COMPUTING FOR EFFICIENT ENGINEERS

Presented by the HKN Epsilon Epsilon Chapter
1 October 2020

This workshop is **not** an introduction to programming, nor does it claim to review all of the tools that would be of immediate benefit to you. Rather, it aims to touch upon built-in functionality of your computer and demonstrate the use of programs that can be useful in programming applications.

I have formatted this document as a script (the kind you read off of), but have taken care to mark **important terms** with *emphasis*. I don't really have a formatting guideline, but mind the words that stand out visually!

THE COMMAND LINE

⚠️ Windows users: Please install Debian or Ubuntu using Windows Subsystem for Linux! ⚠️

The command line is where we're going to stay for most of this workshop. Some of you may have interacted with this before, while others have probably dismissed it as outdated. It is literally just a black box, with no buttons, but it provides a complete interface to the contents of your computer, both files and programs. What the command line interface (CLI) lacks in presentation, it makes up for with control and simplicity. The CLI is a powerful tool that is capable of more than you think it is, and can even be used as a programming environment. The CLI does make easy things a bit more difficult, but it also makes difficult things possible if you know how to use it

Many modern GNU/Linux systems use bash, the Bourne-Again Shell. As an aside, the `.sh` extension is short for *shell*. Because bash is the default shell for so many systems, shell scripting is sometimes referred to as *bash scripting*, but all shells support a scripting language. These shells provide essential functionality such as tab completion, command history search, and piping. Some MacOS users may be using zshell - be advised that zshell and bash have separate syntaxes.

Navigating your Filesystem

Let's look at how to interact with files from the command line. The very first thing we should do is get oriented. After starting up a new terminal, type in `pwd` to print your present working directory. This will print an **absolute path**, which displays how to get to that directory from your root folder. If you just started up a terminal instance, this path will be your *home directory*, and it is represented with a `~`. You can now type `ls` to view all files in your current directory. You can add the `-a` *flag* to display hidden files, which start with a `.`. You may see some more files pop up, but don't worry about them. They're mainly there to store cache data or application configuration settings. You might not have anything in your home directory yet, especially if you're using WSL. In that case, you can make a folder with `mkdir` or create a file with `touch`, supplying both commands with a name. If you want to create nested directories, you can add the `-p` flag. Now you can call `cd` to change directory. You can use an absolute path or a **relative path**, which just means the path from your present working directory. To use a relative path, simply type `./` before the folder. The `cd` command actually defaults to relative

paths, so you can also omit the `./` if you just want to change directories. We've covered moving around a pretty static filesystem, apart from adding directories and files. What if we want to change what's already there? We can use the `mv` command to move both files and directories around. In fact, `mv` doubles as a fast way to rename files.

This may not strike you as extremely efficient. If you've got a bunch of files, do you really need to move them all one-by-one? You don't, and you shouldn't.

Regular Expressions and Brace Expansions

Regular expressions describe patterns in text. The UNIX regular expression search function is called `grep`. `grep` uses several key characters to describe particular patterns in text. A common one is the asterisk, which means any character, any number of times. Another one is the period, which means any character, exactly once. Regular expressions come in handy if your file names have some sort of structure to them. As an example, let's modify this test directory to put all the files from folder 2 into folder 1. You can do this with `mv 2/* 1/`.

Regular expressions are all about taking a set of strings and deriving a common pattern from them. In contrast, a brace expansion takes in a common pattern and produces a set of strings. This allows you to easily create multiple files at once. As an example, we can create a new folder and populate it with the command `touch {i..z}`. The brace expansion also works with numerical arguments.

Shell Scripting

With the basics out of the way, let's look at how to create a shell script with control flow. This is a programming language, but you should recognize the syntax

Arguments and Environment Variables

You have the ability to define, initialize, and reference variables inside your command line! Variables are always prepended with a '$' when being referenced. For example, your home directory is stored in the environment variable HOME and can be printed with the command `echo \$HOME`. Similarly, arguments to a function are accessed with `\$1, \$2, \$3, ...`. Shell scripts themselves have to be defined in the current shell. Most programs we execute from the shell are located in the system **path**, which is the set of all directories that contain executables. Usually when you install a program, the installation procedure will put a `.sh` file in one of those directories.

Let's use this to create a function that makes a directory, then immediately changes into it. We first create a script called `mkcdir.sh`, which we place in the workshop directory. The contents of the file are listed below:

```
mkcdir() {
  mkdir -p -- "$1"
  cd -P -- "$1"
}
```

This creates a function that takes in 1 argument, creates a directory of that name, then changes into that directory. To execute this shell script, we need to *source* it by calling `. mkcdir` or `source mkcdir` within the workshop directory. This will execute the program, effectively loading its functionality into the shell. After this is complete, you can call `mkcdir` without an error.

There are several special parameters that are defined in the context of a function and its arguments.

