V(i+1, j) - 8V(i-1, j) + V(i-2, j)}{12 \cdot h_x} - \frac{-U(i, j+2) + 8U(i, j+1) - 8U(i, j-1) + U(i, j-2)}{12 \cdot h_y}$$

Two things to note:

1. Because we require two points in every direction of the point in question, we either have to take an assumed value for values off the 'edge' of the matrix, or we can only compute a vorticity field that is 4 points shorter in each dimension. We will choose the 2nd option for the purposes of this lab.
2. Since we will be accessing one matrix row-wise, and the other column-wise, we will consistently miss the cache on one half of the calculation. In our small example this won't be very noticeable, but if we scale up to computing a larger PIV field over many frames, we could easily increase the running time by orders of magnitude. We will cover this in the next exercise.

Extend your code from the last exercise to do the following:

1. Allocate a new matrix, that has size (n-4) compared to the dimensions of the velocity matrices, to hold your vorticity field
2. Use the above equation to compute the vorticity magnitude scalar field

3. Save this matrix to a text file named `Vort.txt` using your matrix-printing code from before. You may want to keep track of the largest number you find as you are computing the curl for this.

### Exercise 3.

Cache Impact — (25 points):

For your report this week, use Valgrind to analyze the cache misses and hits of your code, then modify your data import routine to store one of the velocity matrices transposed (remember to flip your indices in your computation) and analyze it again. In your report, you should answer the following:

1. How does the cache performance and run time of the two solutions differ?
2. Can you see the impact of the transposition?
3. How would this impact larger programs?
4. Take an educated guess about the big-O complexity of this with and without the transposition.
5. How much data would you have to process for this to pay off?

#### 10.4.1 Lab Submission

*Warning:* As stated in the syllabus, programs and reports that do not compile will be awarded zero points!

If you have issues with your lab work, utilize your resources (lecture notes, text books, lab manuals, personal internet research) and contact a TA for assistance if you are still unable to resolve your issue.

When you are finished with your lab, follow this procedure to turn in your work for grading:

1. Ensure all exercises are in the correctly named files in your lab directory
2. Ensure the source file compiles without errors
3. Ensure that your report is named correctly, is in the correct directory, and builds without errors
4. Use `git status` to verify all the work you want to be graded has been committed to the develop branch
5. Use `git push` to update GitHub with all your changes
6. Open a pull request on GitHub
7. **DO NOT** click the "close and merge" button - your PR should remain open for the autograder to find

Congratulations, you are 71.42857143% of the way though 361!

# 11 | Systems of Linear Equations

*Reference:*

**Introduction to Scientific and Technical Computing:**

Chapter 15: Libraries for Linear Algebra

**C Programming Language, 2nd Edition:**

[Section 5.7 – Multi-dimensional Arrays](#)

**Other Required Reading:**

[Naive Gauss Elimination – Trahan, Kaw, and Martin](#)

The Gauss-Jordan Elimination Method (in lab repository)

The Jacobi & Gauss-Seidel Iterative Methods — Dr. Xu (in lab repository)

## Objectives

- Program an analytical solver for linear systems
- Utilize concise matrix representations that compliment engineering data
- Leverage iterative approximation to solve

## 11.1 Matrix Representation of Systems of Linear Equations

The following sections assume familiarity with the matrix representation of systems of linear equations, commonly expressed in the form:

$$[\mathbf{A}] [\mathbf{X}] = [\mathbf{B}]$$

where  $\mathbf{A}$  is a square matrix ( $n \times n$ ) where each row represents an equation and each column the coefficient of one unknown in the system.  $\mathbf{X}$  is a column vector of the unknowns and  $\mathbf{B}$  is a column vector of the constant terms of each equation. Examples of this formulation will be apparent in the following examples.

In engineering systems, large linear systems are very common. However, these systems are also very sparse. Thinking back to statics and dynamics, there are only a few simple equations operating on each component of the system, and these equations will be entangled with the equations of the immediately surrounding components. While in the broader sense, every

component impacts all others, they do so via loose coupling and most of the coefficients in the matrix are actually zero.

When scaled up, this means that significant time and memory can be saved when doing engineering calculations by taking advantage of these sparse matrices. Computer scientists and engineers have developed the requisite algorithms and techniques to capitalize on this, however it is up to you to use them. In this lab you have been given a simplified sparse matrix representation and will use it when solving systems of equations.

## 11.2 Direct Methods

### 11.2.1 Graphical

The *graphical method* is the simplest of methods for solving linear equations, and as the name suggests, is done using plotting on a graph. The method is useful when solving a small number of equations ( $n \leq 3$ ) and does not require a computer.

Consider two linear equations with two variables  $x_1$  and  $x_2$ .

$$\begin{aligned}a_{11}x_1 + a_{12}x_2 &= b_1 \\a_{21}x_1 + a_{22}x_2 &= b_2\end{aligned}$$

A graphical solution can be obtained for the two equations by plotting them on Cartesian coordinates with one axis for  $x_1$ , and the other for  $x_2$ . Note that since we are dealing with linear systems, each equation is a straight line and can be rewritten as  $x_2 = (\text{slope})x_1 + \text{intercept}$ . The values of  $x_1$  and  $x_2$  at the intersection of the two lines represent the solution.

For three dimensional equations, each equation would represent a plane in a three-dimensional coordinate system. Beyond three equations, the graphical method breaks down and has little value for solving simultaneous equations.

