# **Git - Notes Alert! =>** The Git ‘master’ branch is now known as the ‘**main**’ branch. Use ‘main’ not master in subsequent Git commands **Last Update: 16 April 2023**

My Git notes file location: Software > SoftwareDevelopment > Git/ **/////// Bash uses forward slashes ///////** not back slashes \\\\\\

Using spaces within Bash commands: Use a backslash to represent a space. e.g. cd Hello\ World changes the director to [ Hello World ]

*Launch:* ***Git Bash*** *from within Window’s* ***Start***

**Configure Git:** First configuration actions to perform after a Git installation on your PC Using **--global** options means all future project will be connected to this identity

**Git Commands:**

**git --version** After installation, verify, the Git response will show a version number: *git version x.xx.x*

**git config - -global - -edit** Opens an editor showing parameters <or>| git –global --edit

**git config --global user.name <***“John Doe”>*Set your username and e-mail:

**git config --global *user.email*** [*emailAddress@gmail.*com](mailto:emailAddress@gmail.com) Set your e-mail address:

**git config --list** Verifies config settings displaying all significant aspects

**git config --global core.editor** <**path**> Establish default text editor-Use full path

< **"'C:\Program Files\Notepad++\notepad++.exe' -multiInst -nosession -noPlugin"** >

**git config --global init.defaultBranch “< . . . >”** **Sets or change the default branch** to < . . . > ~ “main” if placed in <option>

**git config** Get and set configuration options < **-- list** > displays configurations

**git config** <*option*> Show a specific option by typing: Such as: *git config user.name* or *git config user.email*

**git config –unset** <setting> Remove a setting from configuration files: i.e. < git config –unset –global user.email >

**git config –global alias.**<name> Config Alias, e.g. git config - -global alias.show-graph \ ‘log - -graph - -abbrev-commit –pretty=oneline’ == showgraph

**git cat-file -p** <commit\_hash> Display the parent commit from the current commit

**git init** <*optional directory >* Create local repo within current or optional dir. Delete .git to revert to normal non-repo folder

**git init -b** <main>Create local repo within current folder. Designate the default branch name to ‘main’

**git clone** < *repo URL path*> Clones entire existing remote repository to repository on local PC, Checks out an initial branch

**git add** <*filename, … >*Stages a new file or modified contents files previously created or modified.

**git add .** Adds all new modified files to the staging area or optionally use **git add** **- -all**

**git add -f** Adds (forces) ignorge files to staged status readying for next commit

**git commit** Commits to local repository: ‘ .git repo ’ after launch default txt editor to insert commit message

**git commit –v** This commit version displays additional detail of the commit action

**git commit –m** <*“msg here “>*Commits (added) files in staging area – msg describes purpose of commit – Snapshot is executed.

**git commit –a –m** <*“msg here “>* Commits all file changes from stage area and commits simultaneously in single step (skips ‘add’)

**git commit - - amend** Retracts most recent commit before pushing – Original msg displayed to correct typo if necessary

**git status** Provides status of present repository: View the status of each file in a repository.

**git log** Displays commit history options: < -(n), - -oneline, - -decorate, - -stat, - -after > n limits max no.

**git log - -pretty=oneline** Displays short list of commits with one line per commit.

**git log –all** Displays all the commits, not just the ones back to the parent branches.

**git log - -graph**  Presents graphical representation of the log

**git checkout <***branchName***>** Moves ‘HEAD’ to point to the branch <*branchName*>

**git checkout - -**<*filename*> Discards changes in working directory

**git branch** Displays branches, Add <branchName> arg to create a new branch **git branch** <branchName>

**git branch -d**  <*branchName*> Deletes branch from current directory ?

**git switch** <branchName > Switch to the branch <branchName >

**git merge** <branchName > Merges source branch <branchName > into current branch (where Head points)

**git rm - -cached** < filename > Removes file from committed status … Caution ‘ git rm < filename > ‘ permanently deletes file

**git rm - -cached** <*filename>* Removes file from version control but preserves file locally.

**git diff** < *filename* > Displays difference between current directory file and ‘filename’ in other directory

**git diff - -staged** Displays changes but not yet committed.

**.gitignore** Populate ‘.gitignore’ w/ contents of files not to be committed. Git ignores these files during remote push .

**git.ignore** <*filename*>The git.ignore <*filename*> instructs git to ignore certain files when committing.

**git #** Text following the # character indicates a comment

**Git Reference Commands:**

**git --global** Opt applies attribute to subsequent repos. (git config –global user.name) Omit - -global for specific dir.

**git cat** <*filename>* Displays contents of the <*filename>*

**git cat ~/.gitconfig V**erifies config by displaying user name and email address

**git - -help** Common (short list) Git commands. **- -help** < topic> for particular topic. Use **help -a** (long list)

**git help** < -a, -g, - > List commands and start with the Main Porcelain Commands (Use the ‘**q**’ command to **quit** help)

**git - -help** < concept > Use git –help < concept >, git help –a, git help –g (i.e *git help config*)

**git rm** <*filename>*Delete file from wrkg dir and stages the file deletion. (options -r, -f) (May use explorer as well)

**git rm -rf** <directory> Delete an entire directory (and all its contents)

**git checkout - -** <*filename>*Undelete a file! Follow this up by doing a git status and ls -la to confirm file is back

**git mv** *oldName newName* Change filename and prepares (staged) for commit (staged)

**git clear** Clear out commands/actions history from window - Scroll up to see preserved command history.

**git clean**  Removes untracked files from the working current directory.

**git commit –m “***first commit***”** Commiting the local directories & files*Before committing is always advisable to do a ‘diff’ before the commit*

**git.ignore** <*filename*>The git.ignore <*filename*> instructs git to ignore certain files when committing.

**git remote add origin <** [*https://github.com/githubUserName/githubRepositoryName.git*](https://github.com/githubUserName/githubRepositoryName.git) *>* ( URL of remote repository)

**git remote show origin** Provides paths for fetching and pushing to remote origin <GitHub> (my choice of remote)

**git remote -v** Display remote repository location/directory: My GitHub: origin <https://github.com/BestTry/Cpp.git>

**git clone** <*remote url addr*> Paste the remote URL address to duplicate remote repository on local PC

**git push –u origin main** Pushes committed files to remote repository (i.e. GitHub) on repository ‘origin’ / main branch

**git push –u origin main** Flag < u > creates tracking ref & links with remote branch which allows pulls without arguments

**git status** Provides status of file, that is whether it is untracked, being modified, staged or committed.

**git fetch** Retrieves latest meta-data from origin without doing any file transfer (Safer – do before pull)

**git diff** View what has changed but not yet staged. Compare working directory content with staged area

**git diff - -staged** View difference staged area and committed area content (last committed content)

**git diff - -cached**  View difference ( See previous line for description ) - - staged and - - cached are synonyms

**git diff … origin** Follow fetch and do before pull

**git stash** Temporarily shelves files content from working copy

**git stash pop** Applies previously stashed content to the working copy and removes content from stashed copy

**git stash apply** Applies stashed content to both working copy and keeps content in the stashed copy

**git rev-parse** <1st 9 hash dgts> Displays full 40 hexadecimal hash number given the first 9 digits of hash #

**git ls-file -s** Displays the git index hash #, the file hash # and , the file name

<https://git-scm.com> Git website URL

[https://git-scm.com/docs](https://git-scm.com/docs/gitignore) Documentation: https://git-scm.com/docs

