# Git installation

yum install -y git

git –version

mkdir practice

cd practice

git init => Initialized empty Git repository in /root/practice/.git/ that is .git

touch 12Aug.txt

# Creating branches and adding files

git status

On branch master

No commits yet

Untracked files:

(use "git add <file>..." to include in what will be committed)

12Aug.txt

nothing added to commit but untracked files present (use "git add" to track)

git add .

git status

On branch master

No commits yet

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: 12Aug.txt

git rm --cached 12Aug.txt remove the file

git status

On branch master

No commits yet

Untracked files:

(use "git add <file>..." to include in what will be committed)

12Aug.txt

nothing added to commit but untracked files present (use "git add" to track)

git branch

git add .

git commit . -m "This is my first commit"

[master (root-commit) c811b9f] This is my first commit

Committer: root <root@ip-172-31-33-83.ap-south-1.compute.internal>

Your name and email address were configured automatically based on your username and hostname. Please check that they are accurate. You can suppress this message by setting them explicitly. Run the following command and follow the instructions in your editor to edit

your configuration file:

git config --global --edit

After doing this, you may fix the identity used for this commit with:

git commit --amend --reset-author

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 12Aug.txt

git log

commit c811b9f588d204bd4d6e16493d21ba189db1958c (HEAD -> master)

Author: root <root@ip-172-31-33-83.ap-south-1.compute.internal>

Date: Thu Aug 15 18:12:18 2019 +0000

This is my first commit

git ls-files

git branch

\* master

git branch dev

git branch

dev

\* master

git checkout dev => switch to dev branch

git branch

touch 13Aug.txt => add a file in dev branch

git commit . -m "This is second commit"

On branch dev

Untracked files:

13Aug.txt

nothing added to commit but untracked files present

git add .

git commit . -m "This is second commit"

[dev d18c3c0] This is second commit

Committer: root <root@ip-172-31-33-83.ap-south-1.compute.internal>

Your name and email address were configured automatically based on your username and hostname. Please check that they are accurate. You can suppress this message by setting them explicitly. Run the following command and follow the instructions in your editor to edit

your configuration file:

git config --global --edit

After doing this, you may fix the identity used for this commit with:

git commit --amend --reset-author

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 13Aug.txt

git branch

\* dev

master

git checkout master

ll

total 0

-rw-r--r-- 1 root root 0 Aug 15 17:55 12Aug.txt

git ls-files

# Merging files added at dev to master branch

git merge dev => adding the newly added files in dev branch to master branch

Updating c811b9f..d18c3c0

Fast-forward

13Aug.txt | 0

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 13Aug.txt

git ls-files

12Aug.txt

13Aug.txt

git branch -d dev

Deleted branch dev (was d18c3c0).

git branch

\* master

# Merging conflict files to master

git branch dev => adding dev branch again

git checkout dev

touch 14Aug.txt => add a file in dev branch

git add .

git commit . -m "adding to dev branch "

vi 14Aug.txt => adding content to the file

git status

On branch dev

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: 14Aug.txt

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

git add .

git commit . -m "updating the changes in dev branch"

git checkout master => switch to master branch

vi 14Aug.txt => adding this file in master branch

git add .

git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: 14Aug.txt

git commit . -m "Master"

[master 2a6a858] Master

1 file changed, 2 insertions(+)

create mode 100644 14Aug.txt

git merge dev

Auto-merging 14Aug.txt

CONFLICT (add/add): Merge conflict in 14Aug.txt

Automatic merge failed; fix conflicts and then commit the result.

git config merge.tool vimdiff => merge tool vimdiff is used to resolve the conflict issue

git config merge.conflictstyle diff3

git config merge.tool vimdiff

git config mergetool.prompt false

git mergetool

Merging:

14Aug.txt

Normal merge conflict for 14Aug.txt:

{local}: created file

{remote}: created file

3 files to edit

Do the necessary changes in the merged file and close all the files by :w and :q commands

ll

total 8

-rw-r--r-- 1 root root 0 Aug 15 17:55 12Aug.txt

-rw-r--r-- 1 root root 0 Aug 16 12:25 13Aug.txt

-rw-r--r-- 1 root root 53 Aug 17 10:05 14Aug.txt

-rw-r--r-- 1 root root 83 Aug 17 09:48 14Aug.txt.orig

cat 14Aug.txt

Master Branch

script

Adding content in dev branch

cat 14Aug.txt.orig

<<<<<<< HEAD

Master Branch

script

=======

Adding content in dev branch

>>>>>>> dev

git status

On branch master

All conflicts fixed but you are still merging.

(use "git commit" to conclude merge)

Changes to be committed:

modified: 14Aug.txt

Untracked files:

(use "git add <file>..." to include in what will be committed)

14Aug.txt.orig

To consider 14Aug.txt.orig , add this file

git add .

git commit -a or git commit -a --allow-empty-message -m ''

[master a5562db] Merge branch 'dev'

git status

On branch master

nothing to commit, working tree clean

rm 14Aug.txt

rm: remove regular file ‘14Aug.txt’? y

mv 14Aug.txt.orig 14Aug.txt

git ls-files

12Aug.txt

13Aug.txt

14Aug.txt

To reflect the file deleted and renaming file name.

git add .

git commit -a

# GIT Tutorial

<https://opensource.com/article/18/1/step-step-guide-git>

<https://www.tutorialspoint.com/git/git_commit_changes.htm>

Version Control System

**Version Control System (VCS)** is a software that helps software developers to work together and maintain a complete history of their work.

Listed below are the functions of a VCS −

* Allows developers to work simultaneously.
* Does not allow overwriting each other’s changes.
* Maintains a history of every version.

Following are the types of VCS −

* Centralized version control system (CVCS).
* Distributed/Decentralized version control system (DVCS).

In this chapter, we will concentrate only on distributed version control system and especially on Git. Git falls under distributed version control system.

