Git Tutorial

GIT TUTORIAL

Simply Easy Learning by tutorialspoint.com

tutorialspoint.com

ABOUT THE TUTORIAL

Git Tutorial

Git is a distributed revision control and source code management system with an emphasis on speed. Git was initially designed and developed by Linus Torvalds for Linux kernel development. Git is free software distributed under the terms of the GNU General Public License version 2.

This tutorial will teach you how to use Git in your day-2-day life for project version control in a distributed environment while working on web based and non web based applications development.

Audience

This tutorial has been prepared for the beginners to help them understand basic functionality of Git version control system. After completing this tutorial you will find yourself at a moderate level of expertise in using Git version control system from where you can take yourself to next levels.

Prerequisites

We assume you are going to use Git to handle all levels of Java and non Java projects. So it will be good if you have knowledge of software development life cycle and knowledge of developing web based and non web based applications.

Copyright & Disclaimer Notice

Table of Content

CHAPTER

1

Basic Concepts

Version Control System (VCS)

Version Control System (VCS) is software that helps Software developers to work together and maintains the complete history of their work.

Following are goals of VCS

1. Allow developers to work simultaneously.
2. Do not overwrite each other's changes.
3. Maintain history of every version of everything.

Following are goals of VCS

1. Centralized version control system(CVCS) and
2. Distributed/Decentralized version control system(DVCS)

In this tutorials session we will concentrate only on Distributed version control system and especially Git. Git falls under distributed version control system.

Distributed Version Control System (DVCS)

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

DVCS clients not only checkout the latest snapshot of the directory but they also fully mirror the repository. If sever goes down, then repository from any client can be copied back to server to restore it. Every checkout is full backup of the repository. Git does not rely on central server 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 latest changes.

Advantages of Git

1. **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 propriety projects without paying single penny. As it is open source you can download its source code and also perform changes according to your requirements.

2. **Fast and small**

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

3. **Implicit backup**

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

4. **Security**

Git uses 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. Meaning that, it is impossible to change file, date, commit message and any other data from Git database without knowing Git.

5. **No need of powerful hardware**

In case of CVCS, the central server needs to be powerful enough to serve request of the entire team. For smaller team it's not an issue but as team size grows, the hardware limitation of the server can be a performance bottleneck. In case of DVCS developers don't interact with the server unless they need to push or pull changes. All the heavy lifting happens on the client side so the server hardware can be very simple indeed.

6. **Easier branching**

CVCS uses cheap copy mechanism, means if we create new branch it will copy all code to the new branch so it's 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 few seconds to create, delete and merge branches.

DVCS Terminologies

LOCAL REPOSITORY

Every VCS tool provides private workplace as a working copy. Developer does changes in his private workplace and after commit these changes becomes part of the repository. Git takes this one step further by providing them a private copy of the whole repository. User can perform many operations with this repository like add file, remove file, rename file, move file, commit changes and many more.

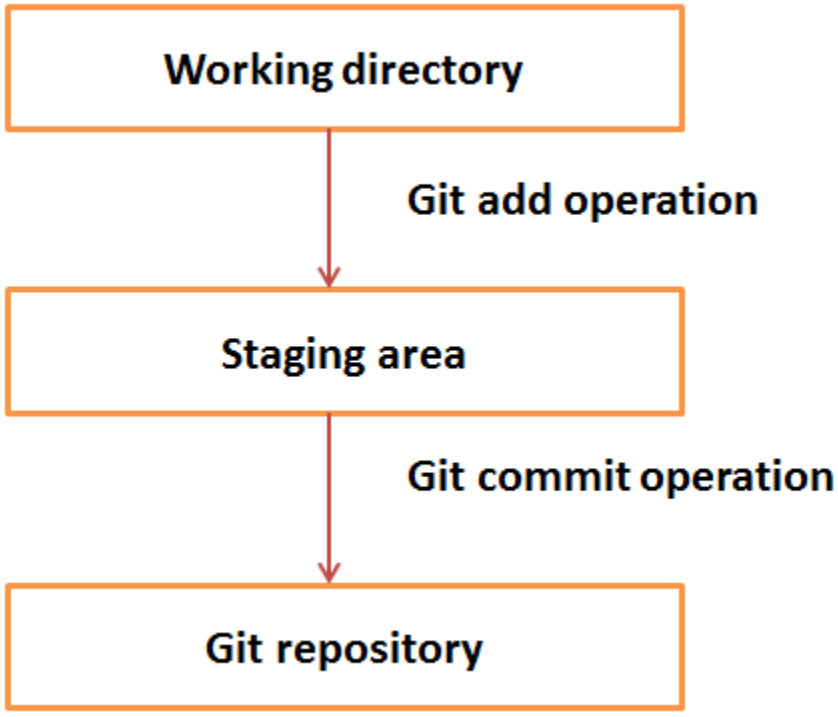
WORKING DIRECTORY AND STAGING AREA OR INDEX

The working directory is the place where files are checked out. In other CVCS; developer generally does modification and commits his changes directly to the repository. But Git uses different strategy. Git doesn't track

each and every modified file. Whenever you do commit operation, Git looks for the files present in staging area. Only files present in the staging area is considered for commit and not all modified files.

Let us see basic workflow of the Git.

Step 1: You modify file from working directory.
Step 2: You add these files to the staging area
Step 3: You perform commit operation. That move files from staging area. After push operation it store changes permanently to the Git repository



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 staging area and do commit. After 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 file is represented by blob. A blob holds file data but doesn't contain any metadata about file. It is a binary file, in Git database it is named as SHA1 hash of that file. In Git, files are not addressed by name. 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 is also named as the **SHA1** hash of the tree object.

COMMITS

Commit holds the current state of the repository. A commit also named by **SHA1** hash. You can consider commit object as a node of the linked list. Every commit object has a pointer to the parent commit object. From 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, that means the particular commit is 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 to work on new feature a branch is created. Once feature is completed; it is merged back with 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 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 tags are immutable. Means tag is a branch which nobody intends to modify. Once tag is created for particular commit, even you create a new commit, it will not be updated. Usually developer creates tags for product releases.

CLONE

Clone operation creates the instance of the repository. Clone operation not only checkouts the working copy but it also mirrors the complete repository. User 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 copy changes from a remote repository instance to local one. The pull operation is used for synchronization between two repository instances. This is same as update operation in Subversion.

PUSH

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

HEAD

HEAD is pointer which always points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with 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.

URL

URL represents the location of the Git repository. Git URL is stored in config file.

```
[tom@CentOS tom_repo]$ pwd
/home/tom/tom_repo

[tom@CentOS tom_repo]$ cat .git/config
[core]
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
[remote "origin"]
url = gituser@git.server.com:project.git
fetch = +refs/heads/*:refs/remotes/origin/*
```

CHAPTER

2

Environment Setup

Before you can use Git, you have to install and do some basic configuration changes. Below are steps to install Git client on Ubuntu and Centos Linux.

Installation of Git client

If you are using Debian base GNU/Linux distribution then apt-get command will do needful.

```
[ubuntu ~]$ sudo apt-get install git-core
[sudo] password for ubuntu:

[ubuntu ~]$ git --version
git version 1.8.1.2
```

And if you are using RPM based GNU/Linux distribution the use yum command as given

```
[CentOS ~]$
su -
Password:

[CentOS ~]# yum -y install git-core

[CentOS ~]# git --version
git version 1.7.1
```

Customize Git environment

Git provides git config tool which allows you to set configuration variable. 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 setup system wide configuration. Git stores theses values is in the **/etc/gitconfig** file, which contains the configuration for every user and repository on the system. To set these values you must have root rights and use the **--system** option.