### 11.2.2 Cramer's Rule

*Cramer's Rule* is another solution technique that is best suited to a small number of equations. Consider the equations:

$$\begin{aligned}a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3\end{aligned}$$

The equations can be represented as  $[A]\{X\} = \{B\}$  where  $[A]$  is the coefficient matrix:

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

The determinant  $D$  of this system is formed from the coefficients of the equations.

$$D = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

The rule states that each unknown in a system of linear equations may be expressed as a fraction of two determinants with denominator  $D$  and with the numerator obtained from  $D$  by replacing the column of coefficients of the unknown in question by the constants  $b_1, b_2, \dots, b_n$ . For example  $x_1$  can be computed as:

$$x_1 = \frac{\begin{vmatrix} b_1 & a_{12} & a_{13} \\ b_2 & a_{22} & a_{23} \\ b_3 & a_{32} & a_{33} \end{vmatrix}}{D}$$

Similarly,  $x_2$  and  $x_3$  can be computed as

$$x_2 = \frac{\begin{vmatrix} a_{11} & b_1 & a_{13} \\ a_{21} & b_2 & a_{23} \\ a_{31} & b_3 & a_{33} \end{vmatrix}}{D} \quad x_3 = \frac{\begin{vmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ a_{31} & a_{32} & b_3 \end{vmatrix}}{D}$$

For more than three equations, Cramer's rule becomes impractical! As the number of equations increases, the determinants are time-consuming to evaluate by hand (or by computer).

### 11.2.3 Elimination Method

The elimination of unknowns by combining equations is an algebraic approach. For example, consider two equations:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \end{aligned}$$

The basic strategy is to multiply the equations by constants so that one of the unknowns ( $x_1$  or  $x_2$ ) can be eliminated. The result is a single equation that can be solved for the remaining unknown. For example, multiply the first equation by  $a_{21}$  and the second equation by  $a_{11}$  to give

$$\begin{aligned} a_{11}a_{21}x_1 + a_{12}a_{21}x_2 &= b_1a_{21} \\ a_{21}a_{11}x_1 + a_{22}a_{11}x_2 &= b_2a_{11} \end{aligned}$$

Subtracting the above two equations eliminates  $x_1$  to yield

$$a_{22}a_{11}x_2 - a_{12}a_{21}x_2 = b_2a_{11} - b_1a_{21}$$

that can be solved for  $x_2$ .

$$x_2 = \frac{a_{11}b_2 - a_{21}b_1}{a_{11}a_{22} - a_{12}a_{21}}$$

The computed value for  $x_2$  can then be substituted in either of the original equations to get the value of  $x_1$ .

The elimination method can be extended to systems with more than two or three equations. However, the numerous calculations required for larger systems become tedious to be done by hand. *Naive-Gauss* elimination formalizes the elimination method for larger systems of equations and can be readily programmed for the computer.

### Naive-Gauss Elimination

The elimination method described above consists of two steps:

1. *Elimination Step*: The equations are modified to eliminate one of the unknowns.
2. *Back-substitute*: The equation with one unknown is solved directly and the result is back-substituted into one of the original equations.

The basic approach can be extended to deal with large sets of equations by developing a formal mechanism. *Gauss Elimination* is one such formalism. The approach is designed to solve a general set of  $n$  equations over  $n$  unknowns.

$$\begin{array}{ccccccc} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \dots + a_{1n}x_n & = & b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \dots + a_{2n}x_n & = & b_2 \\ \vdots & & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \dots + a_{nn}x_n & = & b_n \end{array}$$

**Required Reading.** For a pseudo-code description of Naive Gauss Elimination, please see: *Naive Gauss Elimination* by Jamie Trahan, Autar Kaw, Kevin Martin [http://mathforcollege.com/nm/simulations/nbm/04sle/nbm\\_sle\\_sim\\_naivegauss.pdf](http://mathforcollege.com/nm/simulations/nbm/04sle/nbm_sle_sim_naivegauss.pdf).

#### 11.2.4 Limitations of Elimination Methods

While many systems can be solved with elimination methods, there are some pitfalls that must be explored before implementing the method.

##### 1. Division by Zero

During both the elimination and back-substitution phases, it is possible that a division by zero can occur. For example, if we use naive Gauss elimination to solve

$$\begin{aligned}5x_2 + 2x_3 &= 6 \\x_1 + 3x_2 + 7x_3 &= -5 \\6x_1 + 7x_2 + 2x_3 &= 9\end{aligned}$$

the normalization of the first row would involve division by  $a_{11} = 0$ . Problems also arise when the coefficient is close to zero.

## 2. Rounding Errors

We are limited by the number of significant figures we use in our calculation when using the Gauss elimination method. For example, if we round-off a calculation during the elimination step (due to a fixed number of significant figures), the round-off error propagates to other unknowns during the back-substitution phase. The problem of round-off error can become particularly important when large numbers of equations are to be solved. This is due to the fact that every result is dependent on previous results. **A rough rule of thumb is that round-off error may be important when dealing with 100 or more equations. In any event, you should always substitute your answers back into the original equations to check whether a substantial error has occurred.**

## 3. System Condition

*Ill-conditioned systems* are those in which small changes in the coefficients lead to large changes in the solution. For such systems, a wide range of answers can (approximately) satisfy the equations. Because round-off errors can induce small changes in the coefficients, these small changes can lead to large solutions errors. For example, consider the system of equations

$$\begin{aligned}x_1 + 2x_2 &= 10 \\1.1x_1 + 2x_2 &= 10.4\end{aligned}$$

Solving the above equations, yields  $x_1 = 4$  and  $x_2 = 3$ . However, if we change the value of  $a_{21}$  from 1.1 to 1.05 the result is changes to  $x_1 = 8$  and  $x_2 = 1$ !