**Examples of Centralized version control systems** like Subversion (SVN), CVS, and Perforce.

Centralized Version Control

Centralized version control systems are based on the idea that there is a single “central” copy of your project somewhere (probably on a server), and programmers will “commit” their changes to this central copy.

“Committing” a change simply means recording the change in the central system. Other programmers can then see this change. They can also pull down the change, and the version control tool will automatically update the contents of any files that were changed.

Most modern version control systems deal with “changesets,” which simply are a groups of changes (possibly to many files) that should be treated as a cohesive whole. For example: a change to a C header file and the corresponding .c file should always be kept together.

Programmers no longer have to keep many copies of files on their hard drives manually, because the version control tool can talk to the central copy and retrieve any version they need on the fly.

Some of the most common centralized version control systems you may have heard of or used are CVS, Subversion (or SVN) and Perforce.

A Typical Centralized Version Control Workflow

When you’re working with a centralized verison control system, your workflow for adding a new feature or fixing a bug in your project will usually look something like this:

* Pull down any changes other people have made from the central server.
* Make your changes, and make sure they work properly.
* Commit your changes to the central server, so other programmers can see them.

Distributed Version Control System

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. But the major drawback of CVCS is its single point of failure, i.e., failure of the central server. Unfortunately, if the central server goes down for an hour, then during that hour, no one can collaborate at all. And even in a worst case, if the disk of the central server gets corrupted and proper backup has not been taken, then you will lose the entire history of the project. Here, distributed version control system (DVCS) comes into picture.

DVCS clients not only check out the latest snapshot of the directory but they also fully mirror the repository. If the server goes down, then the repository from any client can be copied back to the server to restore it. Every checkout is a full backup of the repository. Git does not rely on the central server and that is why you can perform many operations when you are offline. You can commit changes, create branches, view logs, and perform other operations when you are offline. You require network connection only to publish your changes and take the latest changes.

Advantages of Git

Free and open source

Git is released under GPL’s open source license. It is available freely over the internet. You can use Git to manage property projects without paying a single penny. As it is an open source, you can download its source code and also perform changes according to your requirements.

Fast and small

As most of the operations are performed locally, it gives a huge benefit in terms of speed. Git does not rely on the central server; that is why, there is no need to interact with the remote server for every operation. The core part of Git is written in C, which avoids runtime overheads associated with other high-level languages. Though Git mirrors entire repository, the size of the data on the client side is small. This illustrates the efficiency of Git at compressing and storing data on the client side.

Implicit backup

The chances of losing data are very rare when there are multiple copies of it. Data present on any client side mirrors the repository, hence it can be used in the event of a crash or disk corruption.

Security

Git uses a common cryptographic hash function called secure hash function (SHA1), to name and identify objects within its database. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. It implies that, it is impossible to change file, date, and commit message and any other data from the Git database without knowing Git.

Easier branching

CVCS uses cheap copy mechanism, If we create a new branch, it will copy all the codes to the new branch, so it is time-consuming and not efficient. Also, deletion and merging of branches in CVCS is complicated and time-consuming. But branch management with Git is very simple. It takes only a few seconds to create, delete, and merge branches.

DVCS Terminologies

Local Repository

Every VCS tool provides a private workplace as a working copy. Developers make changes in their private workplace and after commit, these changes become a part of the repository. Git takes it one step further by providing them a private copy of the whole repository. Users can perform many operations with this repository such as add file, remove file, rename file, move file, commit changes, and many more.

Working Directory and Staging Area or Index

The working directory is the place where files are checked out. In other CVCS, developers generally make modifications and commit their changes directly to the repository. But Git uses a different strategy. Git doesn’t track each and every modified file. Whenever you do commit an operation, Git looks for the files present in the staging area. Only those files present in the staging area are considered for commit and not all the modified files.

Let us see the basic workflow of Git.

**Step 1** − You modify a file from the working directory.

**Step 2** − You add these files to the staging area.

**Step 3** − You perform commit operation that moves the files from the staging area. After push operation, it stores the changes permanently to the Git repository.

The **staging area** is a simple file, generally contained in **Git** directory, that stores information about what will go into your next commit.



Suppose you modified two files, namely “sort.c” and “search.c” and you want two different commits for each operation. You can add one file in the staging area and do commit. After the first commit, repeat the same procedure for another file.

# First commit

[bash]$ git add sort.c

# adds file to the staging area

[bash]$ git commit –m “Added sort operation”

# Second commit

[bash]$ git add search.c

# adds file to the staging area

[bash]$ git commit –m “Added search operation”

Blobs

Blob stands for **B**inary **L**arge **Ob**ject. Each version of a file is represented by blob. A blob holds the file data but doesn’t contain any metadata about the file. It is a binary file, and in Git database, it is named as SHA1 hash of that file. In Git, files are not addressed by names. Everything is content-addressed.

Trees

Tree is an object, which represents a directory. It holds blobs as well as other sub-directories. A tree is a binary file that stores references to blobs and trees which are also named as **SHA1** hash of the tree object.

Commits

Commit holds the current state of the repository. A commit is also named by **SHA1**hash. You can consider a commit object as a node of the linked list. Every commit object has a pointer to the parent commit object. From a given commit, you can traverse back by looking at the parent pointer to view the history of the commit. If a commit has multiple parent commits, then that particular commit has been created by merging two branches.

Branches

Branches are used to create another line of development. By default, Git has a master branch, which is same as trunk in Subversion. Usually, a branch is created to work on a new feature. Once the feature is completed, it is merged back with the master branch and we delete the branch. Every branch is referenced by HEAD, which points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit.

Tags

Tag assigns a meaningful name with a specific version in the repository. Tags are very similar to branches, but the difference is that tags are immutable. It means, tag is a branch, which nobody intends to modify. Once a tag is created for a particular commit, even if you create a new commit, it will not be updated. Usually, developers create tags for product releases.