When the above code is compiled and executed, it produces 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"
```

AVOID MERGE COMMITS FOR PULLING

You pull latest changes from a remote repository and if these changes are divergent then, by default Git creates merge commits. We can avoid this via following settings.

```
jerry@CentOS project]$ git config --global branch.autosetuprebase always
```

COLOR HIGHLIGHTING

The following commands enable color highlighting for Git in the console.

```
[jerry@CentOS project]$ git config --global color.ui true

[jerry@CentOS project]$ git config --global color.status auto

[jerry@CentOS project]$ git config --global color.branch auto
```

SETTING DEFAULT EDITOR

By default Git uses the system default editor which is taken from the VISUAL or EDITOR environment variable. We can configure a different one by using git config.

```
[jerry@CentOS project]$ git config --global core.editor vim
```

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
```

Above command will produce following result.

```
user.name=Jerry Mouse
user.email=jerry@tutorialspoint.com
push.default=nothing
branch.autosetuprebase=always
color.ui=true
color.status=auto
color.branch=auto
core.editor=vim
merge.tool=vimdiff
```

CHAPTER

3

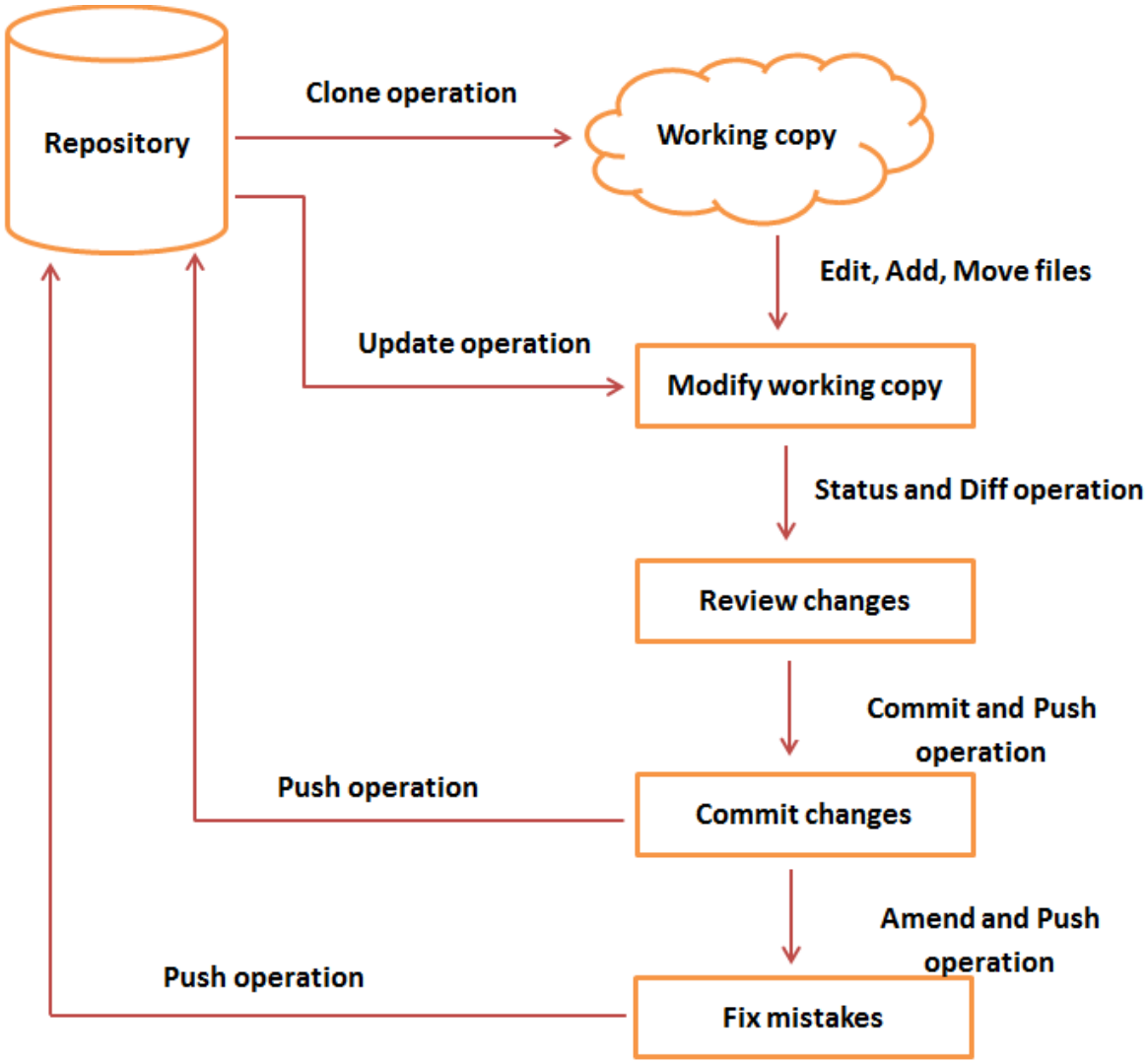
Life Cycle

In this chapter we will discuss life cycle of Git. In later chapters we will see Git command for each operation.

General work flow is like this:

1. You clone Git repository as a working copy.
2. You modify working copy by adding / editing files.
3. If necessary you also update working copy by taking other developers changes.
4. You review changes before commit.
5. You commit changes. If everything was fine, then you push changes to the repository.
6. After committing if you realize something was wrong, then you correct the last commit and push changes to the repository.

Below is pictorial representation of work flow.


Repository
Clone operation
Working copy
Edit, Add, Move files
Update operation
Modify working copy
Status and Diff operation
Review changes
Commit and Push operation
Push operation
Commit changes
Amend and Push operation
Push operation
Fix mistakes

CHAPTER

4

Create Operation

In this chapter we will see how to create a remote Git repository, from now we will refer it as Git Server. We need a Git server to allow team collaboration.

Create new user

```
# add new group
[root@CentOS ~]# groupadd dev

# add new user
[root@CentOS ~]# useradd -G devs -d /home/gituser -m -s /bin/bash gituser

# change password
[root@CentOS ~]# passwd gituser
```

Above command will produce following result.

```
Changing password for user gituser.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
```

Create a bare repository

Let us initialize a new repository by using init command followed by **--bare** option. It initializes repository without working directory. By convention 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
```

Generate public/private RSA key pair

Let us walk through the process of configuring Git server, **ssh-keygen** utility generates public/private RSA key pair, we will use these keys for user authentication.

Open a terminal and enter following command and just press enter for each input. After successful completion it will create **.ssh** directory inside home directory.

```
tom@CentOS ~]$ pwd
/home/tom

[tom@CentOS ~]$ ssh-keygen
```

Above command will produce following result.

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/tom/.ssh/id_rsa): Press Enter Only
Created directory '/home/tom/.ssh'.
Enter passphrase (empty for no passphrase): ----------------> Press Enter Only
Enter same passphrase again: -----------------------------> Press Enter Only
Your identification has been saved in /home/tom/.ssh/id_rsa.
Your public key has been saved in /home/tom/.ssh/id_rsa.pub.
The key fingerprint is:
df:93:8c:a1:b8:b7:67:69:3a:1f:65:e8:0e:e9:25:a1 tom@CentOS
The key's randomart image is:
+--[ RSA 2048]----+
| |
| |
| |
|
.
|
| Soo |
| o*B. |
| E = *.= |
| oo==. . |
| ..+Oo
|
+-----------------+
```

ssh-keygen has generated two keys first one is private(i.e. id_rsa) and another is public(i.e. id_rsa.pub).