Command Line Directives: (Some of these common useful directives/commands have an < Object / Target>)

The ‘High Level Commands’ (Porcelain Commands) section (a few here) are the most commonly used Git commands

**Console / Terminal - Command-Line Syntax (Using Bash):** Commands are ‘Terminal’ not git commands! And not CMD Console

**&&** Directives may be combined in a sequence with && operator e.g. $ mkdir myDir **&&** cd myDir

**mkdir** i:/Code/GitDir Make new directory (folder)

**cd** i:/Code/GitDir Change directory: **cd ..** Move up dir;

**cd ..** || **cd ../../../** Backing up one or more (3) directories.

**cd ~** Moves head focus to beginning root folder/directory

**dir** || **ls**  ‘ls’ - Lists of files in directory & subdirectories (Use ‘**\’** to insert a space e.g. <dir\ Name> with space

**ls –la** < -a > List files including hidden **.git** in directory: stacked vertical < ‘-la’ > or horizontal < ‘-a’ >

**pwd**  Displays current directory including its path

**nano** <filename> Launches text editor …. Use **^x** to exit the editor - (Text editor ‘nano’ is the Git default text editor)

**cp** <filename> <newfilename> Copies existing file to a new file with new name

**cp -r** Copy an entire directory to another directory (e.g. *cp – r dirName1 dirName2*)

**touch**  Creates new empty file within the present working directory ‘pwd’

**mkdir** < … > Create a new empty folder directory or (mkdir <project.gi>.

**rm** <*filename>* Removes file from directory - (May use technique in file explorer as well)

**rm -r** Removes **directory** ( *rm -r dirName*) Caution! hit’g -f instead of -r could remove git all files in directory!

**mv**  Move / Rename a file in a directory (i.e. *mv oldName newname*)

**mv** <srcDir> / <desDir> **.** Moves src directory/folder to present directory. Must include ‘ . ‘ which designates *current* dir/folder

**clear**  Clears screen of text out of view (in cmd console, use **cls**) Nothing lost: Scroll up to view previous history.

**find .git/objects** Displays blob git objects (once tracking) within repo and 40 hexadecimal (160 bits, 20 bytes) hash number

**ca**t <*filename>* Display the contents of the file ‘fileName’

**^c** Symbol ‘^’ == keybrd **Ctrl** - Cancels current cmd prompt ‘>’ - Returns display to Bash cmd with prompt ‘**$**’

**esc** Once within the text editor Vim hitting the esc key places control back to Vim command mode

**q** Use ‘q’ command to return to the Bash command prompt ‘**$**’ - options: < q, ctl + q, shft + q, ctrl q + c >

**:q** Quits Vim from the Vim command mode

**:w** Saves the contents of the text file

**:wq** Saves and quits Vim in one single step

**↓↑** keys Scolls through previous and next commands

>>>>> The following syntax has not been verified … yet.

vi readme.md Creates a new text file using VIM - Use ‘I’ to insert text

cat foo.txt Creates a new text file using CAT use *‘ctrl+d’* to save file

echo >> foo.txt Creates a new text file using power shell command

SSH: Preparing SSH key before **Connecting Local and Remote Repository**

ls –la ~/.ssh Check if a SSH key is already present

Generating a SSH key if one is not present:

ssh-keygen -t rsa -b 4096 -c [lordbless77@hotmail.com](mailto:lordbless77@hotmail.com)

Parameters will be requested or accept defaults.

The key fingerprint will be generated and randomart.

eval $(ssh-agent -s) Add SSH key to the SSH agent

ssh-add ~/.ssh/id\_rsa Add SSH key to where the key was created.

Go to directory where the SSH key was created. (.ssh folder)

Open the ‘id\_rsa.psb’ file and copy the contents (ssh key) of the file

Go to GitHub and under profile > settings > SSH and GPG keys > New SSH Keys

Give it a tile “Home PC”, paste key and supply GitHub password.

Git Ssh -T [git@github.com](mailto:git@github.com) Check communication between PC and Github

At warning message type ‘yes’ to continue connecting.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Git / GitHub -** Distributed Version Control System (DVCS) - Local repository software and the remote repository

Git uses Local Version Control: Version Database – Version is controlled with version patches on your local computer:

To see the present version of the file: Git adds up all the patches to recreate the file.

Introductory Note: When using Git commands, use ‘Bash’ (or PowerShell) instead of the less capable Windows Command Console. Then begin by going to the main local repository folder on the PC. On my home computer this repo is located here:

< C:\Users\lordb\Desktop\Collections\GitRepo >. Within this repository “GitRepo” folder, I have several repositories such as: “Arduino”, “Cpp”, “CSharp”, “Git”, “Sandbox”, and “SimpleWebsite”.

For loss protection, the local database was expanded to other computers (distributed). Files are also store on a remote centralized server for sharing and protection. This is called a ‘Centralized VCS’ – Version Control System.

GIt does incorporate a Distributed Server used for version database sharing among other applicable project computers.

GitHub, Enterprise, Bitbucket, and Gitlab are such Distributed Server Hosts.

Git GUI clients: [*https://www.thewindowsclub.com/git-gui-clients-for-windows*](https://www.thewindowsclub.com/git-gui-clients-for-windows) - such as “Sourcetree”, “GitHub Desktop”, “Git Cola”

Git uses snapshots, not differences to manage changes. It takes a snapshot of the entire whole project not just changes to a particular file. It may not store a file that did not change but it creates a link to the file of its last change.

Git is Local: Git functions locally on your computer even though it is a Distributed Version Control System. If the server connection is not available (presently in the staged state), wait until it is available, and then execute a commit command to the remote server. Git has Integrity: Everything in Git is check-summed before it is stored in the database consisting of 40 long hexadecimal character check-sum. The **check-sum** is done by using a **SHA1 hashing algorithm**. Same data even from different sources will produce the same 40 character hash. Git stores everything (not by its filename but) by the hash value of the data contents. It is possible that two different files may produce the same hash check-sum. This event is called a collision.

**Three Stages of a file:** < Committed / Modified / Staged > that are being tracked by Git from the last committed snapshot. Files may first be generated as untracked ( The file has not yet been ‘added’ to the staging area ).

**Committed:** File is safely stored on local computer (local data base) for the target project. Occurred at “Commit Snapshot” event.

**Modified:** Changes to the file causes the file to move to a Modified status. (File has been Altered) (Work in Progress).

**Staged:** File changes are intended to become committed at next commit command which incorporates the file changes.

**Untracked:** Files added to the project after the last commit are regarded as untrackedfiles. Untracked files are not under version control. To keep a project small and efficient, you should only track source files and omit anything that can be generated from those files. The latter content (generated) is part of the build process – this generated content should not need revision control.

Three Stages of Tracked Files:

“Committed” (Unmodified files from the last commit snapshot)

“Modified” (Changes made to files since the last commit)

“Staged” (Changes marked to be added into the next commit snapshot)

Steps of Version Control: 1.) - Unstaged Work 2.) - Staged Work 3.) - Commit Work 4.) - Push Work

**add:** In order to move a new file from the untracked status to the tracked status utilize the **add** command. The command moves the new file to a staged status. The file will then be added to the project in the next Commit “Snapshot” and committed. Tracked or untracked files that are finished being modified have to be added to the stage area. Stage a file using the **add** command. Any new or modified tracked file must invoke the add <*filename*> command readying it for next commit.