Clone

Clone operation creates the instance of the repository. Clone operation not only checks out the working copy, but it also mirrors the complete repository. Users can perform many operations with this local repository. The only time networking gets involved is when the repository instances are being synchronized.

Pull

Pull operation copies the changes from a remote repository instance to a local one. The pull operation is used for synchronization between two repository instances. This is same as the update operation in Subversion.

Push

Push operation copies changes from a local repository instance to a remote one. This is used to store the changes permanently into the Git repository. This is same as the commit operation in Subversion.

HEAD

HEAD is a pointer, which always points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit. The heads of the branches are stored in **.git/refs/heads/** directory.

[CentOS]$ ls -1 .git/refs/heads/

master

[CentOS]$ cat .git/refs/heads/master

570837e7d58fa4bccd86cb575d884502188b0c49

Revision

Revision represents the version of the source code. Revisions in Git are represented by commits. These commits are identified by **SHA1** secure hashes

Customize Git Environment

Git provides the git config tool, which allows you to set configuration variables. Git stores all global configurations in **.gitconfig** file, which is located in your home directory. To set these configuration values as global, add the **--global** option, and if you omit **--global** option, then your configurations are specific for the current Git repository.

You can also set up system wide configuration. Git stores these values in the **/etc/gitconfig** file, which contains the configuration for every user and repository on the system. To set these values, you must have the root rights and use the **--system**option.

When the above code is compiled and executed, it produces the following result −

Setting username

This information is used by Git for each commit.

[jerry@CentOS project]$ git config --global user.name "Jerry Mouse"

Setting email id

This information is used by Git for each commit.

[jerry@CentOS project]$ git config --global user.email [jerry@tutorialspoint.com](mailto:jerry@tutorialspoint.com)

Setting default merge tool

Git does not provide a default merge tool for integrating conflicting changes into your working tree. We can set default merge tool by enabling following settings.

[jerry@CentOS project]$ git config --global merge.tool vimdiff

Listing Git settings

To verify your Git settings of the local repository, use **git config –list** command as given below.

[jerry@CentOS ~]$ git config --list

Git – Life Cycle

General workflow is as follows −

* You clone the Git repository as a working copy.
* You modify the working copy by adding/editing files.
* If necessary, you also update the working copy by taking other developer's changes.
* You review the changes before commit.
* You commit changes. If everything is fine, then you push the changes to the repository.
* After committing, if you realize something is wrong, then you correct the last commit and push the changes to the repository.

Shown below is the pictorial representation of the work-flow.



Git - Create Operation

Create a Bare Repository

Let us initialize a new repository by using **init** command followed by **--bare** option. It initializes the repository without a working directory. By convention, the bare repository must be named as **.git**.

**[gituser@CentOS ~]$ pwd**

**/home/gituser**

**[gituser@CentOS ~]$ mkdir project.git**

**[gituser@CentOS ~]$ cd project.git/**

**[gituser@CentOS project.git]$ ls**

**[gituser@CentOS project.git]$ git --bare init**

**Initialized empty Git repository in /home/gituser-m/project.git/**

**[gituser@CentOS project.git]$ ls**

**branches config description HEAD hooks info objects refs**

Push Changes to the Repository

Git init command creates **.git** directory to store metadata about the repository every time it reads the configuration from the **.git/config** file.

**[tom@CentOS ~]$ pwd**

**/home/tom**

**[tom@CentOS ~]$ mkdir tom\_repo**

**[tom@CentOS ~]$ cd tom\_repo/**

**[tom@CentOS tom\_repo]$ git init**

**Initialized empty Git repository in /home/tom/tom\_repo/.git/**

**[tom@CentOS tom\_repo]$ echo 'TODO: Add contents for README' > README**

**[tom@CentOS tom\_repo]$ git status -s**

**?? README**

**[tom@CentOS tom\_repo]$ git add .**

**[tom@CentOS tom\_repo]$ git status -s**

**A README**

**[tom@CentOS tom\_repo]$ git commit -m 'Initial commit'**

**[tom@CentOS tom\_repo]$ git log**

Git pushes only to matching branches: For every branch that exists on the local side, the remote side is updated if a branch with the same name already exists there. In our tutorials, every time we push changes to the **origin master** branch, use appropriate branch name according to your requirement.

**[tom@CentOS tom\_repo]$ git remote add origin gituser@git.server.com:project.git**

**[tom@CentOS tom\_repo]$ git push origin master**

Now, the changes are successfully committed to the remote repository.

Git - Clone Operation

We have a bare repository on the Git server and Tom also pushed his first version. Now, Jerry can view his changes. The Clone operation creates an instance of the remote repository.

Jerry creates a new directory in his home directory and performs the clone operation.

**[jerry@CentOS ~]$ mkdir jerry\_repo**

**[jerry@CentOS ~]$ cd jerry\_repo/**

**[jerry@CentOS jerry\_repo]$ git clone** [**gituser@git.server.com:project.git**](mailto:gituser@git.server.com:project.git)

Git - Perform Changes

Git add operation adds file to the staging area.

**[jerry@CentOS project]$ git status -s**

**?? string**

**?? string.c**

**[jerry@CentOS project]$ git add string.c**

Git is showing a question mark before file names. Obviously, these files are not a part of Git, and that is why Git does not know what to do with these files. That is why, Git is showing a question mark before file names.

Jerry has added the file to the stash area, git status command will show files present in the staging area.

**[jerry@CentOS project]$ git status -s**

**A string.c**

**?? string**

To commit the changes, he used the git commit command followed by –m option. If we omit –m option. Git will open a text editor where we can write multiline commit message.

After commit to view log details, he runs the git log command. It will display the information of all the commits with their commit ID, commit author, commit date and **SHA-1** hash of commit.

[jerry@CentOS project]$ git log

Git - Review Changes

Jerry realizes that the string length cannot be negative, that’s why he decides to change the return type of my\_strlen function.