Command	Function
`$@`	Expands to the positional parameters. Effectively `$1, $2, $3, ...`
`$#`	Expands to the number of positional parameters
`$?`	Expands to the exit status of the most recently executed command. 0 if unsuccessful
`$$`	Expands to the process ID of the shell
`$0`	Expands to the name of the shell or shell script

These conveniently solve a problem with our last function. `mkcdir` can only create a single directory! With a slight tweak in parameter use and an additional for loop, we now have

```
mkcdir() {
  for d in "$@"; do
    mkdir -p -- "$d"
  done
  cd -P -- "$1"
}
```

Conditional Expressions

Bash has support for binary and unary logical operators. A common one is `-f`, which is a unary operator that checks to see if a file exists and is a regular file. You can see this logic implemented in your `.bashrc` file:

```
if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi
```

The `.bash_aliases` file contains all the aliases used by bash. You can list the current aliases with the command `alias -p`.

Useful packages

In addition to bash scripting and shell commands, the CLI also gives you access to every single piece of software installed on your machine (that are also in PATH). It can be daunting reading all of the documentation, so I recommend you install `tldr` to get some examples of how to use some of these applications. I've listed a few below.

Command	Functionality
`pdftk`	Manipulate pdf contents and join pdf documents
`convert`	Converts images between formats. Super handy for pdf/jpg conversion
`youtube-dl`	Downloads content from a YouTube link. Can be configured to only download audio if you want to download a song/album

Command	Functionality
`xdotool`	Command line automation for interacting with your desktop. Can control mouse/keyboard inputs as well as window appearance
`xbindkeys`	Rebinds keys to sequences of keys or scripts. Can also rebind mouse inputs
`man`	Prints documentation of a program. Very handy reference material

The last command is especially important. `tldr` only gets you so far, and the `man` page of a program will help you better understand how that program works.

Remarks

This has been a brief primer on Bash, and is not intended to be comprehensive. You can find the official documentation page here, which is hopefully a bit more readable now that you have seen some examples.

TEXT EDITORS

When I first came to UH, the first engineering software I used was MATLAB, which came with its own Integrated Development Environment, or IDE. When I took the introductory CS courses, more IDEs. PyCharm, CLion, CodeBlocks, Notepad++, VSCode, Atom. Installing these was a pain, and learning how to use them was equally painful. Do you know what a workspace in Code Composer Studio is? I don't, and I honestly don't think I'm supposed to. During these courses, all I could think was –

Why.

Why are there so many. Have these people not heard of notepad?

Of course, a good IDE will have a lot more features than notepad. A good editor should have syntax error flags, autocomplete, an integrated terminal . . .

Wait. Hold on a second. What's the point of creating a fully fleshed-out GUI just to include the ubiquitous black box that most ECE students dismiss? And to make matters more confusing, all of these IDEs print their *output* to this integrated terminal. So you have to use the terminal. It's literally just sitting there, a glorified trash can for all the garbage you pipe through the text streams. But nobody told me why it was there. Everyone just told me to click the green play button and ignore the error messages.

No more. I'm done. If this is how people did it back then, then that's how we're going to do it right now. This is your introduction to the olden days of command line text editors, and the never-ending struggle to not use the mouse.

Editor Wars

Let's just get this out of the way. There are two text editors. There's Emacs, and then there's Vim. The people that have neither probably don't use a command line editor, or they use Nano. Nano is the default editor for GNU/Linux, and has the same keystrokes as Emacs.

Let's get another thing out of the way. I use Emacs. This is out of preference for GNU software, not a preference for one editor over the other.

This section will be very short in terms of code, since I'll mainly try to explain the appeal of both editors. Functionally, both accomplish the goal of editing text very well, although it takes practice. That's the other reason why this section is short. You probably need to spend a good chunk of time with an editor to become proficient with it.

Emacs and Vim

Emacs has many, many different configurations. Each of these configurations is called a *major mode*, and each major mode can have multiple *minor modes*. After installing emacs, run it from the command line with `emacs -nw`. This causes emacs to run within your terminal instead of opening a separate window. If you press `ctrl-h t`, a tutorial will start and you can follow it.

Vim is also a modal editor. You can toggle between modes by pressing different keys. Vim also has a tutorial that you can follow.

Here are some commands that you might find useful

Action	Vim Command	Emacs Command
Change theme	`:colorscheme [colorscheme]`	`M-x customize-themes` + `M-x C-x`
Add line numbers	`:set number`	Add `(setq linum-format "%4d \u2502 ")` and `(global-linum-mode)` to your .emacs file
Add word wrapping	`:set wrap linebreak nolist`	Add `(global-visual-line-mode t)` to your .emacs file

GIT

Have you ever worked in a team for a programming assignment? If not, how would you distribute code to a person that wanted to look at it?

I remember sending code through email for ENGI 1331. Every time anything changed, a new email would be sent and I would have to re-download the file *again*. This is not a good system for collaborative coding. Maybe you looked into solutions and discovered **version control**, which is basically a fancy term for email evasion. In case you didn't, I'll make it easy for you: the only version control system you should bother learning is Git.

You may recognize Git from GitHub, a popular code sharing platform that integrates well with Git. You can create a free account on GitHub, create a repository, add collaborators, and then . . . and then what? Then you should probably keep reading, because I'll go over the structure of Git, as well as how to use it from the command line.