Visual Studio: - From Visual Studio menu bar select ‘Git’ -> Settings to enter name and e-mail

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Foundations of Git

Git Three States: *Committed* 🡸 *Staged* 🡸 *Modified* 🡸 New created files {Promotion based system}

Locally Stored 🡸 Pre-commit 🡸 Working 🡸 Untracked files {Local states of a file}

Git Three Areas: Local repo dir 🡸 Staging 🡸 Working {File Locations}

.git repo Area Directory

**↓ Push ↑ Pull**

Remote Origin/Master repo dir ====================🡺 Remote Origin/Master repo dir (GitHub/Server) (GitHub/Server)

Timeline

Description automatically generated

GitHub:Hosting social code repository service for GIT owned by Microsoft. Holds code that can be pulled from its repository with ‘Pull’ requests.

Searching GitHub: Global Search, Scope of search, Repositories, code, issues, …… Advanced search page,

Search syntax examples: dotnet

**Three States of a Git Project:** ( Locations of a project or file(s) ) **Working Directory** / **Staging Area / Repository**

Working Directory / Staging Area (Index) / (Repository (.git Directory)) ( Repository locations -> local & remote).

Start with the **Repository** (.git Directory) – Pull down the origin project data from the remote Repository Server.

Repository (with its latest revision) should now be stored on the local computer. This mega data, the project’s object-based data is transferred to the (Local Computer) into the Working Directory.

**Working Directory** (Checked out version from the .git Directory – Repository - project origin).

**Staging Area** (Index) Sits between the Repository and the Working Directory. Use the Staging Area to build of a change or set of changes to commit later. When the commit is fired off, only the changes in the Staging Area is committed not the rest of the unmodified files in the Working Directory. The staged files are pushed (committed) to the Repository for future use.

Getting Git Up and Running - Installation on Windows: (**Launch Bash from Windows**)

Use the Command Line Interface ‘CLI’ tool: “Command Prompt” or “PowerShell” within Windows or “Terminal” in Linux.

Within Windows use (**Git Bash**) as its Terminal to enter commands. ‘Bash’ emulates the command line tool of Linux (Make sure you are in your project directory – Open up Git Bash).

Graphical User Interface Tools: Git comes with GUI clients / tools such as ‘ Source Tree ‘ ‘ Git Cola’

GUI Directives within Git: ( <https://git-scm.com/downloads/guis> ) (Recommend after becoming familiar with command line interactions)

Installing Git --Windows install technique: Go to URL: <https://git-scm.com/download/win> (Chose download tool i.e. win 64 bit)

(Automatically starts the Git installer. I installs a command line tool called: “ Git Bash “ – Windows version ) ‘Terminal’ Command Line Interface - CLI

( “ Git Bash ” gives user a more Linux feel using commands within a command line window ) (When I installed Git, I selected Git ‘Bash’ for the console window).

Set up username and e-mail address with the commands: **git config** **- -global user.name** “1stName 2ndName’ **git config - -global user.email** “emailAddress@hodtsite.com”

Visual Studio Usage: (Recommend investigating Visual Studio usage later). Using Git within Visual Studio:

From Visual Studio > Tools > Options > Source Control > Git > New Project: (Gives one local Git repositories) - Select project template.config

**Repository**

Initializing a Local PC Git Repository:

Each project in Git is referenced as a unique repository. The repository is the virtual storage of a project. Initializing the project repository with the ‘**git init**’ command within a folder designated to be a local repository establishes the folder as a local repository.

To set up a repository, go to the folder where the project resides using the ‘Bash’ command ‘cd’. You might want to begin with the ‘Bash’ command ‘**cd ~** ‘ to get to the ‘Local Repository root folder. I place my repository folder off my desktop ( Desktop > GitRepro Then use the command (e.g. (e.g. git *cd /pathFolder1/…/my\_project\_folder*). To get to the GitRepro local project folder. Once in the project folder that will be used as an unique repository, use the command: **git init.** This initializes the repository creating a hidden **\*** ‘.git’ directory. By initializing Git, a ‘main branch’ is created locally. Git responds to the init command with a message that the repository (in this project folder) is initialized. This git command added a hidden < .git > subdirectory to the project folder containing all the configuration meta data to the repository. With the repository holding the ‘ .git ‘ subfolder, The folder is now a Git version controlled repository. The ‘.git’ subdirectory is hidden by default. Contents of the ‘.git’ subdirectory can be shown by typing the command**: cd .git** followed by **ls** on the next line**.** To return to the main repository directory type the command **cd ..** Follow this by typing **ls** once again and then this will display all the files presently within the project’s (.git) repository folder. This folder is now under version control by Git. (I initialized folder *GitRepo/C++* on my PC).

* To view hidden directories/files: Within ‘File Explorer”, double click on the ‘**View**’ tab. Select the ‘Options Icon’, and select ‘Change folder and search options’ from the drop-down menu. Within the popup dialog, select the ‘**View**’ tab. Inspect the ‘Advancd settings’ checkboxes and enable the radio button: “Show hidden files, folders, and drives”

Repository Local PC Reset/Delete:

To remove the project from being version controlled, just delete the ‘ .git ‘ subfolder. All version control is now disabled form any files in this folder and is no longer a repository. The ‘ .git ‘ subfolder / directory is the only difference between a Git repository and an ordinary folder. Deleting the ‘.git’ subfolder will turn the project back into an un-versioned collection of untracked files.

Staging : Stage a Snapshot (Within one local repository / folder / directory)

**git add <*filename>*** Add a file to Git to begin tracking it ( File will become staged ) (or a modified file).

**git add .**..or.. **git add -a** Adds all the files presently in the working directory of repository to the staged area

**git status** View / Check the repository project’s status repository

**git rm –cached <***filename>* Unstaging a previously staged file, To unstage a file use this ….. Or commands below:

Unstaging:

**git restore --staged** <*filename,* ...> to unstage a file(s)

**git restore --staged .** Unstages all files that were just staged using the ‘add .’ command ( the ‘.’ is required)

Committing:

Committed snapshots are considered “safe” versions for a project. The commit command opens a text editor and prompts you to enter a message describing the commit if you optionally don’t include the ‘ **-m** ‘ option.

**git commit** or **git commit –m** *“Add commit info/comment message here”*Type info / purpose for the file when prompted

The “–m” option lets the developer specify a commit message on the command line instead of opening a text editor.

Type the message (line 1) leave the remaining text, save the file and exit the editor.

**git commit –m “**<msg>**”** “Msg”describing purpose of committing” **(**The dash m < **-m** > is for including the descriptive message)

Ignore Select Files from Commits:

The git ignore file **‘.gitignore**’ within the repository directs Git to ignore and not track files within the repository. Do this by creating the **‘.gitinore**’ file from the text editor and placing the file names or patterns withing this ‘.gitignore’ file. Below are common hints and hacks that are used:

* Asterisk implements a pattern where zero or more characters are masked: **\*.ext** Such as \*.log cause all log files to be ignored.
* Patterns starting with a forward slash masks files only in the same directory of where the ‘.gitignore’ resides.
* Patterns ending with a forward slash mask will only mask the folder names.
* Patterns starting with ( ! ) char negates common pattern: Example: The file ‘ !TODO.txt ‘ will be committed if ‘ \*.txt ‘ are being ignored.
* Double asterisks direct git to ignore everything in subdirectories: ‘documents/\*\*/\*.html ‘ will ignore html files in the documents folder.
* The ( **#** ) character at the beginning of the line within the ‘.ignore’ signifies a **comment** as is ignored by Git for direction.
* Common files that are designated to be ignored: log files \*.log, bin files \*.bin etc.
* For additional information see: <http://github.com/github/gitignore>

**.gitignore** Populate ‘.gitignore’ w/ contents of files not to be committed. Git ignores these files during remote push .

