An Introduction to Git Concepts

What is GIT?

Git is a distributed version control system that monitors changes made to any set of computer files. It is typically used to coordinate the work of programmers who are working together to build source code for software.

But Why Git?

Advantages of using Git

Speed and Performance

Git is designed to be fast and efficient, even with large repositories and extensive history.

Efficient Co-Ordination

Git facilitates the collaboration between developers, enabling effective teamwork.

Branching and Merging

Git Branching helps create seperate lines of development, thus not having to alter the main codebase

Time Travel

Git keeps a history of all commits, enabling users to access and edit previous versions of the files

Few Basics of Git

Before we move on, let us look at some basic Git concepts and terms

Git command/term	Their function
Repository	A central location where version-controlled files are stored.
git init	init, short for initialize, initializes a new repository in the current directory.
git add	Adds a file into the staging area
git commit	Commits the changes from the staging area into the repository, with a message

Git command/term	Their function
git status	Shows the current status of the repository, along with modified and committed files.
git log	DIsplays us the commit history, showing details of the author, time and the commit message
Branches	This is a independent line of development, branched from the main base, allowing for better management.

Let us now look at the functions

Tip: Use links to go to a different page inside your presentation.

How: Highlight text, click on the link symbol on the toolbar, and select the page in your presentation you want to connect.

Git stash

Git bisect

Git reflog

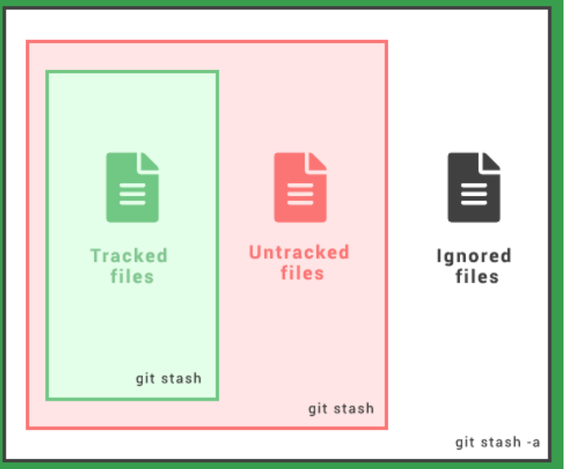
Git switch

Git rebase

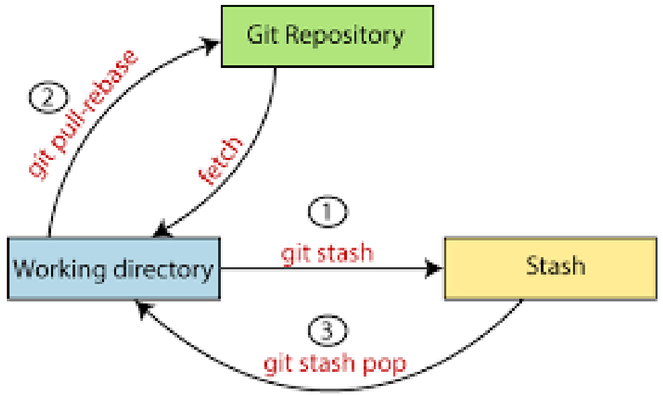
Git cherry-pick

Git diff

Git Stash



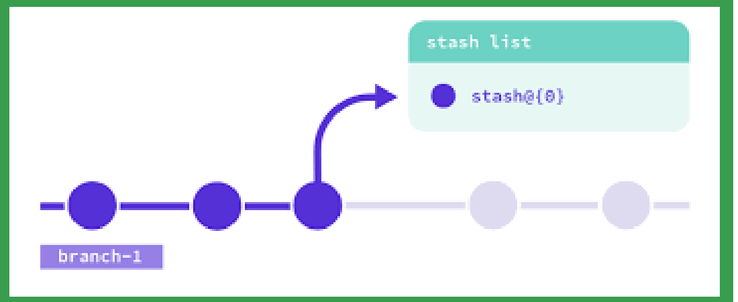
- Let us say, we are working on some code and suddenly, we have to switch to another branch to check out some other issue.
- But, you wouldn't commit uncompleted code, and you don't want to loose the entire progress either. So, you use **Git Stash.**
- **Git stash** is a function which lets us to temporarily save our changes without committing them.
- **Git stash** stores all your uncommitted files in a new stash, cleaning your working directory.