Jerry uses the **git log** command to view log details.

[**jerry@CentOS project]$ git log**

**commit cbe1249b140dad24b2c35b15cc7e26a6f02d2277**

**Author: Jerry Mouse <jerry@tutorialspoint.com>**

**Date: Wed Sep 11 08:05:26 2013 +0530**

**Implemented my\_strlen function**

Jerry uses the **git show** command to view the commit details. The git show command takes **SHA-1** commit ID as a parameter.

[jerry@CentOS project]$ git show cbe1249b140dad24b2c35b15cc7e26a6f02d2277

The above command will produce the following result −

**commit cbe1249b140dad24b2c35b15cc7e26a6f02d2277**

**Author: Jerry Mouse <jerry@tutorialspoint.com>**

**Date: Wed Sep 11 08:05:26 2013 +0530**

**Implemented my\_strlen function**

**diff --git a/string.c b/string.c**

**new file mode 100644**

**index 0000000..187afb9**

**--- /dev/null**

**+++ b/string.c**

**@@ -0,0 +1,24 @@**

**+#include <stdio.h>**

**+**

**+int my\_strlen(char \*s)**

**+{**

**+**

**char \*p = s;**

**+**

**+**

**while (\*p)**

**+ ++p;**

**+ return (p -s );**

**+**

**}**

**+**

He changes the return type of the function from int to size\_t. After testing the code, he reviews his changes by running the **git diff** command.

[jerry@CentOS project]$ git diff

The above command will produce the following result −

**diff --git a/string.c b/string.c**

**index 187afb9..7da2992 100644**

**--- a/string.c**

**+++ b/string.c**

**@@ -1,6 +1,6 @@**

**#include <stdio.h>**

**-int my\_strlen(char \*s)**

**+size\_t my\_strlen(char \*s)**

**{**

**char \*p = s;**

**@@ -18,7 +18,7 @@ int main(void)**

**};**

**for (i = 0; i < 2; ++i)**

**{**

**- printf("string lenght of %s = %d\n", s[i], my\_strlen(s[i]));**

**+ printf("string lenght of %s = %lu\n", s[i], my\_strlen(s[i]));**

**return 0;**

**}**

Git diff shows **'+'** sign before lines, which are newly added and **'−'** for deleted lines.

Git - Commit Changes

**git amend** operation will help. The amend operation changes the last commit including your commit message; it creates a new commit ID.

**[jerry@CentOS project]$ git commit --amend -m 'Changed return type of my\_strlen to size\_t'**

**[master d1e19d3] Changed return type of my\_strlen to size\_t**

**1 files changed, 24 insertions(+), 0 deletions(-)**

**create mode 100644 string.c**

Now, git log will show new commit message with new commit ID −

[jerry@CentOS project]$ git log

The above command will produce the following result.

**commit d1e19d316224cddc437e3ed34ec3c931ad803958**

**Author: Jerry Mouse <jerry@tutorialspoint.com>**

**Date: Wed Sep 11 08:05:26 2013 +0530**

**Changed return type of my\_strlen to size\_t**

**commit 19ae20683fc460db7d127cf201a1429523b0e319**

**Author: Tom Cat <tom@tutorialspoint.com>**

**Date: Wed Sep 11 07:32:56 2013 +0530**

**Initial commit**

Git - Push Operation

After modifying last commit by using the amend operation and the changes are ready to be pushed. The Push operation stores data permanently to the Git repository. After a successful push operation, other developers can see changes.

git log command is executed to view the commit details.

[jerry@CentOS project]$ git log

The above command will produce the following result:

**commit d1e19d316224cddc437e3ed34ec3c931ad803958**

**Author: Jerry Mouse <jerry@tutorialspoint.com>**

**Date: Wed Sep 11 08:05:26 2013 +0530**

**Changed return type of my\_strlen to size\_t**

Before push operation, he wants to review his changes, so he uses the **git show**command to review his changes.

**[jerry@CentOS project]$ git show d1e19d316224cddc437e3ed34ec3c931ad803958**

The above command will produces the result:

the changes are ready to be pushed.

[jerry@CentOS project]$ git push origin master

The above command will produce the following result:

Counting objects: 4, done.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 517 bytes, done.

Total 3 (delta 0), reused 0 (delta 0)

To gituser@git.server.com:project.git

19ae206..d1e19d3 master −> master

Jerry’s changes have been successfully pushed to the repository; now other developers can view his changes by performing clone or update operation.

Git - Update Operation

#### Modify Existing Function

Tom performs the clone operation and finds a new file string.c. He wants to know who added this file to the repository and for what purpose, so, he executes the **git log** command.

[tom@CentOS ~]$ git clone gituser@git.server.com:project.git

The above command will produce the following result −

Initialized empty Git repository in /home/tom/project/.git/

remote: Counting objects: 6, done.

remote: Compressing objects: 100% (4/4), done.

Receiving objects: 100% (6/6), 726 bytes, done.

remote: Total 6 (delta 0), reused 0 (delta 0)

The Clone operation will create a new directory inside the current working directory. He changes the directory to newly created directory and executes the **git log** command.

[tom@CentOS ~]$ cd project/

[tom@CentOS project]$ git log

The above command will produce the following result −

commit d1e19d316224cddc437e3ed34ec3c931ad803958

Author: Jerry Mouse [jerry@tutorialspoint.com](mailto:jerry@tutorialspoint.com)

Date: Wed Sep 11 08:05:26 2013 +0530

Changed return type of my\_strlen to size\_t

commit 19ae20683fc460db7d127cf201a1429523b0e319

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit

After observing the log, he realizes that the file string.c was added by Jerry to implement basic string operations. He is curious about Jerry’s code. So he opens string.c in text editor and immediately finds a bug. In my\_strlen function, Jerry is not using a constant pointer. So, he decides to modify Jerry’s code. After modification, the code looks as follows −

[tom@CentOS project]$ git diff

The above command will produce the following result −

diff --git a/string.c b/string.c

index 7da2992..32489eb 100644

--- a/string.c

+++ b/string.c

@@ -1,8 +1,8 @@

#include <stdio.h>

-size\_t my\_strlen(char \*s)

+size\_t my\_strlen(const char \*s)

{

- char \*p = s;

+ const char \*p = s;

while (\*p)

++p;

}

After testing, he commits his change.