Note: Never share your PRIVATE KEY with others.

Adding keys to authorized_keys

Suppose there are two developers working on a project namely Tom and Jerry. Both users has generated public key. Let us see how to use these keys for authentication.

Tom added his public key to server by using **ssh-copy-id** command as given below

```
[tom@CentOS ~]$ pwd
/home/tom

[tom@CentOS ~]$ ssh-copy-id -i ~/.ssh/id_rsa.pub gituser@git.server.com
```

Above command will produce following result.

```
gituser@git.server.com's password:
Now try logging into the machine, with "ssh 'gituser@git.server.com'", and check
in:
.ssh/authorized_keys
to make sure we haven't added extra keys that you weren't expecting.
```

Similarly Jerry also added his public key to server by using ssh-copy-id command.

```
[jerry@CentOS ~]$ pwd
/home/jerry

[jerry@CentOS ~]$ ssh-copy-id -i ~/.ssh/id_rsa gituser@git.server.com
```

Above command will produce following result.

```
gituser@git.server.com's password:
Now try logging into the machine, with "ssh 'gituser@git.server.com'", and check
in:
.ssh/authorized_keys
to make sure we haven't added extra keys that you weren't expecting.
```

Push changes to the repository

We have created bare repository on the server and allowed access for two users. From now Tom and Jerry can push their changes to the repository by adding it as and remote.

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

Tom creates a new directory, add READE file and commits his change as initial commit. After commit he verifies commit message by running git log command.

```
[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'
```

Above command will produce following result.

```
[master (root-commit) 19ae206] Initial commit
1 files changed, 1 insertions(+), 0 deletions(-)
```

```
create mode 100644 README
```

Tom checks log message by executing git log command.

```
[tom@CentOS tom_repo]$ git log
```

Above command will produce following result.

```
commit 19ae20683fc460db7d127cf201a1429523b0e319
Author: Tom Cat
Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit
```

Tom committed his changes to the local repository. Now it's time to push changes to the remote repository. But before that we have to add repository as a remote, this is one time operation. After this he can safely push changes to the remote repository.

Note: By default, Git pushes only to matching branches: For every branch that exits on the local side, the remote side is updated if a branch with the same name already exists there. In our tutorials every time I am pushing changes to **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
```

Above command will produce following result.

```
Counting objects: 3, done.
Writing objects: 100% (3/3), 242 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To gituser@git.server.com:project.git
* [new branch]
master -> master
```

Now changes are successfully committed to the remote repository.

CHAPTER

5

Clone Operation

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

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

```
[jerry@CentOS ~]$ mkdir jerry_repo

[jerry@CentOS ~]$ cd jerry_repo/

[jerry@CentOS jerry_repo]$ git clone gituser@git.server.com:project.git
```

Above command will produce following result.

```
Initialized empty Git repository in /home/jerry/jerry_repo/project/.git/
remote: Counting objects: 3, done.
Receiving objects: 100% (3/3), 241 bytes, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
```

Jerry changes directory to new local repository and lists directory contents.

```
[jerry@CentOS jerry_repo]$ cd project/

[jerry@CentOS jerry_repo]$ ls
README
```

CHAPTER

6

Perform Changes

Jerry cloned repository and he decides to implement basic string operations. So he creates string.c file. After adding contents string.c will look like this.

```
#include <stdio.h>

int my_strlen(char *s)
{
   char *p = s;

   while (*p)
      ++p;

   return (p - s);
}

int main(void)
{
   int i;
   char *s[] = {
      "Git tutorials",
      "Tutorials Point"
   };

   for (i = 0; i < 2; ++i)
      printf("string lenght of %s = %d\n", s[i], my_strlen(s[i]));

   return 0;
}
```

He compiled and tested his code everything is working fine. Now he can safely add these changes to the repository.

Git add operation add file to the staging area.

```
[jerry@CentOS project]$ git status -s
?? string
?? string.c

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

Git is showing question mark before file names. Obviously these files are not part of Git, that's why Git don't know what to with these files. That is why Git is showing question mark before file names.

Jerry has added 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 changes he used git commit command followed by –m option. If we omit –m option git will open text editor where we can write multiline commit message.

```
[jerry@CentOS project]$ git commit -m 'Implemented my_strlen function'
```

Above command will produce following result.

```
[master cbe1249] Implemented my_strlen function
1 files changed, 24 insertions(+), 0 deletions(-)
create mode 100644 string.c
```

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

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

Above command will produce following result.

```
commit cbe1249b140dad24b2c35b15cc7e26a6f02d2277
Author: Jerry Mouse <jerry@tutorialspoint.com>
Date: Wed Sep 11 08:05:26 2013 +0530

Implemented my_strlen function


commit 19ae20683fc460db7d127cf201a1429523b0e319
Author: Tom Cat <tom@tutorialspoint.com>
Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit
```

CHAPTER

7

Review Changes

But after viewing commit details Jerry realizes that string length cannot be negative, that's why he decides to change return type of my_strlen function.

Jerry uses git log command to view log details.

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

Above command will produce following result.

```
commit cbe1249b140dad24b2c35b15cc7e26a6f02d2277
Author: Jerry Mouse <jerry@tutorialspoint.com>
Date: Wed Sep 11 08:05:26 2013 +0530

Implemented my_strlen function
```

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

```
[jerry@CentOS project]$ git show cbe1249b140dad24b2c35b15cc7e26a6f02d2277
```

Above command will produce 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 return type of function from int to size_t. After testing code he reviews his changes by running git diff command.

```
[jerry@CentOS project]$ git diff
```

Above command will produce 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 shows **−**sign which are deleted.

CHAPTER

8

Commit Changes

Jerry has already committed changes and he wants to correct his last commit, in this case git amend operation will help. Amend operation changes last commit including your commit message, it creates new commit ID.

Before amend operation he checks the commit log.

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

Above command will produce following result.

```
commit cbe1249b140dad24b2c35b15cc7e26a6f02d2277
Author: Jerry Mouse <jerry@tutorialspoint.com>
Date: Wed Sep 11 08:05:26 2013 +0530

Implemented my_strlen function


commit 19ae20683fc460db7d127cf201a1429523b0e319
Author: Tom Cat <tom@tutorialspoint.com>
Date: Wed Sep 11 07:32:56 2013 +0530

Initial commit
```

Jerry commits new changes with -- amend operation and views the commit log.

```
[jerry@CentOS project]$ git status -s
M string.c
?? string

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

[jerry@CentOS project]$ git status -s
M string.c
?? string

[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
```

Above command will produce 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
```

CHAPTER

9

Push Operation

Jerry modified his last commit by using amend operation and he is ready to push changes. Push operation stores data permanently to the Git repository. After successful push operation other developers can see Jerry's changes.

He execute git log command to view commit details.

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

Above command will produce 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 git show command to review his changes.

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

Above command will produce 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

diff --git a/string.c b/string.c
new file mode 100644
index 0000000..7da2992
--- /dev/null
+++ b/string.c
@@ -0,0 +1,24 @@
+#include <stdio.h>
+
+size_t my_strlen(char *s)
+{
+
char *p = s;
+
```

```
+
while (*p)
+ ++p;
+ return (p -s );
+}
+
+int main(void)
+{
+ int i;
+ char *s[] = {
+ "Git tutorials",
+ "Tutorials Point"
+
};
+
+
+
for (i = 0; i < 2; ++i)
printf("string lenght of %s = %lu\n", s[i], my_strlen(s[i]));
+
+
return 0;
+}
```

Jerry is happy with his changes and he is ready to push his changes.

```
[jerry@CentOS project]$ git push origin master
```

Above command will produce 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 successfully pushed to the repository, now other developers can view his changes by performing clone or update operation.