### 11.2.5 Gauss-Jordan

The Gauss-Jordan method is a variation of the Gauss elimination. The major difference is that when an unknown is eliminated in the Gauss-Jordan method, it is eliminated from all other equations rather than just the subsequent ones (if you haven't already, please read *Naive Gauss Elimination* by Jamie Trahan, Autar Kaw, Kevin Martin [http://mathforcollege.com/nm/simulations/nbm/04sle/nbm\\_sle\\_sim\\_naivegauss.pdf](http://mathforcollege.com/nm/simulations/nbm/04sle/nbm_sle_sim_naivegauss.pdf)). The elimination step results in an identity matrix rather than a triangular matrix. Consequently, it is not necessary to employ back-substitution to obtain the solution.

**Required Reading.** *Gauss-Jordan Elimination Method* (with examples): Available in the lab repository.

### 11.2.6 LU Decomp & Solve

While the methods explored so-far provide valuable insight into the comparative strengths and weaknesses of computation when directly solving linear systems, if you ever need direct solutions in production, you should make use of a battle-tested *LU Decomposition* solver. The BLAS (Basic Linear Algebra Subprograms) routines have been used in scientific and supercomputing since the late 70s, and can be used with everything from C and FORTRAN to Python.

## 11.3 Approximate Methods

While direct methods can arrive at an exact solution, this is not always possible (or requires more computing resources than feasible). To solve systems that are non-linear, ill-conditioned, or otherwise arduous – approximate numerical methods are available. These techniques provide a cornerstone for modern computer-aided engineering, as they are necessary in almost every sub-field.

The most common approximate methods are iterative, meaning they start from an initial guess and progressively refine their solution. In the best case, this can be repeated to get arbitrarily close to the exact answer. Such a method is said to "converge."

### 11.3.1 Jacobi Method & Gauss-Seidel

In your lab repository is a pdf file, courtesy of Professor Zhiliang Xu at the University of Notre Dame. It describes the theory of the Jacobi and Gauss-Seidel methods of numerical solutions for systems of linear equations.

By the nature of these methods, they side-step the usual round-off error we were cautious of with the direct methods. Instead, the error is a product of the starting location and number of iterations - much easier to control than computer rounding. You can also quickly and simply get rough numbers even on many poorly-posed problems which can be useful for first round design.

Read the theory of operation for the two methods and note the provided pseudo code for both methods. Engineers are often required to apply the body-of-knowledge of one discipline to their own; aerospace engineers will often implement algorithms designed by computer engineers, scientists or mathematicians to improve their simulations and analysis tools. You will implement a Gauss-Seidel solver as described below based on the algorithm described in the pdf.

## 11.4 Exercises

When grading, the command `make` will be run in the directory and should produce a `gauss-jordan` and a `gauss-seidel` program for the respective exercises. You are in control of your compiling process via the makefile and have discretion in naming and organizing your source code.

### Exercise 1.



Naive-Gauss — (15 points):

In your L<sup>A</sup>T<sub>E</sub>X report, give the time complexity for the Naive-Gauss elimination in terms of the number of equations ( $n$ ). You can assume that  $n$  equations are available to solve for  $n$  unknowns.

**Hint:**

Try writing out pseudo-code for this. How does the number of lines executed increase as you increase  $n$ ?

**Exercise 2.**

Gauss-Jordan — (15 points):

Starting with the (mostly complete) `gauss_jordan.c` file in your lab repo - fill in the necessary lines to create a Gauss-Jordan elimination solver. You may modify the template in any way you see fit (including replacing the existing code), however, your program will be expected to accept and return data in the same format as the template. An example of the expected behavior:

How many equations: 3

Enter the elements of augmented matrix row-wise:

```
A[1][1]:10
A[1][2]:-7
A[1][3]:5
A[1][4]:9
A[2][1]:3
A[2][2]:6
A[2][3]:0
A[2][4]:-9
A[3][1]:9
A[3][2]:3
A[3][3]:-2
A[3][4]:-1
```

The solution is:

```
x1=0.224806
x2=-1.612403
x3=-0.906976
```

**Exercise 3.**

Gauss-Seidel — (45 points):

Implement a Gauss-Seidel solver by following the algorithm laid out in Professor Xu's design document. Your code will be run with either:

- A filename of a CSV file containing the augmented matrix

- No arguments, which will cause it to prompt the user for the number of equations and their coefficient values

and either create a CSV file of the column-vector solution named `ans.csv` or a print-out of the answer respectively. To be clear, you are writing one program that is capable of both interfacing with computers via files, and humans via the CLI - not one or the other.

Additionally, your code should have sane defaults for iteration and error limits, but should allow them to be overridden by command line options.

This program should also make use of sparse matrix storage (you should anticipate sparse arrays, they are common in engineering problems) and the `COO.c` file in the lab repo contains sparse matrix code written in lecture for your use.

Use double float precision for all floating math. Document assumptions and decisions you made as the engineer on this project.