Git Stash

```
MINGW64:/c/Users/Sri Chaitanya/Desktop/lmao

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$ git status
On branch master
Your branch is up to date with 'Lmao/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   code.c
        modified:   code.exe

no changes added to commit (use "git add" and/or "git commit -a")

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$ git stash
Saved working directory and index state WIP on master: ab0fa6c This might work

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$ git status
On branch master
Your branch is up to date with 'Lmao/master'.

nothing to commit, working tree clean

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$ git stash list
stash@{0}: WIP on master: ab0fa6c This might work
stash@{1}: WIP on master: 2ba09a1 This is the base commit
stash@{2}: WIP on master: 2ba09a1 This is the base commit
```

Clear working directory after stashing, with the stash list



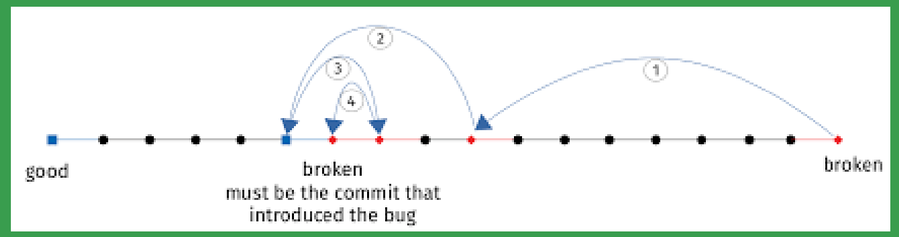
- To retrieve the files in stash into the working directory,we can use:-
  a) **git stash apply**
  b) **git stash pop**

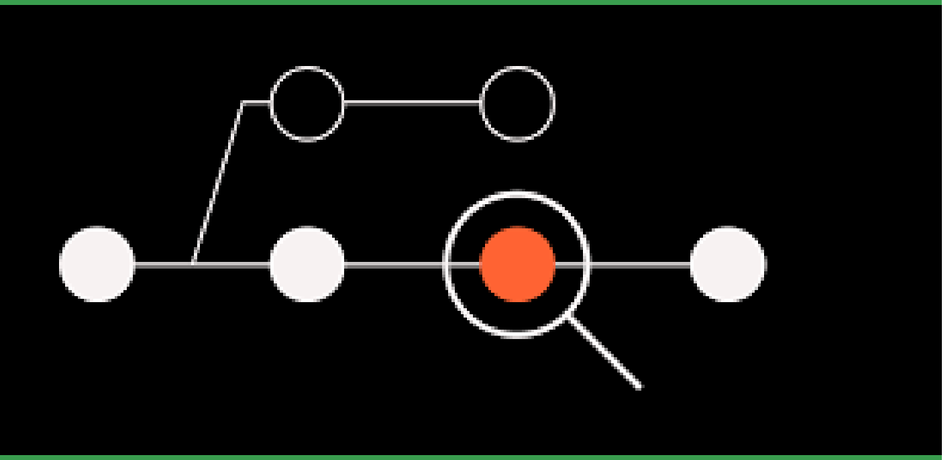
- When using apply, the files aren't deleted from the stash . However, with pop, the files are essentially cut from the stash.

- The other important variants of git stash include using **--keep- index** to stash only the unstaged files, or stashing even untracked files by using **--include-untracked**, or **-u**.

- In case we have merge conflicts while applying, we cna create a new branch with the stash using
  git stash branch <newbranchname>

Git Bisect





- **git bisect** is a very powerful tool that helps us identify where exactly a bug has been introduced into the project.

- It uses binary search among the commits to find out where the bad commit is .

- We start bisect by using **git bisect start**, and specify **git bisect bad** and **git bisect good <commit name>** to give the start and end.