First off, what is Git? Git is a software tool that allows people to work on code at the same time. This is useful for group projects. Linus Torvalds found value in Git when the group project he was working on got a little out of hand. Git is formally known as a **distributed version control system**, which means that content is distributed to individuals locally, instead of relying on a steady connection between a server and a programmer. This means

you can work offline, then upload your code remotely whenever. Other version control systems are delta-based, meaning that they only track changes from one version to the next. Git just stores every single file. If you want an earlier version, you can have it in memory, ready to go.

Git Basics

You can install git on Linux by running `sudo apt-get install git-all`. Git must be configured with your name and email. You can set this with the commands

```
1 git config --global user.name "John Doe"
2 $ git config --global user.email johndoe@example.com
```

You can check your git configuration settings with the command `git config --list`.

Programmers collaborating through Git all access the same codebase, or **repository**. A repository is just a server that collaborators can retrieve and send code. You can create a Git repository from a local directory, or clone it from elsewhere. If you are creating a repository locally, you'll want to run `git init` inside your directory. However, we'll be cloning a directory from GitHub. Type the command `git clone https://` to download the repository I have made for this workshop.

Using Git

There are some key concepts to understand about Git before we start typing commands. Git only keeps track of files that were included in the last *snapshot*. These files are called **tracked**, and they can be either unmodified, modified, or staged. You can use the command `git status` to check which files are in which state.

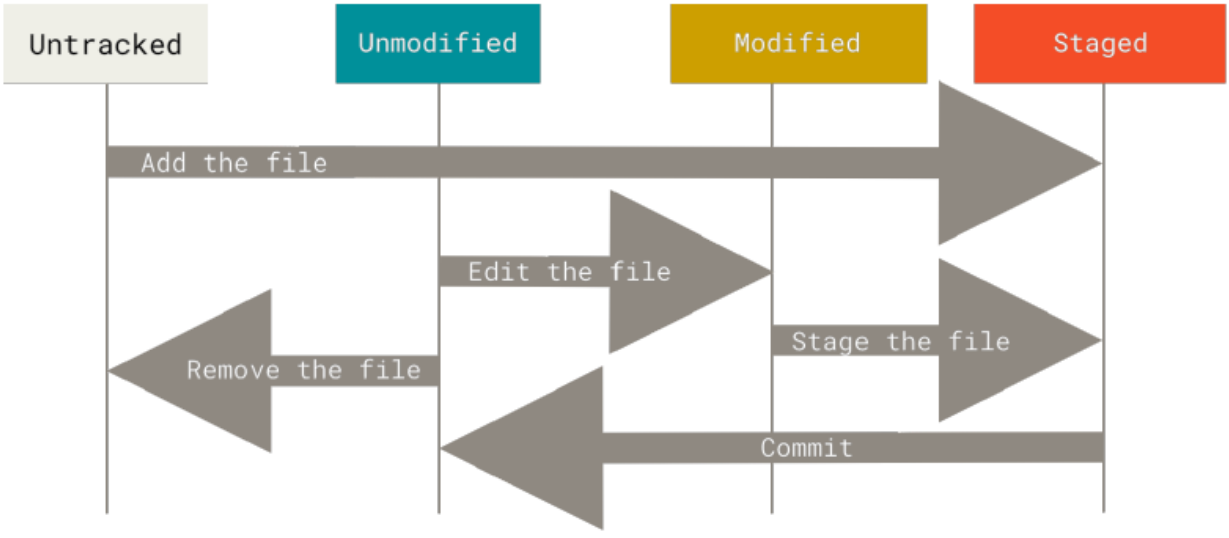


Figure 8. The lifecycle of the status of your files.

If you have an untracked file, then that means it is in your repository, but Git doesn't know about it. To alert Git of its presence, run the command `git add` with the filename. This will add the file to the **staging area**. Run `git status` after doing this, and you should see that Git has recognized the file, and is waiting for you to **commit** your changes.

The `git commit` command will take a snapshot of all the files in your staged area, merge it with the current snapshot, and produce a new snapshot. That snapshot then goes into your local Git repository's collection of snapshots. You will get an error message if you do not include a message. A message is required! Add a message with the `-m` flag.

A convenient shortcut for skipping the staging area is to include the `-a` flag to the `commit` command. This will put every tracked file into the staging area before committing, essentially treating the commit as if you manually entered every single file into the staging area.

Removing a file is a two-step process: Remove it from your staging area, and commit. If you simply remove it from your directory, that change won't be staged yet. You'll have to also stage that change before committing. The `git rm` command takes care of removing the file from your directory and staging it.

Once you've got a remote repository cloned, you can **push** new code to it or **pull** code from it. The command for pushing is `git push origin master`. The command for pulling is simply `git pull`. Git push will update the remote repository you are pushing to with all of the changes you have made. Git pull will update your local Git project with all changes made to the repository, and also tries to merge it with your code.

Notoe that this assumes your local Git project was initialized with a remote repository named *origin*. You can see which remote servers you've configured by running the `git remote` command. You can also add the `-v` flag to see the URL that each repository is linked to.