[tom@CentOS project]$ git status -s

M string.c

?? string

[tom@CentOS project]$ git add string.c

[tom@CentOS project]$ git commit -m 'Changed char pointer to const char pointer'

[master cea2c00] Changed char pointer to const char pointer

1 files changed, 2 insertions(+), 2 deletions(-)

[tom@CentOS project]$ git log

The above command will produce the following result −

commit cea2c000f53ba99508c5959e3e12fff493b

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 08:32:07 2013 +0530

Changed char pointer to const char pointer

commit d1e19d316224cddc437e3ed34ec3c931ad803958

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 08:05:26 2013 +0530

Changed return type of my\_strlen to size\_t

commit 19ae20683fc460db7d127cf201a1429523b0e319

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit

Tom uses git push command to push his changes.

[tom@CentOS project]$ git push origin master

The above command will produce the following result −

Counting objects: 5, done.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 336 bytes, done.

Total 3 (delta 1), reused 0 (delta 0)

To gituser@git.server.com:project.git

d1e19d3..cea2c00 master −> master

#### Add New Function

Meanwhile, Jerry decides to implement **string compare** functionality. So he modifies string.c. After modification, the file looks as follows −

[jerry@CentOS project]$ git diff

The above command will produce the following result −

index 7da2992..bc864ed 100644

--- a/string.c

+++ b/string.c

30Git Tutorials

@@ -9,9 +9,20 @@ size\_t my\_strlen(char \*s)

After testing, he is ready to push his change.

[jerry@CentOS project]$ git status -s

M string.c

?? string

[jerry@CentOS project]$ git add string.c

[jerry@CentOS project]$ git commit -m "Added my\_strcpy function"

[master e944e5a] Added my\_strcpy function

1 files changed, 13 insertions(+), 0 deletions(-)

Before push operation, he verifies commit by viewing log messages.

[jerry@CentOS project]$ git log

The above command will produce the following result −

commit e944e5aab74b26e7447d3281b225309e4e59efcd

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 08:41:42 2013 +0530

Added my\_strcpy function

commit d1e19d316224cddc437e3ed34ec3c931ad803958

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 08:05:26 2013 +0530

Changed return type of my\_strlen to size\_t

commit 19ae20683fc460db7d127cf201a1429523b0e319

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit

Jerry is happy with the changes and he wants to push his changes.

[jerry@CentOS project]$ git push origin master

The above command will produce the following result −

To gituser@git.server.com:project.git

! [rejected]

master −> master (non-fast-forward)

error: failed to push some refs to 'gituser@git.server.com:project.git'

To prevent you from losing history, non-fast-forward updates were rejected

Merge the remote changes before pushing again. See the 'Note about

fast-forwards' section of 'git push --help' for details.

But Git is not allowing Jerry to push his changes. Because Git identified that remote repository and Jerry’s local repository are not in sync. Because of this, he can lose the history of the project. To avoid this mess, Git failed this operation. Now, Jerry has to first update the local repository and only thereafter, he can push his own changes.

#### Fetch Latest Changes

Jerry executes the git pull command to synchronize his local repository with the remote one.

[jerry@CentOS project]$ git pull

The above command will produce the following result −

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 3 (delta 1), reused 0 (delta 0)

Unpacking objects: 100% (3/3), done.

From git.server.com:project

d1e19d3..cea2c00 master −> origin/master

First, rewinding head to replay your work on top of it...

Applying: Added my\_strcpy function

After pull operation, Jerry checks the log messages and finds the details of Tom’s commit with commit ID **cea2c000f53ba99508c5959e3e12fff493ba6f69**

[jerry@CentOS project]$ git log

The above command will produce the following result −

commit e86f0621c2a3f68190bba633a9fe6c57c94f8e4f

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 08:41:42 2013 +0530

Added my\_strcpy function

commit cea2c000f53ba99508c5959e3e12fff493ba6f69

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 08:32:07 2013 +0530

Changed char pointer to const char pointer

commit d1e19d316224cddc437e3ed34ec3c931ad803958

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 08:05:26 2013 +0530

Changed return type of my\_strlen to size\_t

commit 19ae20683fc460db7d127cf201a1429523b0e319

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit

Now, Jerry’s local repository is fully synchronized with the remote repository. So he can safely push his changes.

[jerry@CentOS project]$ git push origin master

The above command will produce the following result −

Counting objects: 5, done.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 455 bytes, done.

Total 3 (delta 1), reused 0 (delta 0)

To gituser@git.server.com:project.git

cea2c00..e86f062 master −> master

Git - Stash Operation

In Git, the stash operation takes your modified tracked files, stages changes, and saves them on a stack of unfinished changes that you can reapply at any time.

[jerry@CentOS project]$ git status -s

M string.c

?? string

Now, you want to switch branches for customer escalation, but you don’t want to commit what you’ve been working on yet; so you’ll stash the changes. To push a new stash onto your stack, run the **git stash** command.

[jerry@CentOS project]$ git stash

Saved working directory and index state WIP on master: e86f062 Added my\_strcpy function

HEAD is now at e86f062 Added my\_strcpy function

Now, your working directory is clean and all the changes are saved on a stack. Let us verify it with the **git status** command.

[jerry@CentOS project]$ git status -s

?? string

Now you can safely switch the branch and work elsewhere. We can view a list of stashed changes by using the **git stash list** command.