CHAPTER

10

Update Operation

Modify existing function

Tom performs clone operation and after that sees new file string.c and he wanted to know who added this file to the repository ? And for what purpose ? So he executes git log command.

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

Above command will produce 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)
```

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

```
[tom@CentOS ~]$ cd project/

[tom@CentOS project]$ git log
```

Above command will produce 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
```

After observing log, he realizes that string.c file was added by Jerry to implement basic string operations. He is curious about Jerry's code. So he opens string.c file in text editor and immediately finds a bug. In my_strlen

function Jerry is not using constant pointer. So he decides to modify Jerry's code. After modification code will look like this.

```
[tom@CentOS project]$ git diff
```

Above command will produce 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
-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
```

Above command will produce 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
```

Above command will produce 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 file. After modification file will look like this.

```
[jerry@CentOS project]$ git diff
```

Above command will produce following result.

```
index 7da2992..bc864ed 100644
--- a/string.c
+++ b/string.c
30Git Tutorials
@@ -9,9 +9,20 @@ size_t my_strlen(char *s)
return (p -s );
}
+char *my_strcpy(char *t, char *s)
+{
+
char *p = t;
+
+ while (*t++ = *s++)
+ ;
+
+
return p;
+}
+
int main(void)
{
int i;
+
char p1[32];
char *s[] = {
"Git tutorials",
"Tutorials Point"
@@ -20,5 +31,7 @@ int main(void)
for (i = 0; i < 2; ++i)
printf("string lenght of %s = %lu\n", s[i], my_strlen(s[i]));
+
printf("%s\n", my_strcpy(p1, "Hello, World !!!"));
+
return 0;
}
```

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
```

Above command will produce 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 changes and he wants to push his changes

```
[jerry@CentOS project]$ git push origin master
```

Above command will produce 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 is not in sync. Because of this he can lose history of project. So to avoid this mess Git failed this operation. Now Jerry's has to first update its local repository and then and then only he can push his own changes.

Fetch latest changes

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

```
[jerry@CentOS project]$ git pull
```

Above command will produce 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
```

Above command will produce 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
```

Above command will produce 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
```

CHAPTER

11

Stash Operation

Suppose you are implementing a new feature for your product. Your code is in progress and suddenly customer escalation comes. Because of this you have to keep aside your new feature work for few hours. You cannot commit your partial code and also cannot throw away your changes. So you need some temporary space where you can store your partial changes and later on commit it.

In Git, stash operation takes your modified tracked files and stage changes and saves it 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 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 you're working directory is clean and all changes are saved on stack. Let us verify it with git status command.

```
[jerry@CentOS project]$ git status -s
?? string
```

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

```
[jerry@CentOS project]$ git stash list
stash@{0}: WIP on master: e86f062 Added my_strcpy function
```

Suppose you resolved customer escalation and you are back on your new feature and you want your half-done code. Just execute git stash pop command, it will remove changes from stack and place it in the current working directory.

```
[jerry@CentOS project]$ git status -s
?? string

[jerry@CentOS project]$ git stash pop
```

Above command will produce 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
```

CHAPTER

12

Move Operation

As name suggests move operation moves directory or file from one location to another. Tom decides to move source code into **src** directory. So modified directory structure will look like this.

```
[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 modified directory structure to remote repository so other developer 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 old directory structure.

```
[jerry@CentOS project]$ pwd
/home/jerry/jerry_repo/project

[jerry@CentOS project]$ ls
README string string.c
```

But after pull operation directory structure will get updated. Now Jerry can see **src** directory and 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
```

CHAPTER

13

Rename Operation

Till now Tom and Jerry both were using manual commands to compile their project. Jerry decides to create Makefile for their project and also give proper name to “string.c” file.

```
[jerry@CentOS project]$ pwd
/home/jerry/jerry_repo/project

[jerry@CentOS project]$ ls
README src

[jerry@CentOS project]$ cd src/

[jerry@CentOS src]$ git add Makefile

[jerry@CentOS src]$ git mv string.c string_operations.c

[jerry@CentOS src]$ git status -s
A Makefile
R string.c -> string_operations.c
```

Git is showing **R** before file name to indicate that file has been renamed.

For commit operation Jerry used -a flag, this makes git commit automatically detect modified files.

```
[jerry@CentOS src]$ git commit -a -m 'Added Makefile and renamed strings.c to
string_operations.c '

[master 94f7b26] Added Makefile and renamed strings.c to string_operations.c
1 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 src/Makefile
rename src/{string.c => string_operations.c} (100%)
```

After commit he pushesh his changes to the repository.

```
[jerry@CentOS src]$ git push origin master
```

Above command will produce following result.

```
Counting objects: 6, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 396 bytes, done.
Total 4 (delta 0), reused 0 (delta 0)
To gituser@git.server.com:project.git
```

```
7d9ea97..94f7b26 master -> master
```

Now other developers can view these modifications by updating their local repository.

CHAPTER

14

Delete Operation

Tom updated his local repository and finds compiled binary into src directory. After viewing commit message he realizes that compiled binary was added by Jerry.

```
[tom@CentOS src]$ pwd
/home/tom/project/src

[tom@CentOS src]$ ls
Makefile string_operations string_operations.c

[tom@CentOS src]$ file string_operations
string_operations: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked (uses
shared libs), for GNU/Linux 2.6.18, not stripped

[tom@CentOS src]$ git log
commit 29af9d45947dc044e33d69b9141d8d2dad37cc62
Author: Jerry Mouse <jerry@tutorialspoint.com>
Date: Wed Sep 11 10:16:25 2013 +0530

Added compiled binary
```

VCS is used to store source code only and not to executable binaries. So Tom decides to remove this file from the repository. For further operation he uses git rm command.

```
[tom@CentOS src]$ ls
Makefile string_operations string_operations.c

[tom@CentOS src]$ git rm string_operations
rm 'src/string_operations'

[tom@CentOS src]$ git commit -a -m "Removed executable binary"

[master 5776472] Removed executable binary
1 files changed, 0 insertions(+), 0 deletions(-)
delete mode 100755 src/string_operations
```

After commit he pushes his changes to the repository.

```
[tom@CentOS src]$ git push origin master
```

Above command will produce following result.

```
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 310 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To gituser@git.server.com:project.git
29af9d4..5776472 master -> master
```

CHAPTER

15

Fix Mistakes

Too err is human. So every VCS provides a feature to fix mistakes till certain point. Git provides feature by using that we can undo modifications that have been made to the local repository.

Suppose user accidently does some changes to his local repository and now he wants to throw away these changes. In such cases revert operation plays important role.

Revert uncommitted changes

Let us suppose Jerry accidently modifies file from his local repository. But he wants to throw away his modification. To handle this situation we can use git checkout command. We can use this command to revert the contents of a file.

```
[jerry@CentOS src]$ pwd
/home/jerry/jerry_repo/project/src

[jerry@CentOS src]$ git status -s
M string_operations.c

[jerry@CentOS src]$ git checkout string_operations.c

[jerry@CentOS src]$ git status -s
```

Even we can use git checkout command to obtain deleted file from local repository. Let us suppose Tom deletes file from the local repository and we want this file back. We can achieve this by using same command.

```
[tom@CentOS src]$ pwd
/home/tom/top_repo/project/src

[tom@CentOS src]$ ls -1
Makefile
string_operations.c

[tom@CentOS src]$ rm string_operations.c

[tom@CentOS src]$ ls -1
Makefile

[tom@CentOS src]$ git status -s
D string_operations.c
```

Git is showing letter **D** before filename. This is indicating that file has been deleted from the local repository.