Just like bash, Git supports aliasing. You can create aliases with the `git config --global alias` command. For example, you can define an alias `unstage` with `git config --global.unstage 'reset HEAD--'.`

Branching

The core feature of Git is its branching system. Git branching allows programmers to diverge from a 'main' branch, work on separate issues, then re-combine those branches. This branch is simply a pointer to the most recent snapshot. You can create a new branch with the `git branch` command, followed by a branch name. By default, this branch will also be a pointer to the most recent snapshot. Git keeps a special pointer called **HEAD** that points to the current branch of your repository.

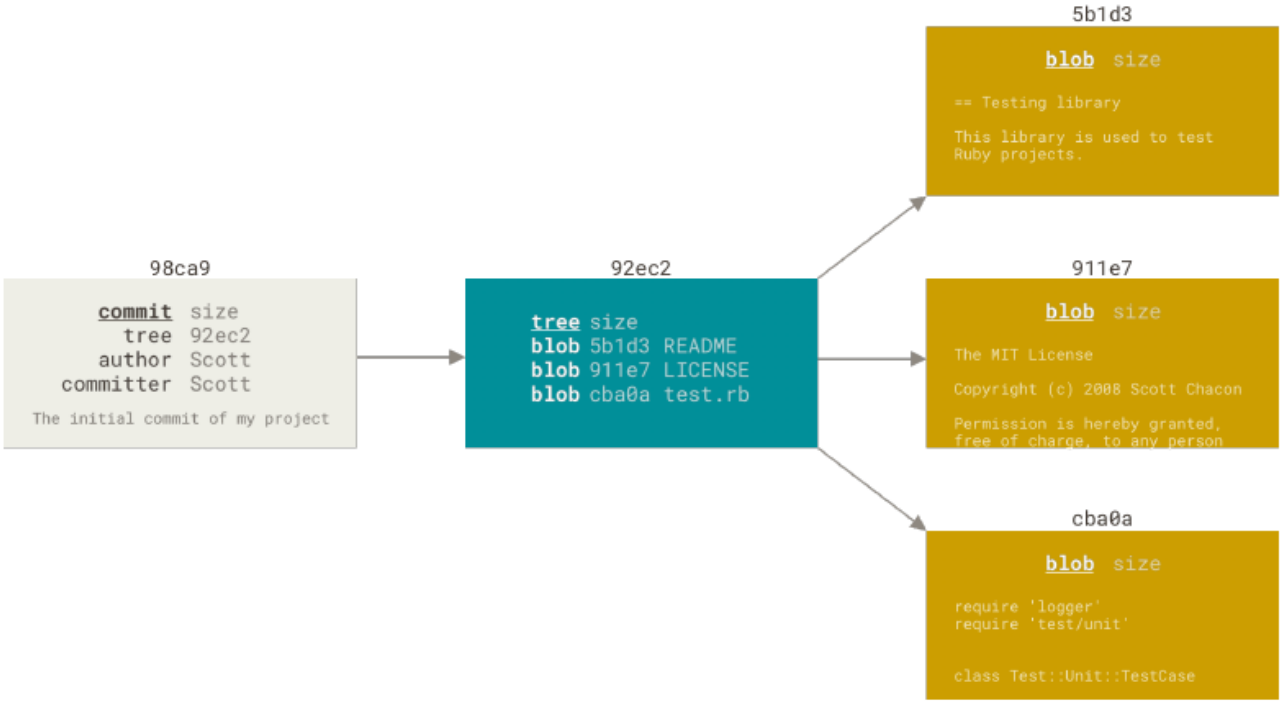


Figure 9. A commit and its tree

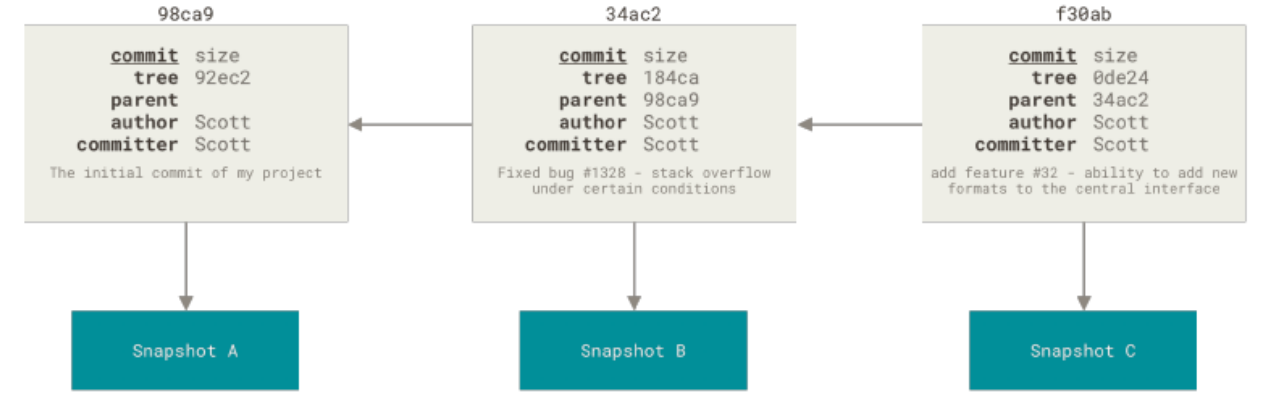


Figure 10. Commits and their parents

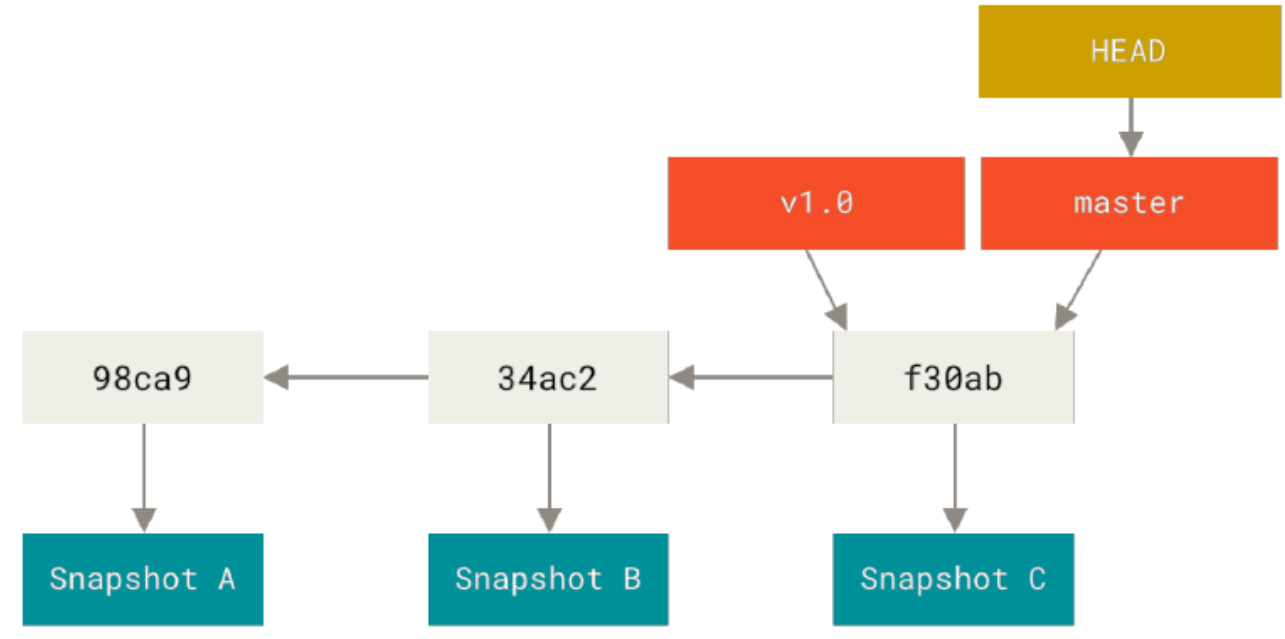


Figure 11. A branch and its commit history

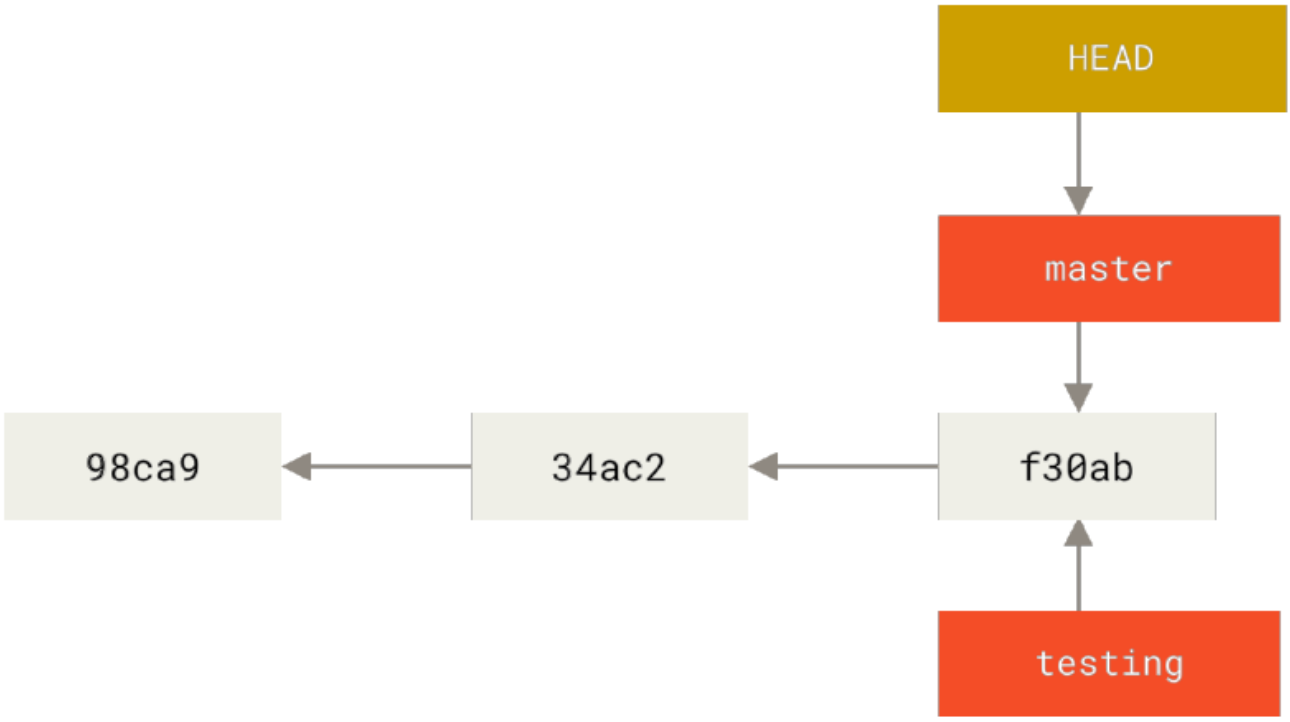
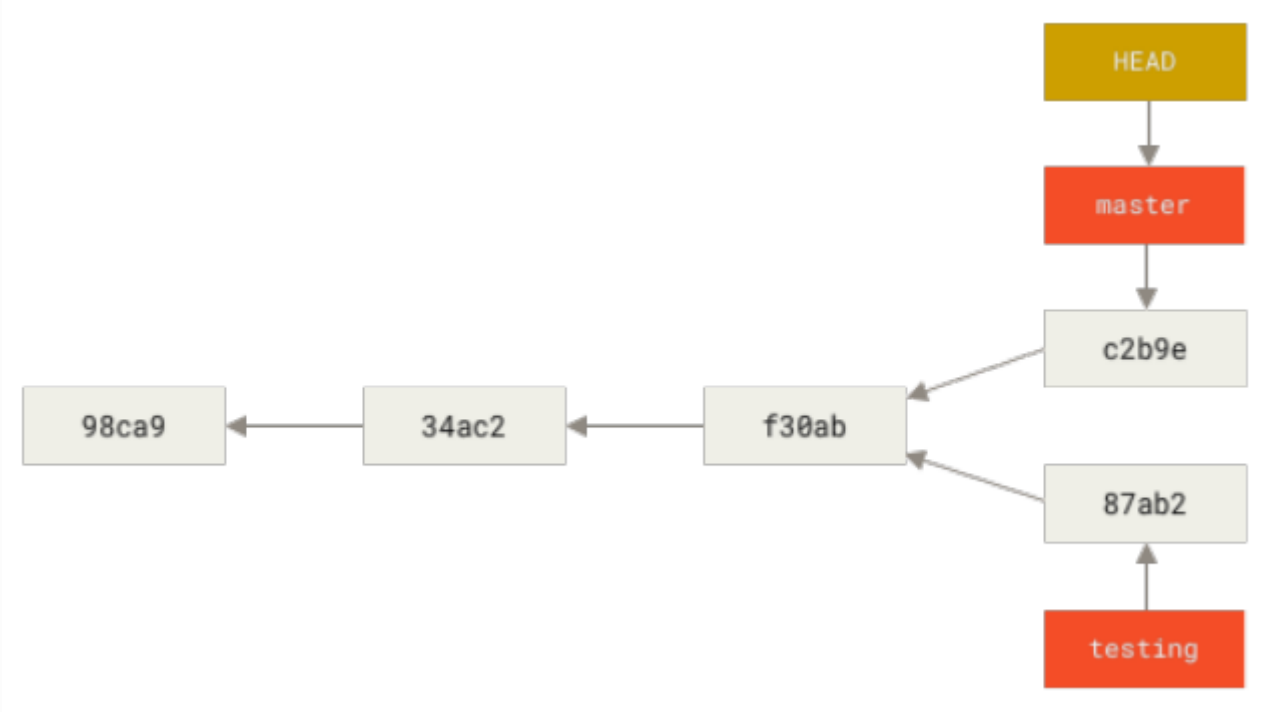


Figure 13. HEAD pointing to a branch

Let's say you create a new branch called 'testing' with this command. Git won't redirect HEAD to point to this new branch. You have to do it yourself with the command `git checkout [branch name]`. Now whenever you commit, the master branch still points to the same tree. The testing branch will point to the new commit tree. If you switch back to the master branch wiht `git checkout master`, the files in your working directory revert back to the files stored in the master branch. The files stored in the testing branch are safe, but they are not accessible from this branch. If Git cannot do this cleanly, it won't let you switch back.