[jerry@CentOS project]$ git stash list

stash@{0}: WIP on master: e86f062 Added my\_strcpy function

Suppose you have resolved the customer escalation and you are back on your new feature looking for your half-done code, just execute the **git stash pop** command, to remove the changes from the stack and place them in the current working directory.

[jerry@CentOS project]$ git status -s

?? string

[jerry@CentOS project]$ git stash pop

The above command will produce the following result:

# On branch master

# Changed but not updated:

# (use "git add ..." to update what will be committed)

# (use "git checkout -- ..." to discard changes in working directory)

#

modified: string.c

#

# Untracked files:

# (use "git add ..." to include in what will be committed)

#

string

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

Dropped refs/stash@{0} (36f79dfedae4ac20e2e8558830154bd6315e72d4)

[jerry@CentOS project]$ git status -s

M string.c

?? string

Git - Move Operation

The move operation moves a directory or a file from one location to another. To move the source code into **src** directory. The modified directory structure will appear as follows −

[tom@CentOS project]$ pwd

/home/tom/project

[tom@CentOS project]$ ls

README string string.c

[tom@CentOS project]$ mkdir src

[tom@CentOS project]$ git mv string.c src/

[tom@CentOS project]$ git status -s

R string.c −> src/string.c

?? string

To make these changes permanent, we have to push the modified directory structure to the remote repository so that other developers can see this.

[tom@CentOS project]$ git commit -m "Modified directory structure"

[master 7d9ea97] Modified directory structure

1 files changed, 0 insertions(+), 0 deletions(-)

rename string.c => src/string.c (100%)

[tom@CentOS project]$ git push origin master

Counting objects: 4, done.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 320 bytes, done.

Total 3 (delta 0), reused 0 (delta 0)

To gituser@git.server.com:project.git

e86f062..7d9ea97 master −> master

In Jerry’s local repository, before pull operation, it will show the old directory structure.

[jerry@CentOS project]$ pwd

/home/jerry/jerry\_repo/project

[jerry@CentOS project]$ ls

README string string.c

But after the pull operation, the directory structure will get updated. Now, Jerry can see the **src** directory and the file present inside that directory.

[jerry@CentOS project]$ git pull

remote: Counting objects: 4, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 0 (delta 0)

Unpacking objects: 100% (3/3), done.

From git.server.com:project

e86f062..7d9ea97 master −> origin/master

First, rewinding head to replay your work on top of it...

Fast-forwarded master to 7d9ea97683da90bcdb87c28ec9b4f64160673c8a.

[jerry@CentOS project]$ ls

README src string

[jerry@CentOS project]$ ls src/

string.c

Git - Managing Branches

Branch operation allows creating another line of development. We can use this operation to fork off the development process into two different directions. For example, we released a product for 6.0 version and we might want to create a branch so that the development of 7.0 features can be kept separate from 6.0 bug fixes.

#### Create Branch

Tom creates a new branch using the git branch <branch name> command. We can create a new branch from an existing one. We can use a specific commit or tag as the starting point. If any specific commit ID is not provided, then the branch will be created with HEAD as its starting point.

[jerry@CentOS src]$ git branch new\_branch

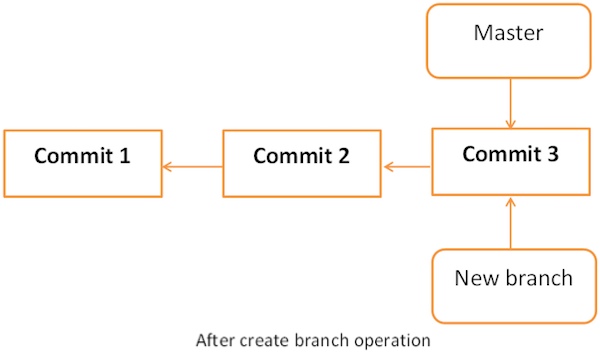
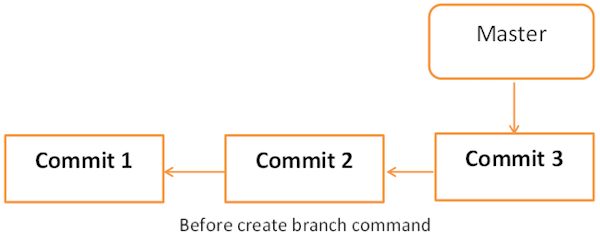
[jerry@CentOS src]$ git branch

\* master

new\_branch

A new branch is created; Tom used the git branch command to list the available branches. Git shows an asterisk mark before currently checked out branch.

The pictorial representation of create branch operation is shown below −



#### Switch between Branches

Jerry uses the git checkout command to switch between branches.

[jerry@CentOS src]$ git checkout new\_branch

Switched to branch 'new\_branch'

[jerry@CentOS src]$ git branch

master

\* new\_branch

#### Shortcut to Create and Switch Branch

In the above example, we have used two commands to create and switch branches, respectively. Git provides **–b** option with the checkout command; this operation creates a new branch and immediately switches to the new branch.

[jerry@CentOS src]$ git checkout -b test\_branch

Switched to a new branch 'test\_branch'

[jerry@CentOS src]$ git branch

master

new\_branch

\* test\_branch

#### Delete a Branch

A branch can be deleted by providing –D option with git branch command. But before deleting the existing branch, switch to the other branch.

Jerry is currently on **test\_branch** and he wants to remove that branch. So he switches branch and deletes branch as shown below.

[jerry@CentOS src]$ git branch

master

new\_branch

\* test\_branch

[jerry@CentOS src]$ git checkout master

Switched to branch 'master'

[jerry@CentOS src]$ git branch -D test\_branch

Deleted branch test\_branch (was 5776472).

Now, Git will show only two branches.

[jerry@CentOS src]$ git branch

\* master

new\_branch

#### Rename a Branch

Jerry decides to add support for wide characters in his string operations project. He has already created a new branch, but the branch name is not appropriate. So he changes the branch name by using **–m** option followed by the **old branch name** and the **new branch name**.