After creating your sparse matrix, compare the number of non-zero elements to the number of elements of a regular matrix and report the space savings as a percentage by printing to standard error (not standard out!) like this:

Compressed matrix contains  $n$  elements, compressed by  $x\%$

So if there are 22 equations (with 22 unknowns), there will be a  $22 \times 23$  augmented matrix. Assuming that there are only 197 non-zero elements, the space savings is  $1 - \frac{192}{22 \cdot 23}$  and the standard error output would be:

Compressed matrix contains 197 elements, compressed by 61%

The point breakdown is as follows:

**15 points** Correctly solve a human-entered system and print the results

**15 points** Correctly read an input CSV, solve the system, and save the output to `ans.csv`

**15 points** Use the compressed matrix form and report compressed size savings to standard error as noted above

### 11.4.1 Lab Submission

*Warning:* As stated in the syllabus, programs and reports that do not compile will be awarded zero points!

If you having issues with your lab work, utilize your resources (lecture notes, text books, lab manuals, personal internet research) and contact a TA for assistance if you are still unable to resolve your issue.

When you are finished with your lab, follow this procedure to turn in your work for grading:

1. Ensure all exercises are in the correctly named files in your lab directory

2. Ensure the source file compiles without errors
  3. Ensure that your report is named correctly, is in the correct directory, and builds without errors
  4. Use `git status` to verify all the work you want to be graded has been committed to the develop branch
  5. Use `git push` to update GitHub with all your changes
  6. Open a pull request on GitHub
  7. **DO NOT** click the "close and merge" button - your PR should remain open for the autograder to find
- Congratulations, you are 78.57142857% of the way though 361!

## 12 | Pair Programming

*Reference:*

**C Programming Language, 2nd Edition**

**GNU Plot Manual:** <http://www.gnuplot.info/documentation.html>

### Objectives

- Learn to collaborate on programming tasks and integrate work in git.
- Reinforce good programming practices with respect to functions, pointers, libraries, data structures, input sanitation, and debugging with gdb.
- Gain experience with gnuplot and the automatic plotting from C code.
- Have fun programming a game with intermediate results plotted to keep score.

### 12.1 Introduction to gnuplot

Gnuplot is a linux application, and as the name suggests it is used for plotting data. It offers several customizable features and is very easy to use. In the first part of this lab, we focus on basic plotting with **gnuplot**. To start using gnuplot, type **gnuplot** in the terminal. As a first step, we want you to be aware how we want the plots to be generated. For this lab, we will use **.png** as the default file format output for the plots. See the commands below for an example on usage of this:

```
gnuplot> set terminal png
gnuplot> set output 'plot.png'
```

The above command sets the desired output format to **.png** and the name of our generated plot will be **plot.png**.

#### 12.1.1 Basic Plotting

In our first graph we want to plot a sine and a cosine functions. Therefore, we specify our functions and plot them.

```
gnuplot> a = 1.3
gnuplot> f(x) = a * sin(x)
gnuplot> g(x) = a * cos(x)
```

```
gnuplot> plot [0:10] f(x) title 'sin(x)' with lines linestyle 1, \
> g(x) notitle w l ls 2
```

The definitions of functions in gnuplot are straightforward. We want to plot more than one function, and that's why we have to divide the two commands with a comma. The backslash tells gnuplot that we have a line break at this position. We can also abbreviate commands ("w" for with, etc.). The result of the command is shown in Figure 12.1.

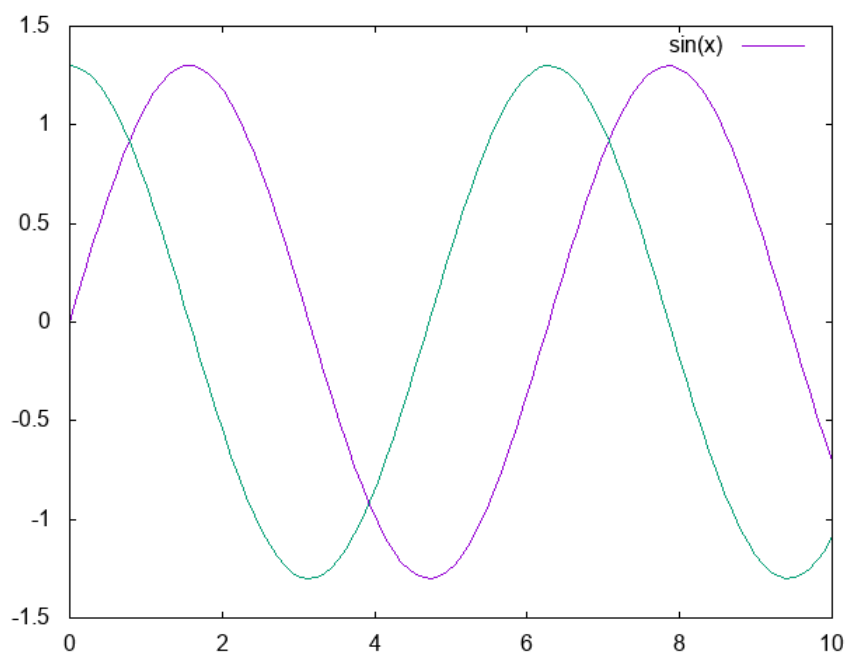


Figure 12.1: Plot for sine and cosine functions

We can also use gnuplot in non-interactive mode by saving the plotting commands to a file. For example, to plot the sine and cosine functions, create a new file named `simpleplots.gp` with the following contents.

```
a = 0.9
b = 1.3
f(x) = a * sin(x)
g(x) = b * cos(x)
plot [0:10] f(x) title 'sin(x)' with lines linestyle 1, \
g(x) title 'cos(x)' w l ls 2
```