Merging Branches

To keep things simple, Git requires that all conflicts between branches be committed when switching between them. When you create a new branch and commit all of the necessary files to it, you can then merge that branch with the master branch by calling

```
git checkout master
git merge [branch name]
git branch -d [branch name]
```

You can safely delete the new branch because your master branch points to the exact same spot. However, consider the following scenario:

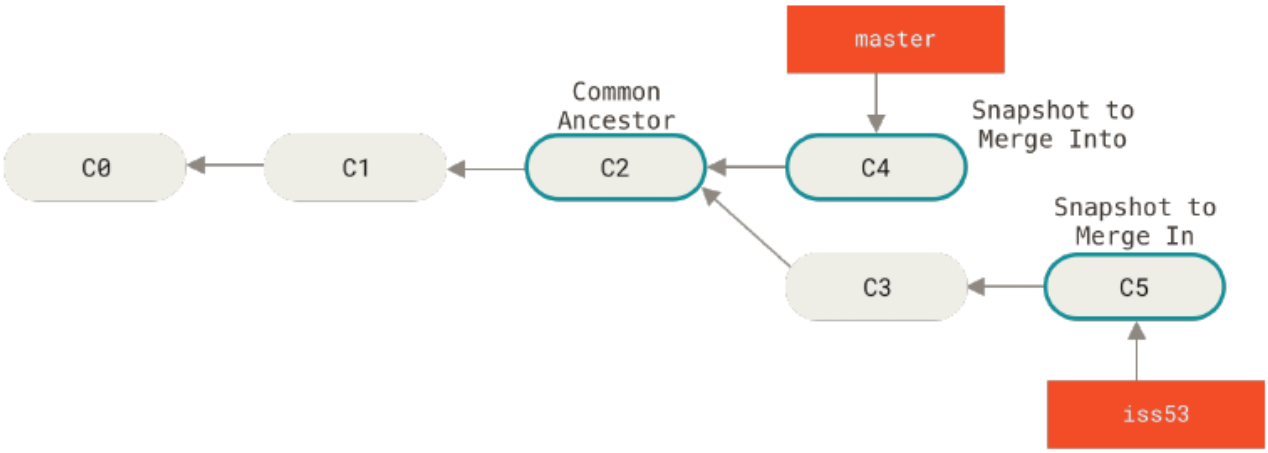


Figure 24. Three snapshots used in a typical merge

The two branches can't be merged directly with one another, because one is not the direct ancestor of the other. What Git will do is find a common ancestor, then add all of the code committed in those two diverging branches. This is called a three-way merge, and results in the following diagram.

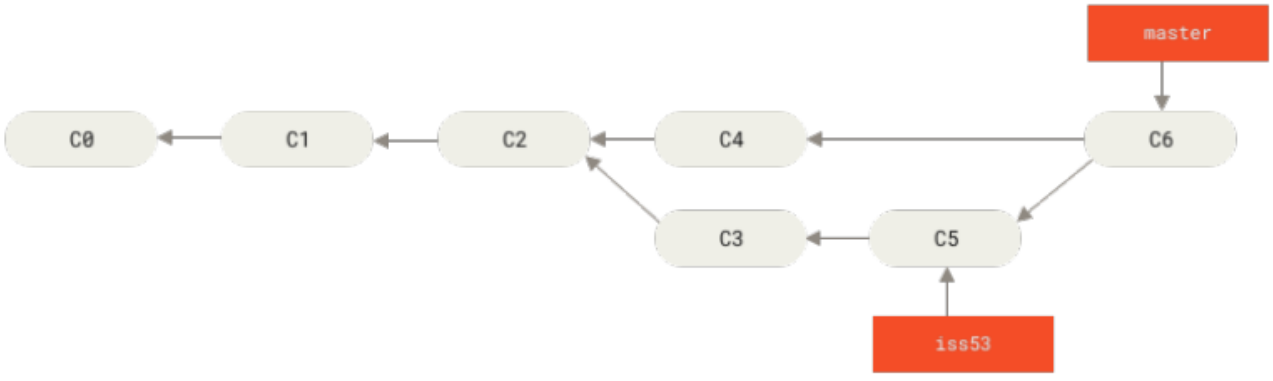


Figure 25. A merge commit

Merge Conflicts

The elephant in the room is about resolving the inevitable **merge conflict**. Merge conflicts result when you try to merge branches that have different versions of the same file. Git becomes confused about which file should go into the branch. You have to manually open this new abomination of a file and resolve its internal conflicts. Git will automatically format the file so that the separate versions are plainly marked, and you can delete either, or delete them both and replace the contents from scratch. This is not always a preferred solution, so there are more tools for resolving conflicts.

Remarks

Git is a powerful but intricate tool, and it takes experience to understand it. Luckily, you have Git installed locally on your machine now, which enables you to track your own local projects, even if you aren't committing to a remote server. for more information on Git, I highly recommend the ProGit manual.

WINDOW MANAGERS

This is literally just a section on i3. There's another window manager called Sway, but honestly it's so similar that you could probably just swap between the two if you wanted. i3 is a *tiling window manager*, named so because it tiles windows to be adjacent to one another. Your typical desktop environment also contains a window manager, but the windows can be clicked and dragged. A tiling window manager is much more efficient when it comes to managing screen real estate, but it's lacking in the user interface department.