- At each iteration, we have to specify whether the commit it shows is good or bad. Eventually , it specifies out the problematic commit.
- At the end, we must use **git bisect reset** to reset the head to where we were before we started.

Process of Bisecting

```
Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$ git bisect start
status: waiting for both good and bad commits

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master|BISECTING)
$ git bisect bad
status: waiting for good commit(s), bad commit known
```

```
Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master|BISECTING)
$ git bisect good f1a32
Bisecting: 0 revisions left to test after this (roughly 1 step)
[606619e93658da8d96626d3dc48fe651436a15a4] 4th
```

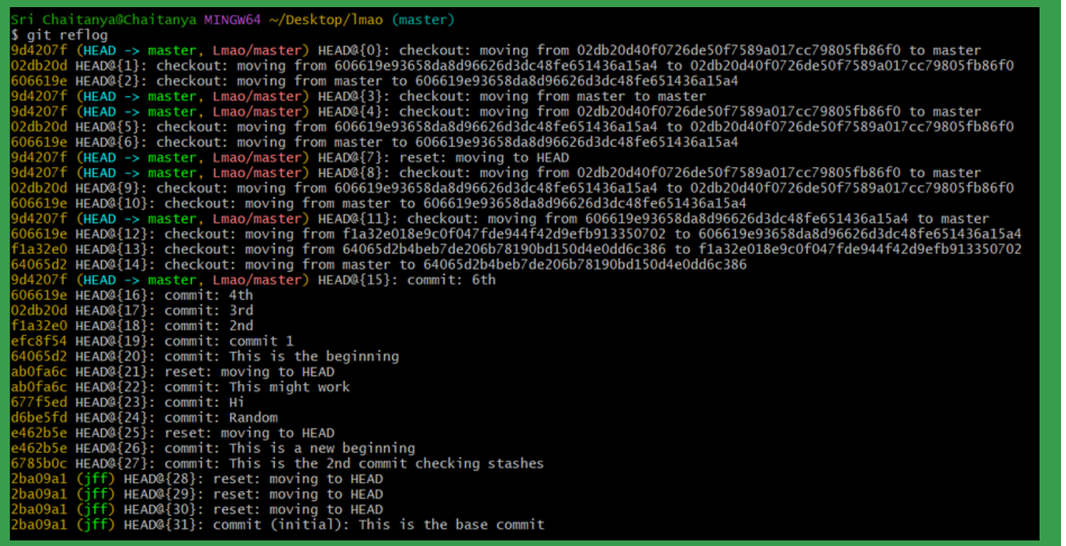
```
Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master|BISECTING)
$ git bisect good f1a32
Bisecting: 0 revisions left to test after this (roughly 1 step)
[606619e93658da8d96626d3dc48fe651436a15a4] 4th

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao ((606619e...)|BISECTI
$ git bisect bad
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[02db20d40f0726de50f7589a017cc79805fb86f0] 3rd

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao ((02db20d...)|BISECTI
$
```

```
#include <stdio.h>

int main(){
    printf("Hello World\n");
    // HI
    // lmao
    // here's not the bad one
    wpgjpe
}
```

```
Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master|BISECTING)
$ git bisect good f1a32
Bisecting: 0 revisions left to test after this (roughly 1 step)
[606619e93658da8d96626d3dc48fe651436a15a4] 4th

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao ((606619e...)|BISECTING)
$ git bisect bad
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[02db20d40f0726de50f7589a017cc79805fb86f0] 3rd

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao ((02db20d...)|BISECTING)
$ git bisect good
606619e93658da8d96626d3dc48fe651436a15a4 is the first bad commit
commit 606619e93658da8d96626d3dc48fe651436a15a4
Author: Chaitanya729 <rockstarman957@gmail.com>
Date:   Wed May 24 23:10:17 2023 +0530

    4th

 code.c | 1 +
 1 file changed, 1 insertion(+)

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao ((02db20d...)|BISECTING)
$
```

```
#include <stdio.h>

int main(){
    printf("Hello World\n");
    // HI
    // lmao
    // here's not the bad one
}
```

Git Reflog



- The **git reflog** command is used to review the reference log , which is a detailed history of all reference changes like branch creations, commits, merges,rebases,etc.