Run `gnuplot` with the above file by calling the command below from the terminal:

```
gnuplot simpleplots.gp
```

### 12.1.2 Plotting Data

Plotting data, e.g., the results of calculations or measurements, using `gnuplot` works the same way as plotting functions. We need a data file and commands to manipulate the data. Let's start with the basic plotting of simple data. A data file can be a plain text

file containing the data points as columns. Create a file with the data below and save it as `dataplot.dat`

```
# dataplot.dat
#   X   Y   Z
    1.  1.  1.
    2.  4.  8.
    3.  9. 27.
    4. 16. 64.
    5. 25.125.
```

A line starting with `#` is a comment and is ignored by `gnuplot`. A command to plot the data in the files is:

```
gnuplot> plot 'dataplot.dat' using 1:2 with linespoints, \
>           '' u 1:3 w lp
```

The above command will generate the plot shown in Figure 12.2.

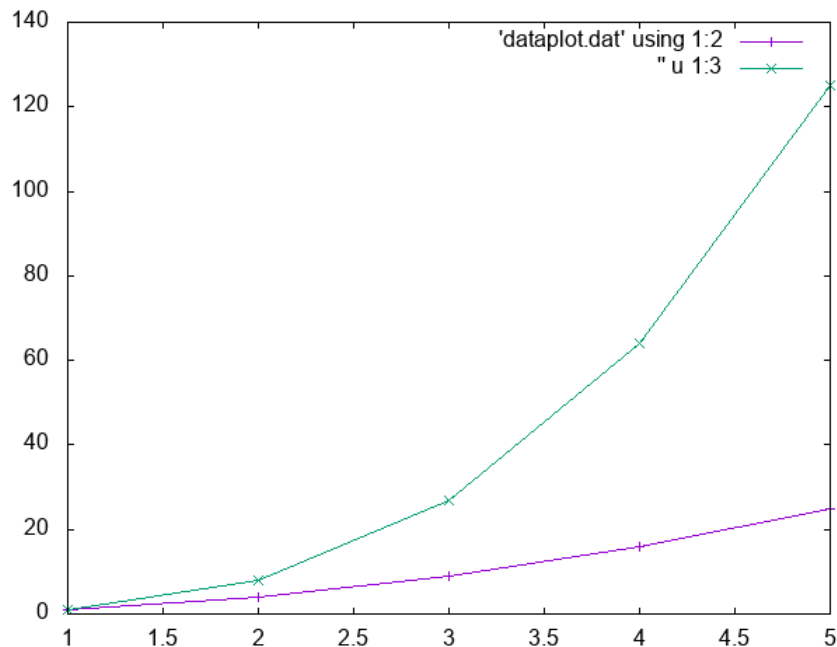


Figure 12.2: Plot for data in `dataplot.dat`

For advanced usage of `gnuplot` and available options, please refer to <https://www.usm.uni-muenchen.de/CAST/talks/gnuplot.pdf>. Typing `help` at the `gnuplot` prompt is also very useful.

### 12.1.3 `gnuplot` Hints

- `gnuplot` can print text-based pictures to your terminal - useful for monitoring a running program or development

- The slides linked above have an example of using color maps — this will be helpful for the project this week
- By exporting snapshots of the data being plotted, then combining the images, you can make animations of your data!
- The project requests plotting in the terminal — remember to correctly set the terminal type to `dumb`

## 12.2 System Calls in C

It is possible to run all linux commands, and other programs, from within a C program. For example, we can call ‘ls’ from within a C program to get the contents of the directory from where the C program is running. You can run anything in linux from your C code! In C, the two most popular commands that allow us to make system calls are `system()` and `popen()`.

### 12.2.1 `system()`

Using `system()`, we can execute any command that can run on the terminal (if the operating system allows). For details and usage of the `system()` command, refer to the man page for `system`.

### 12.2.2 `popen()`

The `popen()` function is closely related to the `system` function. It executes the command as a subprocess. However, instead of waiting for the command to complete, it returns a file pointer to the stream containing the output of that command. For details and usage of the `popen()` command, refer to the man page of `popen`.

## 12.3 Unions and the void

Interfacing with the library provided in the lab will require use of two new types in C: unions and void pointers. This is a brief overview and you are encouraged to make use of the man pages, the C programming book, and other trustworthy resources as needed.

### 12.3.1 Unions

A union is similar to a struct, but where a struct provides an “and” relation between the members, a union is an “or” relation between its members. For example the following program has a struct which contains 3 members, named `a`, `b`, and `c`.

```
#include <stdio.h>

struct myStruct {
    int a;
    float b;
    char *c;
};

int main() {
```

```

    char mystr[] = "Hello, world!";
    struct myStruct foo = { 1, 2.3, mystr};
    printf("%d\t%g\t%s", foo.a, foo.b, foo.c);

    return 0;
}

```

A union type can only contain one of the listed types at a time, for example:

```

#include <stdio.h>

union myUnion {
    int a;
    float b;
    char *c;
};

char mystr[] = "Hello, world!";

int main() {

    union myUnion foo;

    foo.a = 1;
    printf("%d", foo.a);

    foo.b = 2.3;
    printf("\t%g", foo.b);

    foo.c = mystr;
    printf("\t%s", foo.c);

    return 0;
}

```

Notice how we needed to specify which type of data we were storing by using the dot syntax at assignment? Since we can only store one value at a time, what happens if we try to read `foo.a` after writing `foo.c`?