```
[tom@CentOS src]$ git checkout string_operations.c

[tom@CentOS src]$ ls -1
Makefile
string_operations.c

[tom@CentOS src]$ git status -s
```

Note: We can perform all these operations before commit operation.

Remove changes from staging area

We have seen that when we perform add operation; file moves from local repository to the stating area. If user accidently modifies a file and adds it into staging area, but immediately he realizes that he did mistakes. And he wants to revert his changes. We can handle this situation by using git checkout command.

In Git there is one HEAD pointer that always points to latest commit. If you want to undo a change from staged area then you can use the git checkout command, but with checkout command you have to provide addition parameter which is HEAD pointer. The additional commit pointer parameter instructs the git checkout command to reset the working tree and also to remove the staged changes.

Let us suppose Tom modifies a file from his local repository. If we view status of this file, it will show file is modified but not added into staging area.

```
tom@CentOS src]$ pwd
/home/tom/top_repo/project/src
# Unmodified file

[tom@CentOS src]$ git status -s

# Modify file and view it's status.
[tom@CentOS src]$ git status -s
M string_operations.c

[tom@CentOS src]$ git add string_operations.c
```

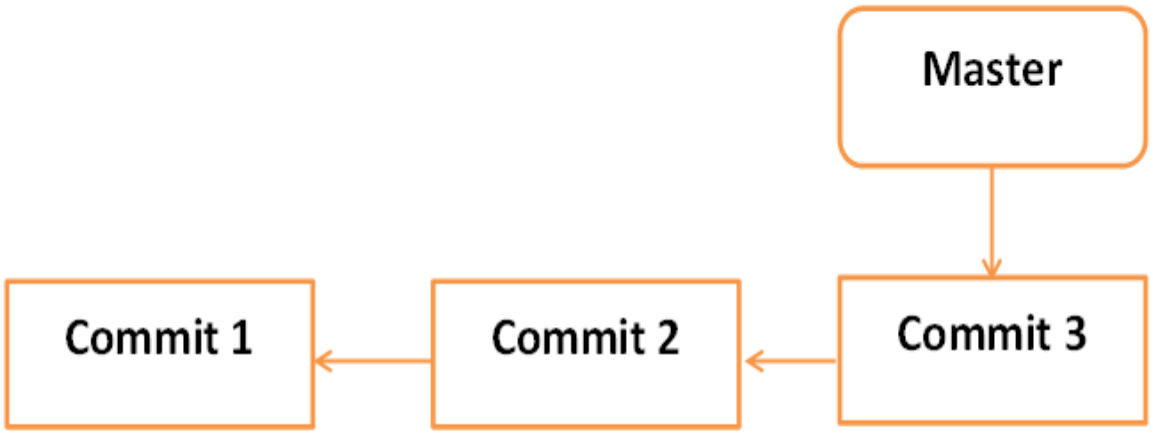
Git status shows that file is present in staging area, now revert it by using git checkout command and view status of reverted file.

```
[tom@CentOS src]$ git checkout HEAD -- string_operations.c

[tom@CentOS src]$ git status -s
```

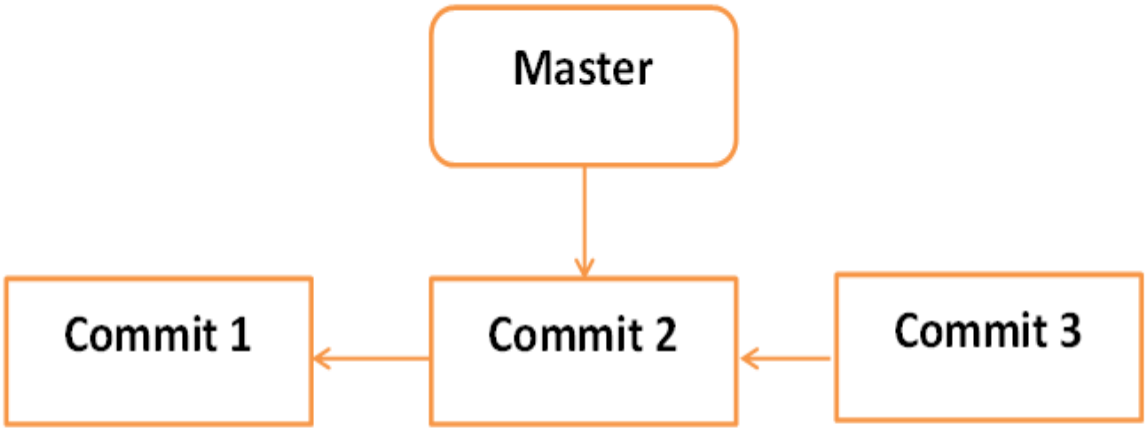
Move HEAD pointer with git reset

After doing few changes you may decide to remove these changes. The Git reset command is used to reset or revert some changes. We can perform three different types of reset operation.

Below diagram shows pictorial representation Git reset command.



Before git reset command



After git reset command

SOFT

Each branch has HEAD pointer which points to the latest commit. If we use Git reset command with --soft option followed by commit ID, then it will reset only HEAD pointer without destroying anything.

.git/refs/heads/master file stores the commit ID of the HEAD pointer. We can verify it by using git log -1 command.

```
[jerry@CentOS project]$ cat .git/refs/heads/master
577647211ed44fe2ae479427a0668a4f12ed71a1
```

Now view latest commit ID, which will match with above commit ID.

```
[jerry@CentOS project]$ git log -2
```

Above command will produce following result.

```
commit 577647211ed44fe2ae479427a0668a4f12ed71a1
Author: Tom Cat <tom@tutorialspoint.com>
Date: Wed Sep 11 10:21:20 2013 +0530

Removed executable binary


commit 29af9d45947dc044e33d69b9141d8d2dad37cc62
Author: Jerry Mouse <jerry@tutorialspoint.com>
Date: Wed Sep 11 10:16:25 2013 +0530
```

```
Added compiled binary
```

Let us reset HEAD pointer.

```
[jerry@CentOS project]$ git reset --soft HEAD~
```

Now we just reset HEAD pointer back by one position. Let us check contents of **.git/refs/heads/master file**.

```
[jerry@CentOS project]$ cat .git/refs/heads/master
29af9d45947dc044e33d69b9141d8d2dad37cc62
```

Commit ID from file is changed, now verify it by viewing commit messages.

```
jerry@CentOS project]$ git log -2
```

Above command will produce following result.

```
commit 29af9d45947dc044e33d69b9141d8d2dad37cc62
Author: Jerry Mouse <jerry@tutorialspoint.com>
Date: Wed Sep 11 10:16:25 2013 +0530

Added compiled binary


commit 94f7b26005f856f1a1b733ad438e97a0cd509c1a
Author: Jerry Mouse <jerry@tutorialspoint.com>
Date: Wed Sep 11 10:08:01 2013 +0530

Added Makefile and renamed strings.c to string_operations.c
```

MIXED

Git reset with --mixed option revert changes from the staging area which hasn't been committed yet. It only revert changes form staging area. The actual changes made to the working copy of the file are unaffected. The default Git reset is equivalent to git reset -- mixed.

For more details refer section **Remove changes from staging area** from same chapter.

HARD

If you use --hard option with Git reset command, it will clear staging area; it will reset HEAD pointer to latest commit of specific commit ID and delete the local file changes too.

Let us check the commit ID

```
[jerry@CentOS src]$ pwd
/home/jerry/jerry_repo/project/src

[jerry@CentOS src]$ git log -1
```

Above command will produce following result.