- **git reflog** is yet another powerful tool , which can be used to perform various functions such as:-
  a) recovering lost commits
  b) undo operation
  c) Review history of operations

- We can browse through the reflogs using realtive references.

- Since the whole point of git is to take snapshost of files at different instant, any deleted files over a period of time can be bought back with the help of reflog

Git Switch

```
Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$ git branch
  jff
  list
* master
  new

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$ git switch new
Switched to branch 'new'

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (new)
$ git branch
  jff
  list
  master
* new

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (new)
$ git switch -c new_branch
Switched to a new branch 'new_branch'

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (new_branch)
$ gi branch
bash: gi: command not found

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (new_branch)
$ git branch
  jff
  list
  master
  new
* new_branch
```

- **git switch** is a command whose main purpose is to switch between branches.

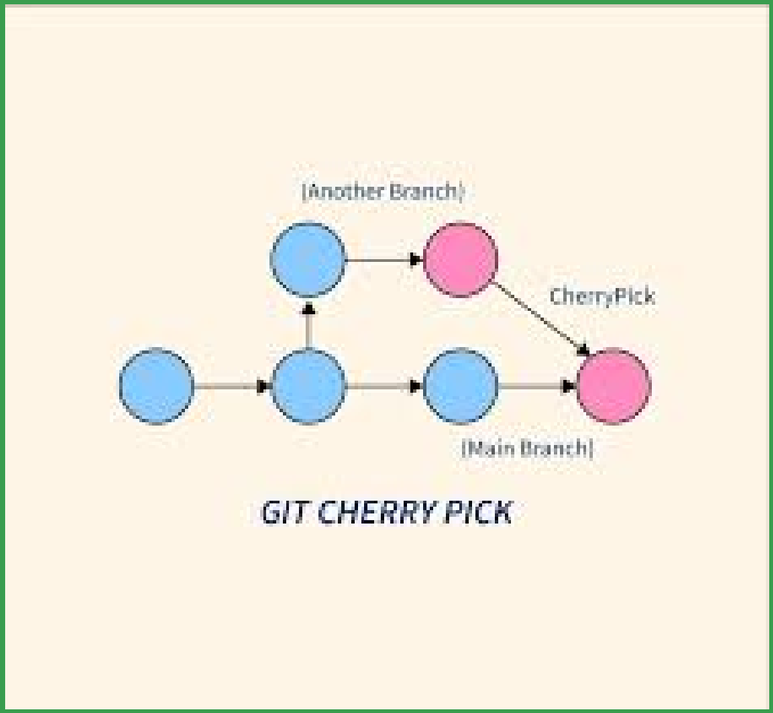
- It allows us to update our working directory and switch to the branch we specify, the code being
  git branch <branchname>

- using **git switch -c <new-branchname>**, we can create a new branch and directly swithc into it.

- when using **git switch** , git will take our uncommitted changes into the branch we want to switch to, incase there are no merge conflicts.

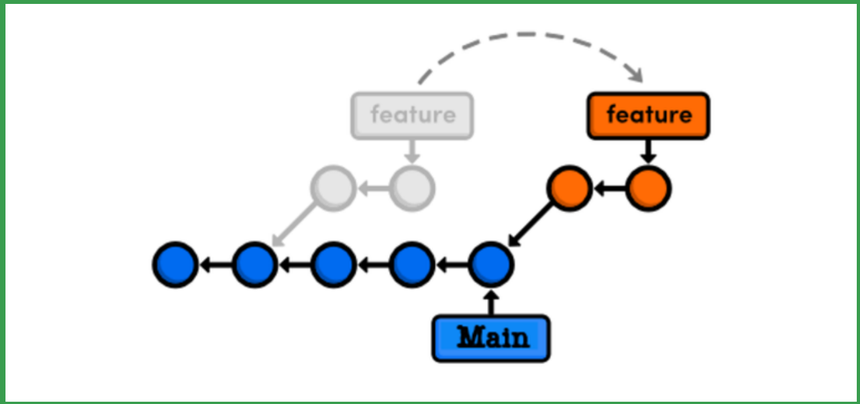
- **git switch** also allows us to stich to a certain commit, if we specify the commit number.