It is common to see unions as members of structs where an enum in the structure tells the reader which variable is stored in the union. Consult the C programming book (free access through the ISU library) for more discussion on unions.

### 12.3.2 Void Pointers

Similar to how a union gives the programmer control over what type is stored in a variable at runtime instead of compile time, a void pointer allows the type of a pointer to be controlled by the programmer. A void pointer represents a pointer where the compiler does not know



the type. To use a void pointer, you must tell C what type to read and write when use use it with a cast:

```
int main()
{
    int a = 10;

    void *ptr = &a; /*Void pointer pointing at variable a */
    printf("%d\t", *(int *)ptr); /* Reading the void pointer with a cast */
    *(int *)ptr = 42; /* Writing the void pointer with a cast */
    printf("%d\n", a); /* Reading var a, changed by the void pointer */

    return 0;
}
```

For more on void pointers, see the C programming book.

## 12.4 Project: Drone Surveillance Challenge

The goals of this project are two fold:

- Tackle some trickier memory usage issues – but you get 1 or 2 partners to assist
- Provide a more realistic workflow, with multiple developers on a project
- Develop a program that has a drone navigate through a virtual environment, answering challenges as it moves

### 12.4.1 The Rules

The object of the drone/agent surveillance game is to guide a drone/agent to survey the majority of a square grid world. The agent begins in a random location of the grid-world and can see every adjacent square, forming a 3-by-3 "vision" matrix centered around them. Cells with a value of 1 contain obstacles and cannot be entered. During each turn, the agent function is called with a structure containing the current vision matrix and information about the challenge to be solved. If the agent answers the challenge correctly, it is allowed to move.

Moves are indicated by the numbers 1-4 corresponding to the cardinal directions, beginning with north and proceeding clockwise. So to move south, an agent would return '3' and to move west it would return '4'. An agent only moves if they correctly answer the challenge and request a valid move (not into an obstacle).

The game engine tracks all cells that have been seen by the agent. The number of rounds and size of the world can be altered by command-line arguments.

### 12.4.2 The Tools

The main function of the program is included in the compiled `libsimulation.a` library. You should not include one in your code. By modifying the `DroneWars.c` code and utilizing the `DroneWars.h` header, you will write:

1. An "agent" function that takes in a structure (defined in the header file) with game information, place the required data at a location pointed to by the answer pointer, and then return the desired move
2. A display function that takes a file pointer to the map of the agent's explored area and uses `gnuplot` to display a map in the console with obstacle cells colored differently than free cells. One of the many ways to accomplish this is to write a `gnuplot` script and call the program with `popen`.

### 12.4.3 Running Your Program

For this project, we intend for the game to play automatically when the program is executed. In the provided makefile, you will see a target called "run." Typing "make run" will compile and execute the program, the game will play out on its own.

### 12.4.4 The Twist

Much of the meat of this exercise comes from determining how to interface with the game from the header file - navigating the structures, unions, enums and pointers that it entails. You have a partner or partners, the internet, your classmates, textbooks and a TA available as resources — make good use of these. Remember to use GDB and the debugging techniques from lab 6 when debugging pointer usage.

### 12.4.5 Challenges

**Calculation** Given a string of a mathematical expression, return the integer value (Hint: Use the function `popen()` and the program `bc`)

**Sub-String** Given 2 strings, return a pointer to the beginning of the first occurrence of that string (Hint: Use `strstr()`)

**Range** Given two integers, return the range between them as an integer

**Mean** Given an array of floats and the number of floats in the array, return the arithmetic mean as a float

**Minimum** Given an array of floats and the number of floats in the array, return the minimum value as a float

**Maximum** Given an array of floats and the number of floats in the array, return the maximum value as a float

**Reverse** Given a string, return a string that is the reverse (Hint: Use `strlen()`)

**Find** Given a string and a char, return the integer index of the position of the first occurrence (Hint: Use `memchr`)

**Tokenize** Given a string, tokenize it by whitespace and return a pointer to a malloced 1-D array of the token strings. Hint: Use `strtok`)

### 12.4.6 Player 2 Has Joined the Game

Version control software like git really shines when used to manage group code. Git automatically mitigates many of the issues having separate copies can cause by keeping them up-to-date without adding bugs or losing progress. However, this is not without a learning curve of it's own. When more than one developer is committing to the same repository, follow these guidelines:

1. Always pull before committing
2. Commit and push often
3. Coordinate with your partner(s) to work on different parts of the code

The last point sounds very restrictive, but logically if you and your partner(s) make contradictory changes, how does the software know which version to use?

If Git cannot merge your code automatically, you will be left in a "merge conflict" state. At this point you should use this excellent resource to better understand what to do next: <https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts>  
If you still cannot resolve the situation, contact a TA for assistance.

### 12.4.7 Deliverables

When finished, submit a pull-request with your code filled into DroneWars.c and any additional files you need along with a L<sup>A</sup>T<sub>E</sub>X report detailing things you learned and difficulties you encountered working through the lab. As per the syllabus, programs that fail to compile will receive zero points.

Glitter points will be awarded at the discretion of the TAs for groups that use gnuplot to make animations of their agents' searches. These are meant to encourage you to have fun, and be creative! We want to see what you're able to develop!

### 12.4.8 Points

There will be 10 points awarded for each successfully completed challenge type (90 total from these), 10 points awarded for the display function, and finally 10 points for the report for a total of 110 points.