**git.ignore** <*filename*>The git.ignore <*filename*> instructs git to ignore certain files when committing.

Creating a New Remote Repository

Git web-based Repository Hosting Service plus collaborator service allowing merging information with other developers.

Create a **GitHub account** or sign in: go to*:* [***www.github.com***](http://www.github.com)and sign up with your credentials. You will select a username, a password and enter your email address. There are two options: a free public repository option and a paid private repository. When creating a new repository at GitHub, it is recommended to elect to include both a “**README.md”** file and an “.**gitignore”** file. Follow the instructions on the web site to verify your membership credentials which will include a response to an e-mail sent from GitHub.

Creating a repository within GitHub: Select/Hit the < **Create Repository** >  button. Select the options under ‘Creating a new repository’.

Creating ReadMe.md File

Creating a ReadMe.md file: After creating a **new repository** while in a local PC folder from the command line (While in “Get Bash”), add a README file: **echo “# repositoryName” >> README.md** Adding a Readme file to the repository ( untracked ) with a h1html header. The echo command with # hashtag places the repository Name within the README.md file. One can place anything desired within the README file. GitHub automatically places the repository name in the README file. All the files within the repository on GitHub are yet to be tracked. They are presently untracked.

All the files in the local repository can be added to the GitHub repository staging area by the first following command:

**git add .** Now all the files in our repository are in the staging area waiting to be committed at the next commit action. Next do a commit and then follow through with a **git push** command.

Pushing to Remote Repository such as ‘GitHub’: *https:// github.com/<gitHubOwnerName>/<gitHubRepositoryName>* GitHub:BestTry **R \_ \_ \_ \_ \_ \_ \_ \_ 1 \_ \_ \_ \_ !**.

Pushing a local repository to GitHub requires developer to select both an owner/user name and repository name. Enter the names within the GitHub hosting service. (Include information within the optional Description text box) when creating the repository.

Check the box □ “Initialize this repository with a README … OR … alternatively, provide a readme file using the ‘Bash’ command line and pushing it to GitHub. Readme files allows collaborators to know information about the project.

There are two ways in pushing the local project to GitHub: “HTTPS” and “SSH” URLs (Recommended default) and “SSH” URLs. The repository may have an accessible level of public or private. Click the appropriate radio button. Using Https “push” / “pull” should work even if the local repository is behind a firewall or proxy server”. The “SSH” is the greater secure method of pushing and pulling and requires keys, passwords etc. Or If the local repository is already linked to that unique folder in GitHub: git push will update.

Linking to Remote Repository plus Cloning from it

Before launching “Bash, or “Command Terminal Console”, copy the URL within the text box of the remote repository e.g. [*https://github.com/userName/repositoryName.*git](https://github.com/userName/repositoryName.git) The following command links/updates the remote origin Github repository to the local repository by creating a link that allows push and pull transfers between the two repositories. The local repository will now point to the remote Github repository. Connect the local repo with the Github repository linking the two repositories with a Git command:

**git remote add origin** [***https://github.com/githubUserName/repositoryName.git***](https://github.com/githubUserName/repositoryName.git)Example: < git remote add origin <https://github.com/BestTry/Cpp.git> >

Use the command **git remote show origin** to verify the remote repo is link to the local repository on the local PC.

Linking to Remote Repository: - Alternative Method with **Cloning** the remote contents After creating the remote repository at GitHub (See above – “Creating a New Repository”), Select the green button on upper right-hand side marked “Code”. Click the code button and select https. Click on the clipboard ‘Icon’ and copy the URL address. Switch back to the local repo on your PC. Within the repo on the PC, enter the command **git clone** <*paste the URL address just copied*> i.e. *(*[*https://github.com/userName/repoName.*git](https://github.com/userName/repoName.git)*)*

Pushing / Committing Repository / Files: Commands to commit and then pushed files to the remote repository.

Pushing an **existing repository** to the remote repository from the command line: First, establish a link between master branches:

The command below ties the local repo to the remote repository by creating a link between the two repositories (from within the local repository). This command creates a link which allows files to be pushed or pulled between the two repositories.

**git remote add origin** [***https://github.com/githubUserName/repositoryName.git***](https://github.com/githubUserName/repositoryName.git)(URL of the remote repository)

**git push –u origin master** GitHub will prompt the user to enter the user ID and password. Then GitHub responds with a message:

“ Branch ‘master’ set up to track remote branch ‘master’ from ‘origin’ “

$**git remote add origin**[*https://github.com/BestTry/Cpp.git*](https://github.com/BestTry/Cpp.git)$ **git remote add origin** [*https://github.com/BestTry/Sandbox.git*](https://github.com/BestTry/Sandbox.git)( Directive I personally took to **establish the link** to GitHub )

Repository Creation & Push Example: (In a single step from the local folder intended to be a local repository and remote repository)

(While in the local folder enter the following commands while in the ‘Get Bash’ command window)

**echo “#** *repoName***”** >> *ReadMe.md* (Once ReadMe.md is created, bring up in an editor (NotePad++) and populate it with pertinent relative info)

**git init** Create a new repository on the command line when in the current target Repo folder

**git add README.md**

**git commit –m “***first commit***”** *Before committing is always advisable to do a ‘diff’ before the commit*

**git remote add origin** [*https://github.com/githubUserName/githubRepositoryName.git*](https://github.com/githubUserName/githubRepositoryName.git)

**git remote show origin** (Provides paths for fetching and pushing to remote origin ‘GitHub’)

**git push –u origin master**

**git status**

Remote origin is on GitHub. After pushing, the two repositories local and remote origin on GitHub should be the same.

Check the status of the project by using the check status command:

**git status** Git responds with status message:

On branch master (This is the branch you are on ‘local’)

*Your branch is up to date with ‘origin/master’ ( ‘origin/master ‘ is at the remote GitHub branch‘)*

*nothing to commit, working tree clean “*

Adding a ReadMe file to the repository: First copy the “HTTPS URL at the top of the page on the Github website.

Jumb over to the linux command line window (Terminal or Bash ?) and add a read me file:

**echo *“# repositoryName”* >> README.md** The echo command says that we want to write the “repositoryName” with a H1HTML header into a new README file. GitHub automatically adds the repository name within the README file but you can put anything you want into it. The README file is currently in an untracked status. Therefore, we need to add the README file and our other repository files to the staged area so that they can be tracked by Git.

Updating Remote Origin Repository:

Push the local repository committed files now pointing to the Github repository):

**git push –u origin main** Pushes the local repository files to the remote GitHub repository ( ‘origin main’ ).

GitHub then prompts for a username and password. Git responds displaying a message of what just happened:

\* [new Branch] main -> main

Branch ‘main’ set up to track remote branch ‘main from ‘origin’

Note: ‘origin’ main refers to the remote branch ‘main’ that sits within the remote repository profile such as the GitHub host, while the local branch ‘main’ refers to the local repository branch main. Both the “origin main” (remote) and the “Local Repository” each have a “Main Branch” and when they are up to date with each other, they have the same checksum.

When one initializes Git, it creates a branch called “Main”. When you make changes, changes are on a specific branch.