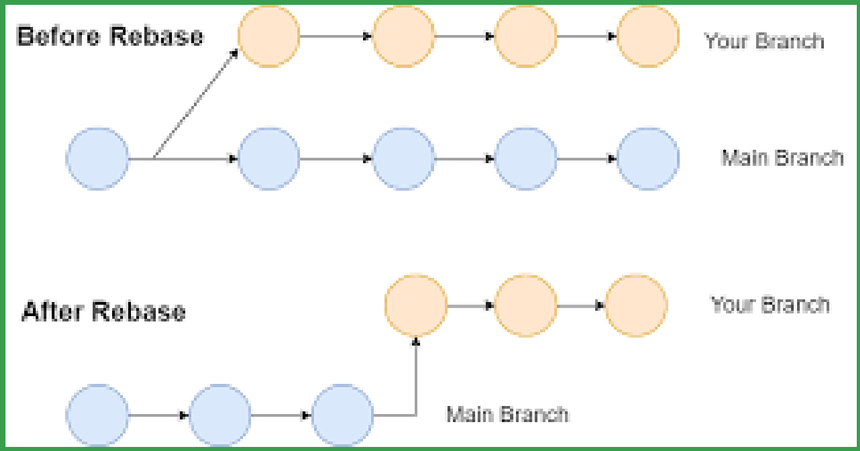
Git cherry-pick



- As the name suggests, **git cherry-pick** is used to pick a desired commit(chnages) of our choice, and apply it onto another branch, selectively.
- After using **git cherry-pick<commit-hash>**, the new commits created however will have new hashes on the branch where we send them.
- This is often used to have a linear history for a branch.
- Multiple commits can be cherry-picked at a time, using **git cherry-pick<start-commit>..<end-commit>**.
- If conflicts appear during cherry-picking, git tells us that there have been conflicts and we have to manually sort them out .

Git Rebase

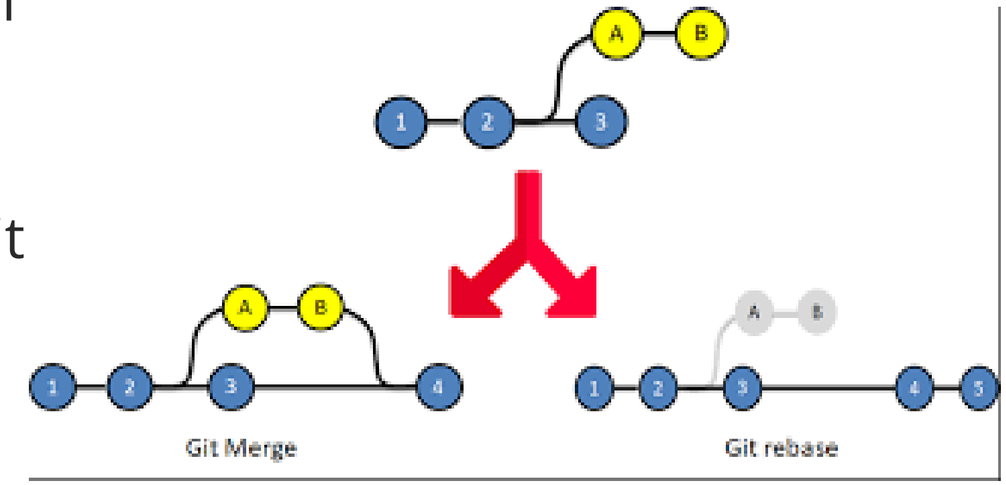




- **git rebase** is basically an automated cherry-pick. It determines a series of commits and apply them onto another branch.
- The general syntax is **git rebase <upstream>**, where upstream is basically the main branch, onto which the changes are added.
- If conflicts occur during rebase, we can resolve them and continue the process using **git rebase --continue**.
- Since the branch is completely modified while using rebase, we can fork the branch to keep a copy of the branch.
- While merge combines the commit history of the two branches, rebase modifies the commit history of the current branch.