```
commit 577647211ed44fe2ae479427a0668a4f12ed71a1
Author: Tom Cat <tom@tutorialspoint.com>
Date: Wed Sep 11 10:21:20 2013 +0530

Removed executable binary
```

Jerry modified file by adding single line comment at the start of file.

```
[jerry@CentOS src]$ head -2 string_operations.c
/* This line be removed by git reset operation */
#include <stdio.h>
```

He verified it by using git status command.

```
[jerry@CentOS src]$ git status -s
M string_operations.c
```

Jerry adds modified file to the staging area and verifies it with git status operation.

```
[jerry@CentOS src]$ git add string_operations.c
[jerry@CentOS src]$ git status
```

Above command will produce following result.

```
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#
modified: string_operations.c
#
```

Git status is showing that file is present in the staging area. Now reset HEAD with --hard option.

```
[jerry@CentOS src]$ git reset --hard 577647211ed44fe2ae479427a0668a4f12ed71a1

HEAD is now at 5776472 Removed executable binary
```

Git reset command succeeded, which will revert file from the staging area as well as remove local changes made to the file.

```
[jerry@CentOS src]$ git status -s
```

Git status is showing that file is reverted from the staging area.

```
[jerry@CentOS src]$ head -2 string_operations.c
#include <stdio.h>
```

Head command also shows that reset operation removed local changes too.

CHAPTER

16

Tag Operation

Tag operation allows giving meaningful name to a specific version in the repository. Tom decides to tag their project code so they can more easily access it later.

Create tag

Let us tag current HEAD by using git tag command. He provides tag name with -a option and provides tag message with –m option.

```
tom@CentOS project]$ pwd
/home/tom/top_repo/project

[tom@CentOS project]$ git tag -a 'Release_1_0' -m 'Tagged basic string operation
code' HEAD
```

If you want to tag particular commit then use appropriate COMMIT ID instead of HEAD pointer. Tom uses below command to push tag to the remote repository.

```
[tom@CentOS project]$ git push origin tag Release_1_0
```

Above command will produce following result.

```
Counting objects: 1, done.
Writing objects: 100% (1/1), 183 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
To gituser@git.server.com:project.git
* [new tag]
Release_1_0 -> Release_1_0
```

View tags

Tom created tag. Now Jerry can view all available tags by using Git tag command with –l option.

```
[jerry@CentOS src]$ pwd
/home/jerry/jerry_repo/project/src

[jerry@CentOS src]$ git pull
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
```

```
From git.server.com:project
* [new tag]
Release_1_0 -> Release_1_0
Current branch master is up to date.

[jerry@CentOS src]$ git tag -l
Release_1_0
```

Jerry uses Git show command followed by tag name to view more details about tag.

```
[jerry@CentOS src]$ git show Release_1_0
```

Above command will produce following result.

```
tag Release_1_0
Tagger: Tom Cat <tom@tutorialspoint.com>
Date: Wed Sep 11 13:45:54 2013 +0530

Tagged basic string operation code


commit 577647211ed44fe2ae479427a0668a4f12ed71a1
Author: Tom Cat <tom@tutorialspoint.com>
Date: Wed Sep 11 10:21:20 2013 +0530

Removed executable binary

diff --git a/src/string_operations b/src/string_operations
deleted file mode 100755
index 654004b..0000000
Binary files a/src/string_operations and /dev/null differ
```

Delete tags

Tom uses below command to delete tag from local as well as remote repository.

```
[tom@CentOS project]$ git tag
Release_1_0

[tom@CentOS project]$ git tag -d Release_1_0
Deleted tag 'Release_1_0' (was 0f81ff4)
# Remove tag from remote repository.

[tom@CentOS project]$ git push origin :Release_1_0
To gituser@git.server.com:project.git
- [deleted]
Release_1_0
```

CHAPTER

17

Patch Operation

Patch is text file whose contents are similar to Git diff, but along with code it also has metadata about commits e.g. commit ID, date, commit message etc. We can create patch from commits and other people can apply them to their repository.

Jerry implements strcat function for their project. Jerry can create path of his code and send to Tom. Then he can apply received patch to his code.

Jerry uses Git format-patch command to create patch for latest commit. If you want to create patch for specific commit then use **COMMIT_ID** with format-patch command.

```
[jerry@CentOS project]$ pwd
/home/jerry/jerry_repo/project/src

[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 my_strcat function"

[master b4c7f09] Added my_strcat function
1 files changed, 13 insertions(+), 0 deletions(-)

[jerry@CentOS src]$ git format-patch -1
0001-Added-my_strcat-function.patch
```

Above command creates **.patch** files inside current working directory. Tom can use this patch to modify his files. Git provides two commands to apply patches **git am** and **git apply** respectively. **Git apply**command modifies the local files without creating commit while **git am** command modifies file and creates commit as well.

To apply patch and create commit use following command.

```
[tom@CentOS src]$ pwd
/home/tom/top_repo/project/src

[tom@CentOS src]$ git diff

[tom@CentOS src]$ git status -s

[tom@CentOS src]$ git apply 0001-Added-my_strcat-function.patch
```

```
[tom@CentOS src]$ git status -s
M string_operations.c
?? 0001-Added-my_strcat-function.patch
```

Patch gets applied successfully, now we can view modifications by using git diff command.

```
[tom@CentOS src]$ git diff
```

Above command will produce following result.