## 12.5 L<sup>A</sup>T<sub>E</sub>X Report

For this lab, we expect your report to contain a title page, a references sheet, and details about the what you learned and the difficulties you encountered while working through this lab. While not required, I also recommend explaining what your code does. You may also describe how the work was allocated amongst the team members.

### 12.5.1 Lab Submission

*Warning:* As stated in the syllabus, programs and reports that do not compile will be awarded zero points!

If you having issues with your lab work, utilize your resources (lecture notes, text books, lab manuals, personal internet research) and contact a TA for assistance if you are still unable to resolve your issue.

When you are finished with your lab, follow this procedure to turn in your work for grading:

1. Ensure all exercises are in the correctly named files in your lab directory
2. Ensure the source file compiles without errors
3. Ensure that your report is named correctly, is in the correct directory, and builds without errors
4. Use `git status` to verify all the work you want to be graded has been committed to the develop branch
5. Use `git push` to update GitHub with all your changes
6. Open a pull request on GitHub
7. **DO NOT** click the "close and merge" button - your PR should remain open for the autograder to find

Congratulations, you are 85.71428571% of the way though 361!

# 13 | Circuit Solver

*Reference — C Programming Language, 2nd Edition:*  
Chapter 6 – Structures  
Section 6.6 – Table Lookup

## Objectives

- Show understanding of loop invariants
- Show understanding of calculating the rough (Big O) worst-case complexity of your code.

## 13.1 Motivation

Looking at the applications of numerical methods in an engineering context, using computers to solve large linear systems link in lab 11 can be used for:

- Steady-State Analysis of Chemical Reactors
- Statics Solutions of Large Structures
- Current and Voltage Prediction of Electrical Circuits
- Spring-Mass Modeling of Structural Dynamics

And more! While as aerospace engineers you are most likely also interested in the structural statics and dynamics solutions, here we present an exercise in resistor network solutions as the governing dynamics are simpler – providing a chance to focus on the core algorithm rather than getting stuck in complexities from the problem domain. When you are done, your code will be easily adapted to other (potentially more complex) domains.

### 13.1.1 Circuit Representation

We will use a CSV format for representing the circuit to be solved. Consider first the circuit in Figure 13.1. It is a resistor network to be solved, and your program should be able to read the corresponding csv form of this circuit.

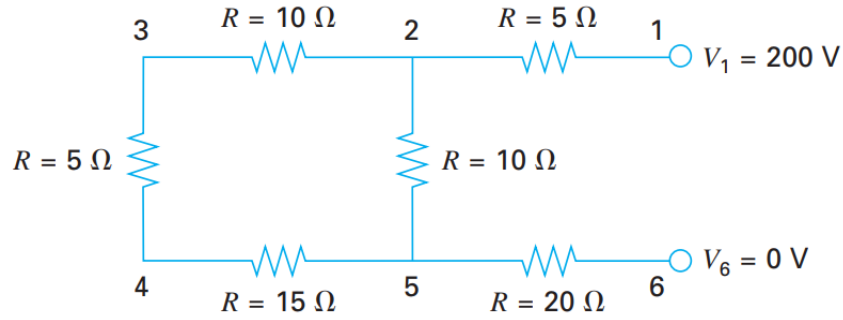


Figure 13.1: A resistor circuit to be solved.

The CSV representing the circuit in Figure 13.1 is shown here:

```
# Type | Node 1 | Node 2 | Value
V,6,1,200
R,1,2,5
R,3,2,10
R,4,3,5
R,5,4,15
R,5,2,10
R,6,5,20
```

Your program should compute all current values for the given circuit.

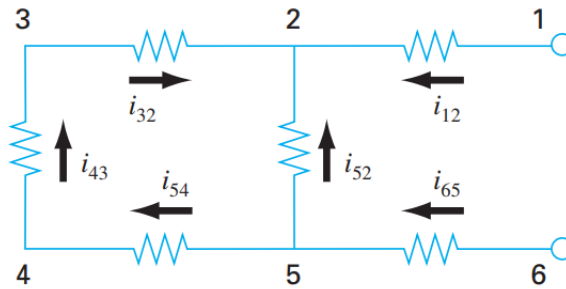


Figure 13.2: Assumed current.

The output of your circuit solver is also a CSV file, in which all current values should be listed. An example of the output of your solver for the above circuit would be:

```
# Type | Node 1 | Node 2 | Value
V,6,1,200
I,1,2,6.1538
I,3,2,-1.5385
I,4,3,-1.5385
I,5,4,-1.5385
I,5,2,-4.6154
```

1, 6, 5, -6.1538

Note that the direction matters and current is assumed to flow from node 1 to node 2. So for current flowing against that direction it should be negative, as shown above.

We explain shortly how to compute the current values in Figure 13.2. A common problem in electrical engineering involves determining the currents and voltages at various locations in resistor circuits. These problems are solved using Kirchhoff's current and voltage rules. The current (or point) rule states that the algebraic sum of all currents entering a node must be zero, or

$$\sum i = 0$$

where all current entering the node is considered positive in sign. The current rule is an application of the principle of conservation of charge. The voltage (or loop) rule specifies that the algebraic sum of the potential differences (that is, voltage changes) in any loop must equal zero. For a resistor circuit, this is expressed as

$$\sum \xi - \sum (i \cdot R) = 0$$

where  $\xi$  is the emf (electromotive force) of the voltage sources,  $i$  is the current at a given resistor and  $R$  is the resistance of the same resistors on the loop. Kirchhoff's voltage rule is an expression of the conservation of energy.