Summary of Repository Creation:

**git remote add origin** [*https://github.com/githubUserName//repositoryName.git*](https://github.com/githubUserName//repositoryName.git)

**git push –u origin main** <Or> **git push -u origin main** (Appears GitHub has substituted “main” for previous master)

Deleting a Remote Repository on GitHub **Warning! Red Zone …………… Caution! ……………**

1. Login to your account. 5) Scroll down to the "Danger Zone" section block.
2. Click on **Repositories**. 6) Click on "**Delete** this **repository**" button.
3. Select your **Repository** (that you want to **delete**) 7) Type the **repository** name (that you want to **delete**)
4. Click on settings (gear icon) Setting tab. 8) Now click "I understand the consequences, **delete** this **repository**" button. (Enter Pass Word)

Git Basic Commands Continued:

**git status** Getting the status of the project by entering: Git responds with up to date branch information.

**git status –s** Two version of same command:Git returns with the short status; coded with < ‘M’ / ‘A’ > as shown:

**git status –short** ‘M’ = Modified; ‘A’ = New file added to staging area; ‘??’ = New file untracked by Git

**touch** *filename*  Adds a new file to the project: Use the git command ‘**touch**’ Example syntax is shown

This filename file is presently in an untracked state. Track by using the add filename command.

**git status** Executing ‘**git status’ once more**, causes git to respond with a messaged indicating that filename is ‘Untracked’ and Git suggests adding the file to the tracked staging area.

**git add** *filename* Adds the file to the stage area. It will be ready to be committed in the next commit snapshot.

**ls** Command shows all the files in the project repository committed and staged.

Make changes to an existing file (such as “fileNameToChg”)

Now since we made changes to the *fileNameToChg.txt*, add it to the staged area and then enter a **git status** command

**git add** *fileNameToChg.tx* **git status**

Git will respond with a message indicating the files that need to be committed.

*modified: otherFileName.txt*

*new file filename.txt*

If the otherFileName.txt is opened once again after the last change / modification, make sure you add it once again otherwise the older version will only be committed (during the next commit process) and not the latest modified version.

Git ‘diff’ Command: What changes are staged and ready to be committed?

-deleted text appears in color red +added text appears in color green

**git diff –staged**

diff - - git a/file1.txt b/file1.txt 🡨 Compared Files

Index 9863745 .. f30c839 100644 🡨 File Metadata (SHA1) First two nums are file Hash data versions, last# File ID

- - - a/file1.txt 🡨 Change Markers for file a and file b (file a asigned ‘-‘, file b asigned ‘+’ symbol)

+++ b/file1.txt Diff Chunk follows:

@@ -12, 2 +12, 3 @@ 🡨 Chunk Header ‘-‘ = file a, 2 lines modified @ line 12

‘+’= file b, modified @ line 12 , 3 lines modified  
Example Lines . . . 🡨 Chunk Actual changes shown

– Old line content 🡨 Minus and plus signs shown to indicate were the changes have occurred

+ New line content

**git diff –staged –no-renames** Use this command to avoid Git confusing similar empty files with different names and that the empty file has not been renamed.

**git diff**  What changes are in the working directory? What changes are made but not yet staged?

Ignored Files From committing: First create a ignore file: <**nano** .ignore>, <**add** .ignore>, <**commit -m** “ignored files”> Place ignore files names in this text files of files to be iginored.

Pushing Local Data to Remote Repository ‘GitHub’:

**git commit –a –m** “msg here” Directly committed – skipped staging, file is committed without first staging (optional message).

**git status** Checks that the files were committed

**git push –u origin master** Pushes the commited directory files to the remote repository (i.e. GitHub) on repository ‘origin’ / master branch. May contain many files.

Typical projects perform multiple commits (of staged files) to the local repository (commit snapshot) before pushing the committed files to the origin / master, remote repository (i.e. GitHub) (or publishing the local commits).

Git responds with message of completion and returns the check sums (Hash Tags) associated with the remote commit. Also, GitHub responds with the URL path used and shows that the push completed master -> master. Run status commands for local status and git log. Git responds providing the full Hash value of the commits that were processed with description from author date and time.

Undoing Unwanted Changes:

**git checkout** <filename> Restores changes to file ‘fileName’ residing in working directory Before staging. Or method 2 below:

**git restore** <filename> Restores changes to file ‘fileName’ residing in working directory Before staging.

**git restore - -staged** <file> Step 1 (Unstage) restores changes to file after Staging that was already in staged state. Step 2 …. follows: . . . .

**git restore** <filename> Step 2 to restore changes to a file that is already in the stage state.

Or method 2:

**git reset HEAD** <filename> Step 1 This action is equivalent to step 1 description annotated above

**git checkout** <filename> Step 2 This action is equivalent to step 2 description annotated above

**git log –oneline --decorate** Restore changes after file has been Committed, first get hash key from executing this command

**git revert** <*fileName*> **--no-edit** Restores file contents back to its orig values in the work’g dir. (No edit prevents editor launching)

**git revert HEAD** Reverts local repository back to previous, last commit

**git revert** < Forty char hash > Reverts local repository to specific commit identified by the Sha hash.

**git reset** <reference commit> **Caution!** Deletes commits time forward from ref. cmt; And deletes hist. forward from ref. cmt.

Checking Commit History:

**git status** Provides status of file, that is whether it is untracked, being modified, staged or committed.

**git log** Log shows most recent commits first, followed by earlier commit dates.

**git log -n** Limits the number of commits to be displayed. The ‘n’ specifies to limit displaying ‘n’ number of commits.

**git log - -oneline** Lists the commits but in a simplified one line format of one commit per line.

**git log - -stat** Displays detailed info about each commits. Displays the commit along with files changed for each commit.

**git log - -patch** Displays more info giving diff details of each commit. Navigate w\arrow key - type ‘**q’** to quit.

**git log - -pretty**=<opt> Directs the log format to options such as <oneline, full, fuller etc. >

**git log - -since** <” opt “ > Specifies commits for periods of time. Options: < ‘n months’>, - -until “ n months’, - -from “n months” >

Commit Message: Adding additional information then that is presented by the git log command, perform an open command and have the following URL launch, use these recommendations: Commit message guidelines (Seven Rules of a Git Message: open <http://chris.beams.io/posts/git-commit> (‘open’ cmd: not work!) Recommend that the message (Subject and Body) be composed in a text editor beforehand.

Message Guidelines/Rules: 🡪 See my Word file within my GitInfo folder

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain *what* and *why* vs. how

Remove / Move Untrack Files:

Removing a file completely from Git control:

**ls**  Confirm the file that is to be removed is presently in the project.

**git rm** <*fileToRemove>* Enter the file name of the file to be remove from Git control.

Git responds with < rm ‘fileToRemove’ >

This is confirmation that Git has staged the file and has stopped tracking the file.

Now confirm the status of the project by using the status command.

**git status** Git responds with a message that the file is to be unstaged and is deleted.

The next time there is a commit action the file will be removed from the project and will not be listed.

**ls**  Confirming the file that was removed from the project, “fileToRemove” is no longer in the list.

Untrack a file in the project but do not remove the file from the project.