```
diff --git a/src/string_operations.c b/src/string_operations.c
index 8ab7f42..f282fcf 100644
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@ -1,5 +1,16 @@
#include <stdio.h>
+char *my_strcat(char *t, char *s)
diff --git a/src/string_operations.c b/src/string_operations.c
index 8ab7f42..f282fcf 100644
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@ -1,5 +1,16 @@
#include <stdio.h>
+char *my_strcat(char *t, char *s)
+{
+
char *p = t;
+
+
+
while (*p)
++p;
+
while (*p++ = *s++)
+ ;
+ return t;
+}
+
size_t my_strlen(const char *s)
{
const char *p = s;
@@ -23,6 +34,7 @@ int main(void)
{
```

CHAPTER

18

Managing Branches

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

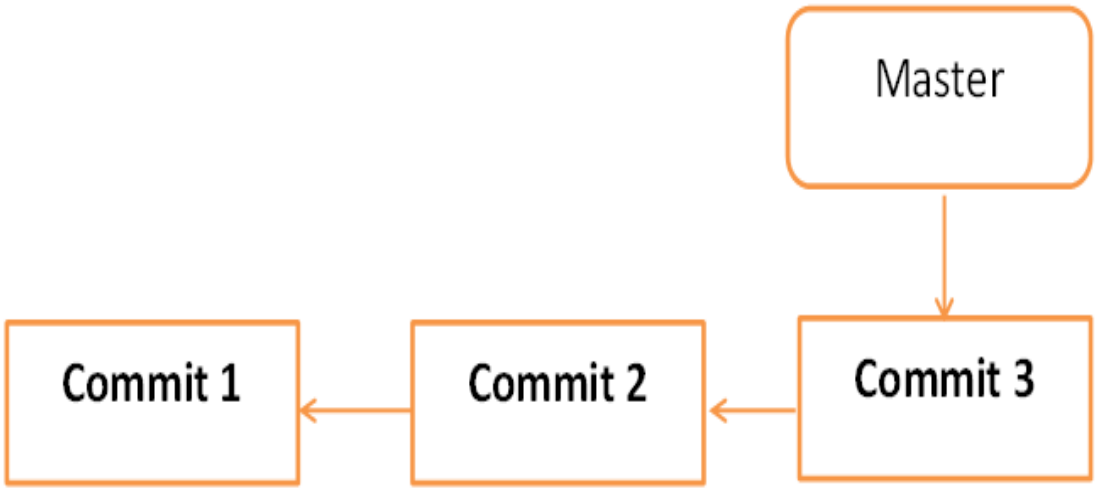
Create branch

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

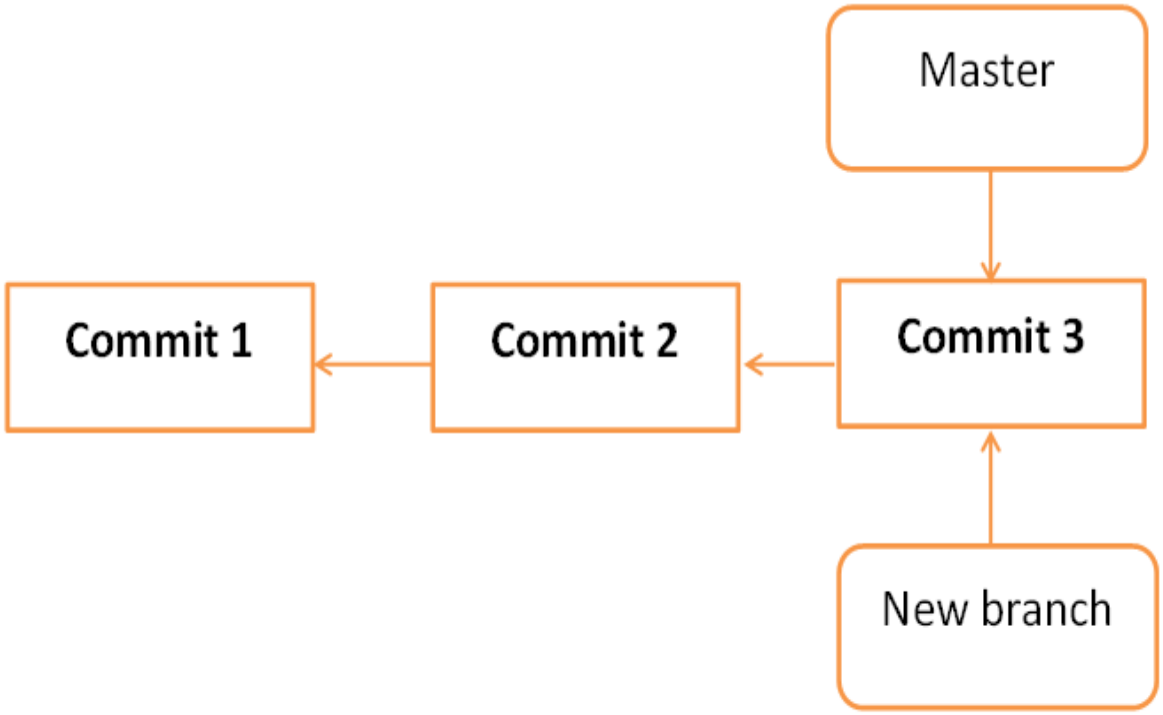
```
[jerry@CentOS src]$ git branch new_branch

[jerry@CentOS src]$ git branch
* master
new_branch
```

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

Below is pictorial representation for create branch operation



Before create branch command



After create branch operation

Switch branch

Jerry uses 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 above example we used two commands to create and switch branch respectively. Git provides –b option with checkout command, this operation creates new branch and immediately switch 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 branch

A branch can be deleted by providing –D option with git branch command. But before deleting 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 branch

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

```
[jerry@CentOS src]$ git branch
* master
new_branch

[jerry@CentOS src]$ git branch -m new_branch wchar_support
```

Now git branch command will show new branch name.

```
[jerry@CentOS src]$ git branch
* master
wchar_support
```

Merge two branches

Jerry implements function to return string length of wide character string. New code will look like this

```
[jerry@CentOS src]$ git branch
master
* wchar_support

[jerry@CentOS src]$ pwd
/home/jerry/jerry_repo/project/src

[jerry@CentOS src]$ git diff
```

Above command producess following result.

```
t a/src/string_operations.c b/src/string_operations.c
index 8ab7f42..8fb4b00 100644
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@ -1,4 +1,14 @@
#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 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(-)
```

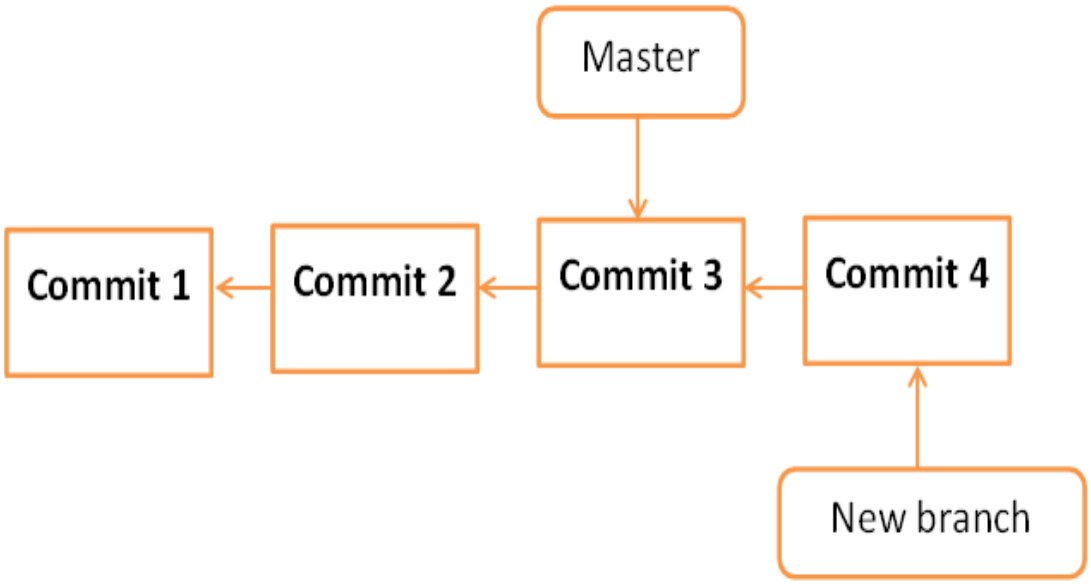
Note that Jerry is pushing these changes to the new branch that is why he used **wchar_support**branch name instead of **master** branch.

```
[jerry@CentOS src]$ git push origin wchar_support <------------- Observer
branch_name
```

Above command will produce 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 commiting changes in branch, new branch will look like this.



After commit in new branch

Tom is curious about what Jerry is doing in his private branch that is why he checks log from**wchar_support** branch.