[jerry@CentOS src]$ git branch

\* master

new\_branch

[jerry@CentOS src]$ git branch -m new\_branch wchar\_support

Now, the git branch command will show the new branch name.

[jerry@CentOS src]$ git branch

\* master

wchar\_support

#### Merge Two Branches

Jerry implements a function to return the string length of wide character string. New the code will appear as follows −

[jerry@CentOS src]$ git branch

master

\* wchar\_support

[jerry@CentOS src]$ pwd

/home/jerry/jerry\_repo/project/src

[jerry@CentOS src]$ git diff

After testing, he commits and pushes his changes to the new branch.

[jerry@CentOS src]$ git status -s

M string\_operations.c

?? string\_operations

[jerry@CentOS src]$ git add string\_operations.c

[jerry@CentOS src]$ git commit -m 'Added w\_strlen function to return string lenght of wchar\_t

string'

[wchar\_support 64192f9] Added w\_strlen function to return string lenght of wchar\_t string

1 files changed, 10 insertions(+), 0 deletions(-)

Note that Jerry is pushing these changes to the new branch, which is why he used the branch name **wchar\_support** instead of **master** branch.

[jerry@CentOS src]$ git push origin wchar\_support  **<−−− Observer branch\_name**

The above command will produce the following result.

Counting objects: 7, done.

Compressing objects: 100% (4/4), done.

Writing objects: 100% (4/4), 507 bytes, done.

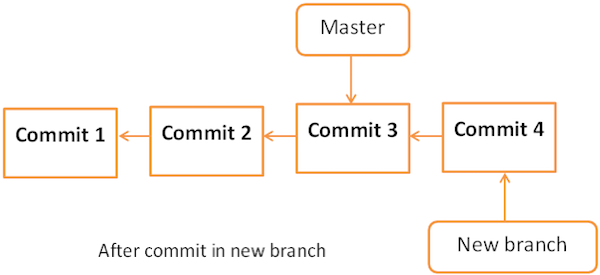
Total 4 (delta 1), reused 0 (delta 0)

To gituser@git.server.com:project.git

\* [new branch]

wchar\_support -> wchar\_support

After committing the changes, the new branch will appear as follows −



Tom is curious about what Jerry is doing in his private branch and he checks the log from the **wchar\_support** branch.

[tom@CentOS src]$ pwd

/home/tom/top\_repo/project/src

[tom@CentOS src]$ git log origin/wchar\_support -2

The above command will produce the following result.

commit 64192f91d7cc2bcdf3bf946dd33ece63b74184a3

Author: Jerry Mouse <jerry@tutorialspoint.com>

Date: Wed Sep 11 16:10:06 2013 +0530

Added w\_strlen function to return string lenght of wchar\_t string

commit 577647211ed44fe2ae479427a0668a4f12ed71a1

Author: Tom Cat <tom@tutorialspoint.com>

Date: Wed Sep 11 10:21:20 2013 +0530

Removed executable binary

By viewing commit messages, Tom realizes that Jerry implemented the strlen function for wide character and he wants the same functionality in the master branch. Instead of re-implementing, he decides to take Jerry’s code by merging his branch with the master branch.

[tom@CentOS project]$ git branch

\* master

[tom@CentOS project]$ pwd

/home/tom/top\_repo/project

[tom@CentOS project]$ git merge origin/wchar\_support

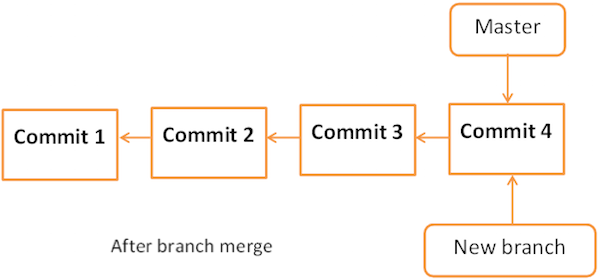
Updating 5776472..64192f9

Fast-forward

src/string\_operations.c | 10 ++++++++++

1 files changed, 10 insertions(+), 0 deletions(-)

After the merge operation, the master branch will appear as follows −



Now, the branch **wchar\_support** has been merged with the master branch. We can verify it by viewing the commit message or by viewing the modifications done into the string\_operation.c file.

[tom@CentOS project]$ cd src/

[tom@CentOS src]$ git log -1

commit 64192f91d7cc2bcdf3bf946dd33ece63b74184a3

Author: Jerry Mouse

Date: Wed Sep 11 16:10:06 2013 +0530

Added w\_strlen function to return string lenght of wchar\_t string

[tom@CentOS src]$ head -12 string\_operations.c

The above command will produce the following result.

#include <stdio.h>

#include <wchar.h>

size\_t w\_strlen(const wchar\_t \*s)

{

const wchar\_t \*p = s;

while (\*p)

++p;

return (p - s);

}

After testing, he pushes his code changes to the master branch.

[tom@CentOS src]$ git push origin master

Total 0 (delta 0), reused 0 (delta 0)

To gituser@git.server.com:project.git

5776472..64192f9 master −> master

#### Rebase Branches

The Git rebase command is a branch merge command, but the difference is that it modifies the order of commits.

The Git merge command tries to put the commits from other branches on top of the HEAD of the current local branch. For example, your local branch has commits A−>B−>C−>D and the merge branch has commits A−>B−>X−>Y, then git merge will convert the current local branch to something like A−>B−>C−>D−>X−>Y

The Git rebase command tries to find out the common ancestor between the current local branch and the merge branch. It then pushes the commits to the local branch by modifying the order of commits in the current local branch. For example, if your local branch has commits A−>B−>C−>D and the merge branch has commits A−>B−>X−>Y, then Git rebase will convert the current local branch to something like A−>B−>X−>Y−>C−>D.