To keep the file around use command but untracked, use this command:

**git rm –cached** <*fileToUntrack>*

**git status** Confirm that the file has been moved to the untracked status but is still within the project.

(use “git add <*file*>…” to include in what will be committed)

**ls** Doing a list will show that the untracked file “file to untrack” is still present within the list but untracked.

Renaming a file: Rename a file, then perform a status command to verify the renaming action has occurred:

Use the ‘ls’ command to further confirm that the file is present. Then issue the command:

**git mv** <*oldName.oln> <newName.nwn>*

**git status** Git shows file is renamed and ready to be included in the next commit snapshot. The command ‘git status” always shows which branch you are on (Which branch the branch pointer “HEAD” is pointing to).

Git Visualization Tool: Launch this GitHub tool to visualize behavior of checkouts, commits, merges etc. Bring the tool up at URL: [*https://git-school.github.io/visualizing-git/*](https://git-school.github.io/visualizing-git/)Note: App has very abbreviated cmd recognition - only helps effect of commit, branching, checking out, which demonstrates hierarchy, dependency, pointer focus.

**clear** deletes/resets the graphics info in the visualization tool and starts a fresh screen.

Branches - Wrkg with: Exercise of working with branches follow: - (Make only small changes on a branch)

Branch is defined in Git as a light weight movable pointer to a commit made in a project.

**git commit** ….. **git commit** Each commit command, moves the pointer “HEAD” pointer right along with master.

Create Branch / Checkout:

**git checkout** *targetBranch* Syntax moves pointer ‘HEAD’ to ‘targetBranch’. (Use this to switch between branches)

Pointer focuses at the new branch named “newBranch” checked out under master.

**git checkout -b** *newBranch*Create another, new branch under master branch. the -b option checks out / opens up a new branch and is checked out. A new pointer has been created pointing to it.

In the GUI, “HEAD” is pointing to the current branch ‘newBranch’ which is under master.

Any commits are done on the present pointer location of the branch – ‘newBranch’.

**git branch** Check / display the new config of branch on local repository. <**git branch -M main** > will point present branch >

**git checkout** *main*The pointer returns to the ‘main’ branch (we are looking at the main branch again)

**git commit** ….. **git commit** Pointer “HEAD” moves along the path master, not the newBranch.

**git checkout** *newBranch*Pointer “HEAD” moves back to the newBranch.

**git commit** Work continues on the ‘newBranch’ version path. Pointer ‘HEAD’ moves to right (later).

Changes on the ‘newBranch’ will not having any effect on the master ‘branch’

Branching off of master allows development along a parallel environment enabling trying out new ideas, alternate solutions or new capabilities without effect the master.

In addition, this enables great capability when working with multiple calibrators.

Comment! The ‘commit’ command takes place on the branch the pointer “HEAD” is on. If there are files in the staging area (from main) and the pointer is on a new branch, that is, a branch off of the main branch, and these files are awaiting a commit in the staging area off of the main branch, it is not necessary to add these files to the new branch for these files to be committed on the new branch. But making a commit while the pointer “HEAD” is on the new branch will commit those files in the staged area off of the main branch and include them as committed on the new branch. One should reflect this fact in the message of the commit to enhance understanding.

**git checkout -b** *nxtBranch* Syntax creates branch ‘nxtBranchl’ and checks it out setting the pointer to ‘nxtBranch’ .

**git branch** *altBranch* Syntax creates a new branch ‘altBranch’, but does not check the branch out

**git branch -d** *altBranch* Deletes branch only if it has already been pushed and merged with remote branch.

**git branch -D** *altBranch* Force branch to be deleted even if it hasn’t been pushed/merged. Now locally deleted.

**git status** If unsure which branch you are on (that is where the “HEAD” pointer is focused on), implement a ‘git status’ command and Git shows where the “HEAD” pointer is directed.

Adding file to Branch As an exercise, I checkout to the altBranch and using Vim created a new file within the

new altBranch. I then did an add *newfileName* entry in order to stage it on the altBranch.

Committing & Pushing Following the staging I then did a commit and status to check for the commit.

Now I am to do a push to the remote but make sure I push it to the alternate branch on the remote repository site. Such as **‘ git push -u origin altBranch ‘** (not master).

(Note the push was using the local ‘altBranch’ name as the source branch).

Work In Progress (WIP) Stash When there is a file that is not wanting to be used as part of the project at a present

time, but we want to still keep it around, Git allows the file to be stashed for future use.

**git add .** Command adds all or files in the master branch directory to the staging area.

**git stash** This command save / moves the working directory (and its files) into a “Work In Progress” state WIP. This cleans the working directory to an empty state.

**git stash list** Git give a list of ‘Work in Progress’ (WIP) changes with branch that has been stashed.

**git stash show** Provide high level view of what was stashed and their changes.

**git stash pop** Files are returned back into the working directory from the WIP stash.

**ls** Check which files are presently within the focused branch (‘HEAD’ pointer).

**touch** <*newfileName*>Introduces a new file called “*newfileName*” to the working directory.

**git add** <*newfileName*>Stages the latest “newfileName” that was just created and modified.

**git commit –m** <*newfile purpose*>Message describes the newfile and its purpose etc.

Merging: Before starting the merging process, assure our local repository main target branch is up to date with the remote master branch. Check this status by first positioning local ‘Head’ pointer to ‘main’:

**git checkout main** Positions ‘**Head**’ pointer to main branch on local repository.

**git branch**  Check where ‘Head’ points – Git responds with ‘ \***main** ‘ if it points to main. Then ..

**git pull origin main** Perform a ‘pull’ from the remote. Assures the local is up-to-date with remote main!

Scenario: A sub-branch was created of the master branch. The sub-branch’s code was

Modified on the sub-branch; After the code was modified, the sub-branch needs to be merged with the main branch:

**git checkout master** Positions pointer ’**HEAD**’ to the main branch.

**git branch** Confirm that pointer is focused on main: The text ‘ \***main** ‘ should be displayed.

**git commit**

Before executing the next command the pointer is at main branch. Enter ‘git branch’.

**git merge** *newBranch* Merge the files of ‘newBranch’ into the main branch. (newBranch 🡪 main).

This brings up a tool called ‘vim’; Enter a message describing the merge.

Scenario: A sub-branch was created off of the main branch; After the code in the

sub-branch was modified, the sub-branch needs to be merged into the main branch.

Notice that the pointer ‘HEAD’ is still on main. But a new merge commit was created

that is pointing to both branch paths. The ‘local’ mainbranch now has changes that was in the newBranch. (Note: Doing a merge is considered an additional commit.)

Go to the terminal window and repeat the above commands. A tool launches called ‘vim’. Type “ i ” to enter the interactive mode and then enter a merge descriptive **:wq** message. Follow this message entry by entering: “ :wq ”; The ‘w’ saves the changes

**git status** while the ‘:q’ quits ‘Vim’. See my ‘Vim’ notes.

**git push origin master** Now it is time to do a push to the master branch on the remote repository (GitHub).

**git log** Update remote main branch with the latest changes reflected on the modifyBranch

Appending a File: Undoing a commit such as forgetting to include a file during the last commit. The following commands will include the file to the last commit and completely remove the last commit and replace it with this latest containing the forgotten file.

**git add** <forgottenFile> Stages the forgotten (to be included) file to state area preparing it to be included in the previous commit.

**git commit - -append** The text editor automatically launches waiting for the latest comment. Once it closes, the appended file is included that was not included with the previous commit. The previous commit is replaced with this current commit containing the appended file.

…………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………

Caution! Take special care when implementing commands in the following section!

…………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………

Reverting a Commit: Reverting a commit will completely remove all the files from the commit. **Warning!** This command will delete the directory and delete all files as well. Recommend saving the files some safe location before starting this reverting process.