- Rebase is often considered a risky yet powerful function and must be used with care.

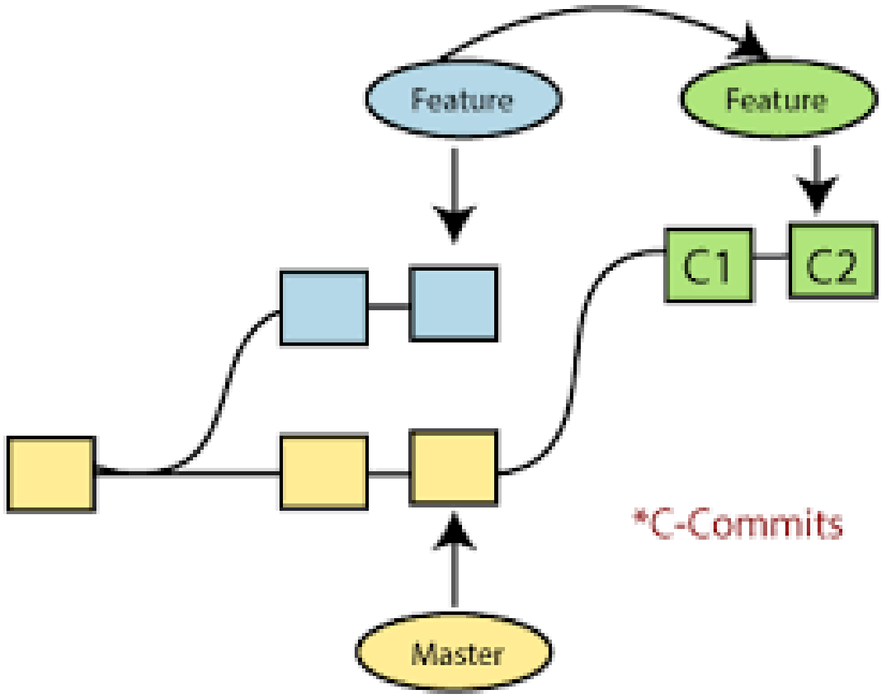
- Firstly, the original commit context can be lost , making it difficult to track the history of the files, or understand the reason for changes.



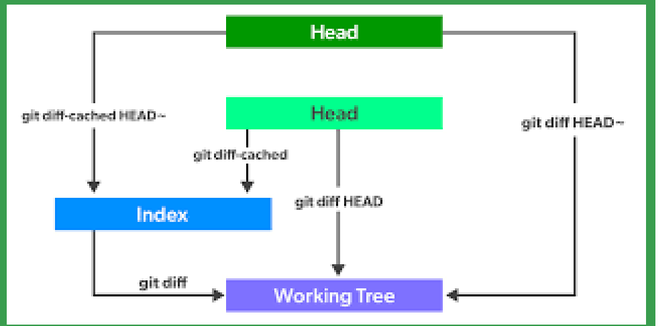
- Rebasing often leads to conflicts leading to time loss as it requires manual resolution.

- Data can easily be lost while rebasing. Hence, it is always better to fork(clone) our branch initially.

- Frequent rebasing, as said, makes the commit line too linear, making it difficult to understand as to how the project originally developed. Therefore, one must be co-ordinated with his fellow developers while using this.