Given the assumptions in Figure 13.2, Kirchhoff's current rule is applied at each node to yield

$$\begin{aligned} (1) \quad & i_{12} + i_{52} + i_{32} = 0 \\ (2) \quad & i_{65} - i_{52} - i_{54} = 0 \\ (3) \quad & i_{43} - i_{32} = 0 \\ (4) \quad & i_{54} - i_{43} = 0 \end{aligned}$$

Application of the voltage rule to each of the two loops gives

$$\begin{aligned} -i_{54}R_{54} - i_{43}R_{43} - i_{32}R_{32} + i_{52}R_{52} &= 0 \\ -i_{65}R_{65} - i_{52}R_{52} - i_{12}R_{12} - 200 &= 0 \end{aligned}$$

or, substituting the resistances from Figure 13.1 and bringing constants to the right-hand side,

$$\begin{aligned} (5) \quad & -15i_{54} - 5i_{43} - 10i_{32} + 10i_{52} = 0 \\ (6) \quad & -20i_{65} - 10i_{52} + 5i_{12} - 200 = 0 \end{aligned}$$

Therefore, the problem amounts to solving the following set of six equations with six unknown currents (pulled from the six numbered equations above, in order):

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 10 & -10 & 0 & -15 & -5 \\ 5 & -10 & 0 & -20 & 0 & 0 \end{bmatrix} \begin{bmatrix} i_{12} \\ i_{52} \\ i_{32} \\ i_{65} \\ i_{54} \\ i_{43} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 200 \end{bmatrix}$$

Although impractical to solve by hand, this system is easily handled using an elimination method. Proceeding in this manner, the solution is

$$\begin{aligned}i_{12} &= 6.1538, i_{52} = -4.6154, i_{32} = -1.5385 \\i_{65} &= -6.1538, i_{54} = -1.5385, i_{43} = -1.5385\end{aligned}$$

## 13.2 Project

The assignment is to produce a project that takes in a CSV file defining a resistor network and outputs a CSV file detailing the currents in that network. Several simplifications will be in-place as detailed in this section.

### 13.2.1 Solving

You may use any numerical method of solving the linear system you choose. Compute all answers with double precision. For your convenience the GNU Scientific Library (GSL) is included along with an example C file showing how to solve linear systems in the form  $\mathbf{A}x = \mathbf{B}$  but you may use either of your solvers from lab 11.

In a realistic setting, you would be required to identify the loops of the circuit yourself. However, since cycle detection is both well studied and yet difficult to properly implement, your input file will be annotated with the loops labeled to assist in creation of the voltage law equations, as demonstrated in the next section. **Importantly, there can be any number of loops and the loops can be of any length (number of nodes).**

Also, assume only one voltage source (i.e., there will be 2 nodes with a "V" label and they can be assumed to be the 2 terminals of a battery or power-supply for purposes of solving the circuit.) See the above worked example for how this is applied in practice.

### 13.2.2 Input

A CSV file defining 2 voltage nodes and a number of resistors with values indicating resistance and the listed order indicating the "positive" direction of current flow through that edge. Therefore current flowing against that direction is "negative" like in the worked example. **Importantly, the nodes will be numbered sequentially starting from 1 with no missing integers.**

Also a number of rows indicating the loops for the voltage law will be included at the end of the file. As an example, the two loops from the above problem would be indicated by:

L,2,3,4,5  
L,1,2,5,6

Where the end is assumed connected to the beginning node.



### 13.2.3 Output

The output should be a near copy of the input with the types changed from resistance (in the form "R") to current with 4 decimal places (in the form "I" with a value of the form "X.XXXX") and preservation of the loop rows optional. This output should be saved to a file as directed in the grading section.

### 13.2.4 Grading

The auto-grader will generate a CSV file that matches the requirements of this lab and run the following two commands, in order:

1. `make`
2. `./kirchhoff_solve ./path/to/file.csv`

in the top level of the lab repository. Note that `kirchhoff_solve` is called with the path to the file as the only command-line argument.

The `kirchhoff_solve` file can either be a C program that does all the work, or a bash script that does any required pre-processing, building of the C program and output formatting if desired. The bash-script methods allows you to harness the powerful shell tools used in the first few labs to reduce the amount of string parsing needed in your C code.

After running, the auto-grader will check the lab repository for a file called `ans.csv` that replaces the resistance values in the input file with the current values computed by your code.

### 13.2.5 Report

Your report this week is worth 10 points. It should contain the following two sections, each worth 5 points:

- Complexity Analysis: What is the complexity of your code relative to the number of resistors in the circuit?
- Loop Analysis: What are the major loops contributing to that complexity? What are the invariants of these loops? Which loops (if any) do not contribute to complexity?

### 13.2.6 Lab Submission

*Warning:* As stated in the syllabus, programs and reports that do not compile will be awarded zero points!

If you have issues with your lab work, utilize your resources (lecture notes, text books, lab manuals, personal internet research) and contact a TA for assistance if you are still unable to resolve your issue.

When you are finished with your lab, follow this procedure to turn in your work for grading:

1. Ensure all exercises are in the correctly named files in your lab directory
2. Ensure the source file compiles without errors
3. Ensure that your report is named correctly, is in the correct directory, and builds without errors
4. Use `git status` to verify all the work you want to be graded has been committed to the develop branch
5. Use `git push` to update GitHub with all your changes
6. Open a pull request on GitHub
7. **DO NOT** click the "close and merge" button - your PR should remain open for the autograder to find

Congratulations, you are 92.85714286% of the way though 361!