**git log** Do a **git log** <-v> and find the commit to be reverted. Copy the commit hash identifier.

**git revert** Enter the command **git revert**

**git revert** <hashID> Past the hash identifier right after **git revert** <*hash#hash#hash#hash#hash#hash#hash#hash#>*

Git Reset – Three Reset Options: Reset**: Warning!** Careful consideration is highly recommended!

This is a destructive command. Implementing this action could be very damaging. So, go slow and think before acting. This command is used for undoing previous commits or undoing the history of commits. Git allows commits to be moved from its history back in the Staging or Working area. Or this command can be used to throw changes away. As seen graphically below there are different options in implementing the command. Careful consideration is highly recommended! Action is not recommended if the commits have already been pushed to the remote for changes. Appears that the default reset ‘git reset - -mixed’ command undoes all the changes on the local master branch as well as the branches off of main. Then by doing staged ‘ add . ‘ from the main branch will group all the changes together, all changes are staged. The last step is to do a commit with message and the main will be ready for a push to the remote origin/master.

**git reset - -soft** Soft reset: Move a specifi commit back into the staging area. Regroup changes into different commits or add more changes before re-committing. **git reset HEAD** <*fileName*> … to unstage

**git reset - -mixed** Or **git reset**. This is the default setting if something else is not already specified. Moves changes(commit) back to the working directory.

**git reset - -hard**  Move all changes to the trash – deletes all changes in staging or working directory. Careful!

…………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………..

Working Directory Staging Area (Index) .git Directory (Repository) 🡨---------------------------------------------------- **git reset –soft**

recycle 🡨--------------------------------------------------------------------------------------------------------------- **git reset –mixed**

bin 🡨--------------------------------------------------------------------------------------------------------------------------------------------- **git reset –hard**

…………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………..

Exercise:Shows were the HEAD pointer is pointing: (**HEAD** **->** **master**) Also displays a history of **git log**

**log - -oneline** or git commits with the recent followed by the earliest. The ‘git log’ syntax option provides much more detail. But use ‘log - -oneline’ to retrieve abbreviated hash number when one needs to use the reset command.

**git reset –mixed** *849f2c1*Move the ‘HEAD’ pointer from were the (origin/master) was pointing to (*849f2c1*)

**git status** Run status to see what happened.

**git add .** Re-stage all files to master branch. (This will group all changes together in master branch)

**git commit -m** *“Group master & newfile” Recommited all files ie. StatusLog and new file ‘textFile’*

**git status** Shows that the remote origin/main is only one commit ahead of the local main.

Before the reset the changes were all in the local main but behind the remote

Origin/main branch by three commits. This exercise grouped all the changes together to take advantage of being only one commit behind the remote origin/main.

**git push -u origin main** Do not push right away. Only do this when confident it is correct. The remote origin/main is now up to date with the local repository main.

Using Remote Repositories:

**git clone** At the remote repository (i.e. GitHub) click on the right-hand side 'Clone or download' button. Copy the URL of the remote repository by clicking on the copy icon button. Then go to the local folder that is designed to be a repository on the local PC.

**git clone** <*remote url*> At local PC, within folder designated to be the local repository, enter this command with the url.

**ls** Check for a folder name representing the name of the repository seen at the remote repo.

**cd** Change directory to that folder and then look for the 'git' folder.

**ls** Confirm that this directory or folder is now a repository by checking for the 'git' folder.

**git remote** Shows (lists) all the repositories on the remote such as the default repo name: 'origin'.

Add and additional repository to the remote by providing a new-repo name and the . . . .

remote url to the repository.

**git add remote** <*reponame remoteUrlPathName>*

**git remote**  Check to see if the new repository has been added to the remote repository.

**git remote show origin** Remote URL path is displayed.

**git remote remove** <*ShortName*> Remove the remote repository such as <origin>. **Warning!** Careful consideration is highly recommended!

Pulling from the Remote Repository ‘GitHub’:

Launch Bash on the PC

Assure ‘HEAD’ is pointing to root directory (use cd ~ command). (Or dir of choice)

Prepare/Check the local computer repository on the receiving local PC. At the command

**pwd, dir, ls, cd ~, cd, cd ..** etc. prompt, navigate to the directory/folder repository location/structure of your choice.

This directory is where the remote repository data is to be populated when the “Pull” Git command is executed. ( On my PC my repository folder is: ‘GitRepro’)

**git init** If not already done so, initialize this folder to a repository with this command:

Check if the hidden ‘ .git ‘ folder exist by: ‘cd .git’, ‘ ls’ commands, then ‘cd ..’ , ‘pwd’

Before performing any additional steps, make sure that the Git application is configured with your user.name and user.email. If not, configure git with this information. (See instructions near the beginning of these notes to accomplish the configuration.

Cloning

Link to Remote Repository: From your Git root directory, establish Remote Repository Link in the local PC folder of choice.

**git remote add origin** **<***urlOfRemoteRepo> or* [*http://github.com/UserName//RepoName***.git**](http://github.com/UserName//RepoName.git) for the repository of Git Hub.

Cloning from Remote/origin Navigate to remote / origin master on the browser such as GitHub. Click on the ‘**Clone**’ button.

Go to the remote repository ( Sign into GitHub Go to ‘github.com’ – Sign in ).

Click ‘**Clone or Download’** button

Copy the URL path Path is required for pulling the remote repository directories/files/data to one’s local repository.

Select & navigate to local repository folder: (assuming **init** to this folder was done previously)

**git clone** <*path to remote/origin>* The repository files should have been pulled on to the local computer with the latest data reflecting their latest versions. Git displays several lines of information ending with the word “done” if it was successful. Perform some ‘git status’ and ls, dir, Bash command to assure everything looks correct. < e.g. git clone https:/github.com/ …

with ID BestTry and repository Cpp.git.

Enumerating Steps:

* + - * After git int, Initialize an empty Git alternate computer folder of choice: (i.e. c:/path/path/….)
      * Configure this folder repository on this new alternate computer with ‘name’ & ‘email’.
      * Link or connect with remote repository @ github.com (<https://github.com/> - Using BestTry ID and remote repository name <remoteRepoName.git> e.g. Cpp.git.
      * Within local repository, use the clone command: **git clone** <*http://github.com/repo*>

Clone- General Git Command: Duplicating an exact copy of the repository version residing in the remote / origin repository

Navigate to the remote repository. Click on green Clone tab/button

**cd** <*targetLocalPCfolder*>Navigate to the directory where the latest remote files are to be placed on the local PC

**git clone <***urlOfRemoteRepo>*  Also known as the < *hash tag* > Option: Clone - SSH (Using Hash Key) Within ‘Bash’ console type ‘ssh < … key gen … -o >

**git pull -r origin master** I used this syntax to pull version from the remote origin and it seemed to work

**git remote add** **<***additionalRepo>*  This adds an additional remote repository reference to your local PC repository.

**git remote S**hows/lists the names of all remote repositories at their respective origin/master URL locations.

**git fetch** <*origin/main*> Downloads/places the info/data into the local repo but DOES NOT merge to the branch.

**git merge** <*origin/main*> <*main*>

Above command DOES Merge the files from the remote origin main to the local main branch.

File versions have been incorporated into the local repository and is up- to-date with the remote

[For me at work I most likely will be using the ‘integration’ branch as the local target.

After working on the files within the local repository or creating new ones, they should be pushed to the remote repository origin master.