Git Diff



- **git diff** , when used, gives us the difference between the files in our working directory, and the file sin the repository.
- If the files are staged and **diff** is used, there"ll be difference. To check staged files against repository, the command **git diff --staged** is used.
- **git diff** can also be used to check the differences between a pair of commits or branches. By specifying file paths or names, even their difference can be checked.
- The output of **git diff** gives a line by line log of what has been added,removed So, this feature can be used hand in hand with other git functions to identify exact differences when needed.
- One example is with **git bisect**, with which we can identify the exact difference between the first bad commit and the good one, identifying the bug quite easily.

```c
#include <stdio.h>

int main(){
    printf("Hello World\n");
    return 0;
}
```

The original code

```
Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$ git status
On branch master
Your branch is ahead of 'Lmao/master' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$ git diff

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$
```

```c
#include <stdio.h>

int main(){
    printf("Hello World\n");
    printf("Hi evryone\n");
    return 0;
}
```

Modified code

```
Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$ git diff
diff --git a/code.c b/code.c
index 3b47771..78f4cd7 100644
--- a/code.c
+++ b/code.c
@@ -2,5 +2,6 @@

 int main(){
     printf("Hello World\n");
+    printf("Hi evryone\n");
     return 0;
 }
\ No newline at end of file
diff --git a/code.exe b/code.exe
index ee4aced..4f1d139 100644
Binary files a/code.exe and b/code.exe differ

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$
```

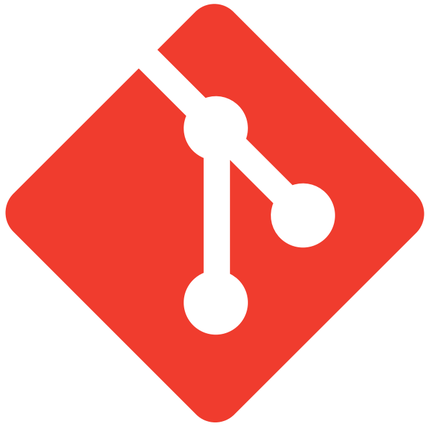
Changes detected, and shown

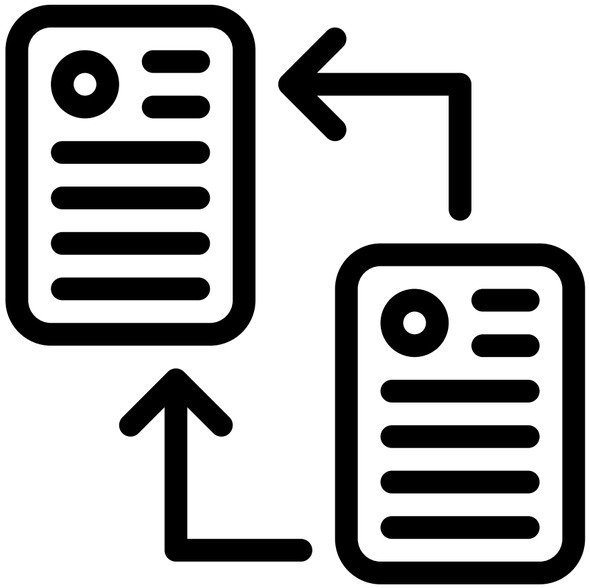
```
Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$ git add .

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$ git diff

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$ git diff --staged
diff --git a/code.c b/code.c
index 3b47771..78f4cd7 100644
--- a/code.c
+++ b/code.c
@@ -2,5 +2,6 @@

 int main(){
     printf("Hello World\n");
+    printf("Hi evryone\n");
     return 0;
 }
\ No newline at end of file
diff --git a/code.exe b/code.exe
index ee4aced..62c53e6 100644
Binary files a/code.exe and b/code.exe differ

Sri Chaitanya@Chaitanya MINGW64 ~/Desktop/lmao (master)
$
```

Code in staging area isn't detected. Using --staged however, works.

Conclusion

- These are a few git functions, briefly explained.
- Git contains a lot more useful functions, which make it a very powerful too to work with.
- Learning how to operate and use git is very essential for any developer, especially for open source developers





“

The power of Open Source is the power of the people. The people rule.

Philippe Kahn

“

Open source development empowers collaboration and innovation, where the collective knowledge of the community paves the way for boundless possibilities, driving progress that knows no boundaries.

ChatGPT

Thank You !!!

B. Sri Chaitanya
22CS10018