When multiple developers work on a single remote repository, you cannot modify the order of the commits in the remote repository. In this situation, you can use rebase operation to put your local commits on top of the remote repository commits and you can push these changes.

**GitHub** is a web-based hosting service for software development projects that uses the Git revision control system. It also has their standard GUI application available for download (Windows, Mac, GNU/ Linux) directly from the service's website. But in this session, we will see only CLI part.

Git - Online Repositories

#### Create GitHub Repository

Go to [github.com](https://github.com/). If you already have the **GitHub** account, then login using that account or create a new one. Follow the steps from [github.com](https://github.com/) website to create a new repository.

#### Push Operation

Tom decides to use the **GitHub** server. To start a new project, he creates a new directory and one file inside that.

[tom@CentOS]$ mkdir github\_repo

[tom@CentOS]$ cd github\_repo/

[tom@CentOS]$ vi hello.c

[tom@CentOS]$ make hello

cc hello.c -o hello

[tom@CentOS]$ ./hello

The above command will produce the following result:

Hello, World !!!

After verifying his code, he initializes the directory with the git init command and commits his changes locally.

[tom@CentOS]$ git init

Initialized empty Git repository in /home/tom/github\_repo/.git/

[tom@CentOS]$ git status -s

?? hello

?? hello.c

[tom@CentOS]$ git add hello.c

[tom@CentOS]$ git status -s

A hello.c

?? hello

[tom@CentOS]$ git commit -m 'Initial commit'

After that, he adds the **GitHub** repository URL as a remote origin and pushes his changes to the remote repository.

[tom@CentOS]$ git remote add origin https://github.com/kangralkar/testing\_repo.git

[tom@CentOS]$ git push -u origin master

Push operation will ask for **GitHub** user name and password. After successful authentication, the operation will succeed.

The above command will produce the following result:

Username for 'https://github.com': kangralkar

Password for 'https://kangralkar@github.com':

Counting objects: 3, done.

Writing objects: 100% (3/3), 214 bytes, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/kangralkar/test\_repo.git

\* [new branch] master −> master

Branch master set up to track remote branch master from origin.

From now, Tom can push any changes to the **GitHub** repository. He can use all the commands discussed in this chapter with the **GitHub** repository.

#### Pull Operation

Tom successfully pushed all his changes to the **GitHub** repository. Now, other developers can view these changes by performing clone operation or updating their local repository.

Jerry creates a new directory in his home directory and clones the **GitHub** repository by using the git clone command.

[jerry@CentOS]$ pwd

/home/jerry

[jerry@CentOS]$ mkdir jerry\_repo

[jerry@CentOS]$ git clone https://github.com/kangralkar/test\_repo.git

The above command produces the following result:

Cloning into 'test\_repo'...

remote: Counting objects: 3, done.

remote: Total 3 (delta 0), reused 3 (delta 0)

Unpacking objects: 100% (3/3), done.

He verifies the directory contents by executing the ls command.

[jerry@CentOS]$ ls

test\_repo

[jerry@CentOS]$ ls test\_repo/

hello.c

**GIT Directory Structure Tutorial**

<https://www.siteground.com/tutorials/git/directory-structure/>

Learn how GIT structures the repository content

A **.git** directory has a structure similar to the following one:

* **objects/ folder**  
  In this directory the data of your Git objects is stored – all the contents of the files you have ever checked in, your commits, trees and tag objects.
  + - **objects/[0-9a-f][0-9a-f] folders**

A newly created object is stored in its own file. The objects are placed over 256 subdirectories using the first two characters of the SHA1 object name to keep the number of directory entries in objects itself to a manageable number. Objects found here are often called unpacked or loose objects.

* + - **objects/pack folder**

Files that store many object in compressed form, along with index files to allow them to be randomly accessed are found in this directory.

* + - **objects/info folder**

Additional information about the object stored is placed in this directory.

* **refs folder**  
  References are stored in subdirectories of this directory. The git prune command knows to preserve objects reachable from refs found in this directory and its subdirectories.
  + - **refs/heads/ folder**

Contains commit objects.

* + - **refs/tags/ folder**

Contains any object name.

* + - **refs/remotes/ folder**

Contains commit objects of branches copied from a remote repository.

* **packed-refs file**

The file consists of packed heads and tags. It is useful for an efficient repository access.

* **HEAD file**

This file holds a reference to the branch you are currently on. This tells Git what to use as the parent of your next commit

* **config file**

This is the main Git configuration file. It keeps specific Git options for your project, such as your remotes, push configurations, tracking branches and more. Your configuration will be loaded first from this file, then from a *~/.gitconfig*file and then from an */etc/gitconfig* file, if they exist.

* **branches**

A deprecated way to store shorthands to be used to specify a URL to *git fetch*, *git pull* and *git push*. This mechanism is legacy and not likely to be found in modern repositories.

* **hooks folder**

This directory contains shell scripts that are invoked after the corresponding Git commands. For example, after you run a commit, Git will try to execute the post-commit script.

* **index file**

The GIT index is used as a staging area between your working directory and your repository. You can use the index to build up a set of changes that you want to commit together. When you create a commit, what is committed is what is currently in the index, not what is in your working directory. It is a binary file containing a sorted list of path names, each with permissions and the SHA-1 of a blob object.

* **info folder**

Additional information about the repository is recorded in this directory.

* **remotes folder**

This folder contains shorthands for URL and default refnames for use when interacting with remote repositories via *git fetch*, *git pull* and *git push* commands. This mechanism is legacy and not likely to be found in modern repositories.

* **logs folder**  
  Stores the changes made to refs in repository.
  + - **logs/refs/heads/** folder

Records all changes made to the different branch tips

* + - **logs/refs/tags/** folder

Records all changes made to the different tags

* **modules folder**

Contains the git-repositories of the submodules.

* **worktrees folder**

Contains administrative data for linked working trees. Each subdirectory contains the working tree-related part of a linked working tree.