**git pull** Downloads/places info/data into the local repo AND merges to branch where HEAD is positioned.

Reverting to Previous Versions: To work on a previous version from the command-line console within the ‘Bash’ terminal,

Checkout the version using the hash ID copied from the local repository git log command display output or the remote repository site history list of commits.

**git log** <-*#*> List the ‘#’ number of past commits with their hash IDs. Copy the hash ID of commit to checkout.

**git checkout** <*hash ID*> Checkout using the hash ID previously copied from the remote repository site or local log.

This will place you in a ‘detached HEAD’ state where file modifications will not affect the local main branch integrity of the files. The ‘Head’ will reside on the branch of the 40 char hash ID or its abbreviation.

**git checkout -b** <*new-branch*> Makes a new branch to retain commits you create, (now or later) by using this

**git status** Red lettering indicates files new or modified. Green lettering indicates staged files.

**git log** Do ‘git status’ and ‘git log’ gives the content/configuration of the detached HEAD state.

(Notice the pointer is not pointing at a branch, or master only ‘HEAD’). Careful! Here.

Do not do too much with the ‘HEAD’ detached it may be hard to get back to the ‘master’.

Forking (Copy existing Repo) Forking copies existing repository into the local repository (Cannot change / edit remote content)

Places the entire project into one’s local profile / account. One can clone this copied / forked content into one’s local repository for editing. Creates a separate repository on the local PC.

One does not have editing rights to the original repository! But once on your profile / account on your local machine, one can push it to one’s remote repository.

Git On-Line Documentation:*https://git-scm.com/doc* Cover Sheet (Color) Retrieved from:

**Tutorials:** Getting started with Git: Understand the Git Version Control System and How to Use it at a Basic Level (From O’Reilly)

**Textbook:** URL: git-scm.com/book/en/v2

Pulling Sequence? Before doing a git pull …  add/commit/stash any work-in-progress first!

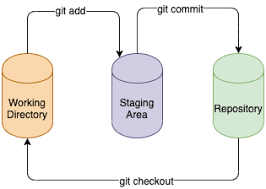
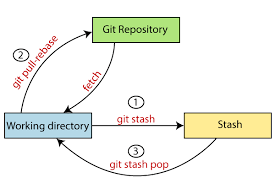
git fetch                               Retrieve the latest meta-data info from the original (yet doesn't do any file transferring.

git diff …. origin

You can do a **git fetch** at any time to update your remote-tracking branches under refs/remotes/<remote>/. This operation never changes any of your own local branches under refs/heads, and is safe to do without changing your working copy. I have even heard of people running git fetch periodically in an icon job in the background (although I wouldn't recommend doing this).

A **git pull** is what you would do to bring a local branch up-to-date with its remote version, while also updating your other remote-tracking branches.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Graphics** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Git Commands** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(List of commands were copied from my Git application following a ‘ git help -a ‘ command)

**Main Porcelain Commands**

add Add file contents to the index

am Apply a series of patches from a mailbox

archive Create an archive of files from a named tree

bisect Use binary search to find the commit that introduced a bug

branch List, create, or delete branches

bundle Move objects and refs by archive

checkout Switch branches or restore working tree files

cherry-pick Apply the changes introduced by some existing commits

citool Graphical alternative to git-commit

clean Remove untracked files from the working tree

clone Clone a repository into a new directory

commit Record changes to the repository

describe Give an object a human readable name based on an available ref

diff Show changes between commits, commit and working tree, etc

fetch Download objects and refs from another repository

format-patch Prepare patches for e-mail submission

gc Cleanup unnecessary files and optimize the local repository

**gitk** The Git repository browser

grep Print lines matching a pattern

gui A portable graphical interface to Git

init Create an empty Git repository or reinitialize an existing one

log Show commit logs

merge Join two or more development histories together

mv Move or rename a file, a directory, or a symlink

notes Add or inspect object notes

pull Fetch from and integrate with another repository or a local branch

push Update remote refs along with associated objects

range-diff Compare two commit ranges (e.g. two versions of a branch)

rebase Reapply commits on top of another base tip

reset Reset current HEAD to the specified state

restore Restore working tree files

revert Revert some existing commits

rm Remove files from the working tree and from the index

shortlog Summarize 'git log' output

show Show various types of objects

stash Stash the changes in a dirty working directory away

status Show the working tree status

submodule Initialize, update or inspect submodules

switch Switch branches

tag Create, list, delete or verify a tag object signed with GPG

worktree Manage multiple working trees

**Ancillary Commands / Manipulators**

config Get and set repository or global options

fast-export Git data exporter

fast-import Backend for fast Git data importers

filter-branch Rewrite branches

mergetool Run merge conflict resolution tools to resolve merge conflicts

pack-refs Pack heads and tags for efficient repository access

prune Prune all unreachable objects from the object database

reflog Manage reflog information

remote Manage set of tracked repositories

repack Pack unpacked objects in a repository

replace Create, list, delete refs to replace objects

**Ancillary Commands / Interrogators**

annotate Annotate file lines with commit information

blame Show what revision and author last modified each line of a file

count-objects Count unpacked number of objects and their disk consumption

difftool Show changes using common diff tools

fsck Verifies the connectivity and validity of the objects in the database

gitweb Git web interface (web frontend to Git repositories)

help Display help information about Git

instaweb Instantly browse your working repository in gitweb

merge-tree Show three-way merge without touching index

rerere Reuse recorded resolution of conflicted merges

show-branch Show branches and their commits

verify-commit Check the GPG signature of commits

verify-tag Check the GPG signature of tags

whatchanged Show logs with difference each commit introduces

**Interacting with Others**

archimport Import a GNU Arch repository into Git

cvsexportcommit Export a single commit to a CVS checkout

cvsimport Salvage your data out of another SCM people love to hate

cvsserver A CVS server emulator for Git

imap-send Send a collection of patches from stdin to an IMAP folder

p4 Import from and submit to Perforce repositories

quiltimport Applies a quilt patchset onto the current branch

request-pull Generates a summary of pending changes

send-email Send a collection of patches as emails

svn Bidirectional operation between a Subversion repository and Git

**Low-level Commands / Manipulators**

apply Apply a patch to files and/or to the index

checkout-index Copy files from the index to the working tree

commit-graph Write and verify Git commit-graph files

commit-tree Create a new commit object

hash-object Compute object ID and optionally creates a blob from a file

index-pack Build pack index file for an existing packed archive

merge-file Run a three-way file merge

merge-index Run a merge for files needing merging

mktag Creates a tag object

mktree Build a tree-object from ls-tree formatted text

multi-pack-index Write and verify multi-pack-indexes

pack-objects Create a packed archive of objects

prune-packed Remove extra objects that are already in pack files

read-tree Reads tree information into the index

symbolic-ref Read, modify and delete symbolic refs

unpack-objects Unpack objects from a packed archive

update-index Register file contents in the working tree to the index

update-ref Update the object name stored in a ref safely

write-tree Create a tree object from the current index

**Low-level Commands / Interrogators**

cat-file Provide content or type and size information for repository objects

cherry Find commits yet to be applied to upstream

diff-files Compares files in the working tree and the index

diff-index Compare a tree to the working tree or index

diff-tree Compares the content and mode of blobs found via two tree objects

for-each-ref Output information on each ref

get-tar-commit-id Extract commit ID from an archive created using git-archive

ls-files Show information about files in the index and the working tree

ls-remote List references in a remote repository

ls-tree List the contents of a tree object