The main feature of a tiling window manager is that it lets you navigate your desktop environment *without* a mouse, which is extremely valuable for people working from the command line. It's worth bringing up the idea of a development ecosystem, where the tools you use integrate well with each other not just mechanically, but also ideologically. The command line interface only supports characters typed by the keyboard. There are tools out there that abide by this same principle, and it may seem restrictive at first, but it can greatly improve your workflow in the long run.

TYPESETTING

MS Word and Google Sheets are great for quick notes and collaborative document editing, but they have a bad reputation as effective software for integrating pictures and images. If you need to insert an image, chances are you're not going to have a good time. There are multiple options for image size and orientation, word wrapping, image interactions, and it gets pretty frustrating that the entire document layout seems to change whenever an image is moved slightly. This is especially frustrating if you've got a long document. Occasionally pages will spill over with a ton of white space at the bottom and you may spend an absurd amount of time trying to work around that "feature". This is the pitfall of WYSIWYG editors, or *What You See Is What You Get*. Their entire design is based on the user's need to have the document render in real time, rather than compiling the contents. Typesetting is a complete and total refusal of this technique. Typesetting languages literally look more like HTML than an actual document, but the conveniences it affords allows you to declare things once and never worry about them again.

Markdown

I made this entire PDF using a Markdown editor (Typora). Markdown is a typesetting language that is easy to pick up and useful to have, especially when you want nice auto-formatting. The Markdown interpreter converts text into HTML, which your editor can then render with preloaded CSS. There are plenty of stylesheets available to you if you want a different theme, and you can even make your own.

$\LaTeX$

$\LaTeX$ is a typesetting environment based off of TeX. We will not be installing a TeX distribution, because the setup can take quite a bit. Instead, we will be using an extremely popular web-based client called Overleaf. Overleaf is especially useful because it allows real-time collaboration between users, similar to Google Docs. As an example, we'll be using a custom CV template class to create a document. On Overleaf, create a blank project and import the workshop file `something`. Follow along with the guide to get a feel for what $\LaTeX$ is useful for.

Remarks

Although typesetting systems are my favorite topic of this talk, they're also the shortest. I can't possibly cover every single use case, and there's already extremely good documentation on Overleaf's website. I highly recommend using the $\LaTeX$ typesetting system for your next written report — it can make academic formatting so easy, and I wish I had know about it sooner.

RASPBERRY PI

This last section is meant to be a bridge between the gap of computer software and hardware. I've talked a lot about programming tools and interfaces, but not at all about the systems which these tools are built on. The Raspberry Pi offers an affordable solution to use an operating system designed by and for programmers - GNU/Linux.

Linux for ARM systems

Linux and MacOS are both Unix-based, which is why they share the same file structure. In terms of development, they're pretty different. MacOS is developed and maintained by a dedicated company that releases new versions incrementally. Linux has multiple different distributions, each maintained by different companies. Popular distributions include Debian, from which Ubuntu is based off of, RedHat Linux, which includes Fedora, and CentOS, Mint, Manjaro, and of course Arch. These distributions all have slight variations, but functionally, they're capable of the same tasks. A key difference between them is the package managers they use. I specifically requested that Windows users used Debian/Ubuntu WSL because those distribution use the *aptitude* package manager, which means that the commands are identical to mine, and we're downloading the same software version.

Some distributions are **rolling**, which means they are updated frequently, while others get updated once every year or so. Ubuntu is one such distribution that provides long term support for its users, which is part of the reason why it's the most popular Linux distribution among users.

There are dedicated teams for rolling out each of these Linux distributions to the Raspberry Pi. The Pi Foundation has Raspberry Pi OS, while the major distributions generally have support for the Raspberry Pi. This is what makes the Raspberry Pi powerful. It can run Linux. Everything I have shown you here runs extremely well on Linux. It is the ideal programming environment, but it's not accessible because you have to learn how to interact with the command line first. After you bridge that gap, however, you can do several things:

1. Use your Raspberry Pi as a platform to test new applications. I tested i3 on my Raspberry Pi for 3 weeks before installing it on my desktop. I plan to do something similar for mutt, the command line email client.
2. Route internet traffic through it, possibly for a PiHole or even a server. Many have successfully installed Ubuntu Server on a Raspberry Pi with good results. There are several other projects that people have

done, all available through the Pi Foundation's website.
3. Experiment with different distributions of Linux. This is a big one, because it's costly to install distributions on your primary computer. A major deterrent to installing Linux is the fear that it will wipe your hard drive. If you have a Raspberry Pi, you don't need to worry about that, because you can always reflash the SD card if it gets corrupted.

A Raspberry Pi with 4 GB is around $50. You can plug in a keyboard (and a mouse, I guess), hook it up to a monitor, and be up and running with Linux. You have access to major programming languages such as C and Python. You can program in those languages through the command line. You can have access to all the code you develop through version control. It is the ideal development platform, especially for people new to the command line.

If you have a Mac, none of that applies to you. You should definitely keep using your own computer, unless you have certain electronics projects that you are interested in pursuing. In which case, feel free to consult with IEEE Makerspace! We have a team of experienced students that can advise you in terms of hardware requirements for any electronics project.