```
[tom@CentOS src]$ pwd
/home/tom/top_repo/project/src

[tom@CentOS src]$ git log origin/wchar_support -2
```

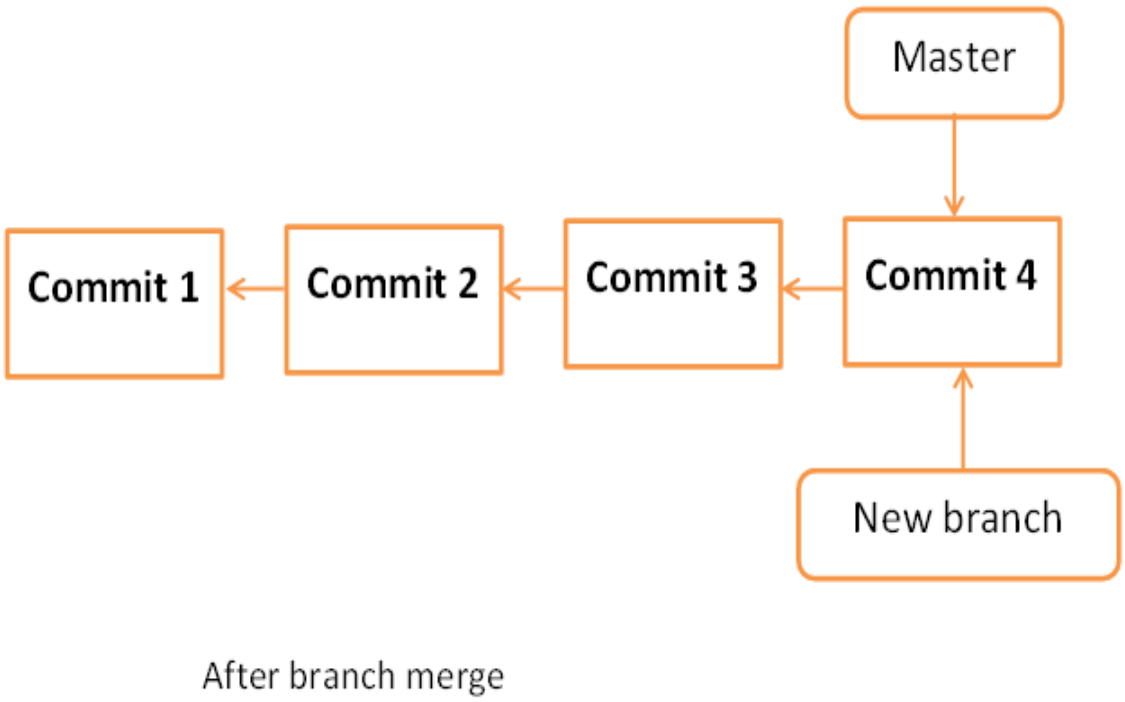
Above command will produce following result.

```
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 strlen function for wide character and he wants the same functionality into master branch. Instead of re-implementing he decides to take Jerry's code by merging his branch to 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 merge operation master branch will look like this.



After branch merge

Now **wchar_support** branch is merged to the master branch we can verify it by viewing commit message and also by viewing modification into 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
```

Above command will produce 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

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

Git merge command tries to put the commits from other branch 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 current local branch to something like this A->B->C->D->X->Y

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 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.

CHAPTER

19

Handling Conflicts

Perform changes in wchar_support branch

Jerry is working in **wchar_support** branch. He changes the name of the functions and after testing he commits his changes.

```
[jerry@CentOS src]$ git branch
 master
* wchar_support
[jerry@CentOS src]$ git diff
```

Above command producess following result

```
diff --git a/src/string_operations.c b/src/string_operations.c
index 8fb4b00..01ff4e0 100644
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@ -1,7 +1,7 @@
#include <stdio.h>
#include <wchar.h>
-size_t w_strlen(const wchar_t *s)
+size_t my_wstrlen(const wchar_t *s)
{
const wchar_t *p = s;
```

After verifying code he commits his changes.

```
[jerry@CentOS src]$ git status -s
M string_operations.c

[jerry@CentOS src]$ git add string_operations.c

[jerry@CentOS src]$ git commit -m 'Changed function name'
[wchar_support 3789fe8] Changed function name
1 files changed, 1 insertions(+), 1 deletions(-)

[jerry@CentOS src]$ git push origin wchar_support
```

Above command will produce following result.

```
Counting objects: 7, done.
```

```
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 409 bytes, done.
Total 4 (delta 1), reused 0 (delta 0)
To gituser@git.server.com:project.git
64192f9..3789fe8 wchar_support -> wchar_support
```

Perfrom changes in master branch

Meanwhile in the master branch Tom also changes the name of the same function and pushes his changes to the master branch.

```
[tom@CentOS src]$ git branch
* master
[tom@CentOS src]$ git diff
```

Above command producess following result.

```
diff --git a/src/string_operations.c b/src/string_operations.c
index 8fb4b00..52bec84 100644
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@ -1,7 +1,8 @@
#include <stdio.h>
#include <wchar.h>
-size_t w_strlen(const wchar_t *s)
+/* wide character strlen fucntion */
+size_t my_wc_strlen(const wchar_t *s)
{
const wchar_t *p = s;
```

After verifying diff he commits his changes.

```
[tom@CentOS src]$ git status -s
M string_operations.c

[tom@CentOS src]$ git add string_operations.c

[tom@CentOS src]$ git commit -m 'Changed function name from w_strlen to
my_wc_strlen'
[master ad4b530] Changed function name from w_strlen to my_wc_strlen
1 files changed, 2 insertions(+), 1 deletions(-)

[tom@CentOS src]$ git push origin master
```

Above command will produce following result.

```
Counting objects: 7, done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 470 bytes, done.
Total 4 (delta 1), reused 0 (delta 0)
To gituser@git.server.com:project.git
64192f9..ad4b530 master -> master
```

In **wchar_support** branch Jerry implements strchr function for wide character string. After testing he commits and pushes his changes to the **wchar_support** branch.

```
[jerry@CentOS src]$ git branch
master
* wchar_support
```

```
[jerry@CentOS src]$ git diff
```

Above command producess following result.

```
diff --git a/src/string_operations.c b/src/string_operations.c
index 01ff4e0..163a779 100644
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@ -1,6 +1,16 @@
#include <stdio.h>
#include <wchar.h>
+wchar_t *my_wstrchr(wchar_t *ws, wchar_t wc)
+{
+
while (*ws) {
+
if (*ws == wc)
+
return ws;
+
++ws;
+ }
+ return NULL;
+}
+
size_t my_wstrlen(const wchar_t *s)
{
const wchar_t *p = s;
```

After verifying changes, he commit his changes.

```
[jerry@CentOS src]$ git status -s
M string_operations.c

[jerry@CentOS src]$ git add string_operations.c

[jerry@CentOS src]$ git commit -m 'Addded strchr function for wide character
string'
[wchar_support 9d201a9] Addded strchr function for wide character string
1 files changed, 10 insertions(+), 0 deletions(-)

[jerry@CentOS src]$ git push origin wchar_support
```

Above command will produce following result.

```
Counting objects: 7, done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 516 bytes, done.
Total 4 (delta 1), reused 0 (delta 0)
To gituser@git.server.com:project.git
3789fe8..9d201a9 wchar_support -> wchar_support
```

Tackle conflicts

Tom wanted to see, what Jerry is doing in his private branch? That is why he tries to pull latest changes from **wchar_support** branch but Git aborts operation with following error message.

```
[tom@CentOS src]$ git pull origin wchar_support
```

Above command producess following result.

```
remote: Counting objects: 11, done.
63Git Tutorials
remote: Compressing objects: 100% (8/8), done.
remote: Total 8 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
From git.server.com:project
* branch
wchar_support -> FETCH_HEAD
Auto-merging src/string_operations.c
CONFLICT (content): Merge conflict in src/string_operations.c
Automatic merge failed; fix conflicts and then commit the result.
```

Resolve conflicts

From error message, it is clear that there is conflict in src/string_operations.c . He runs git diff command to view further details.

```
[tom@CentOS src]$ git diff
```

Above command producess following result.

```
diff --cc src/string_operations.c
index 52bec84,163a779..0000000
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@@ -1,8 -1,17 +1,22 @@@
#include <stdio.h>
#include <wchar.h>
++<<<<<<< HEAD
+/* wide character strlen fucntion */
+size_t my_wc_strlen(const wchar_t *s)
++=======
+ wchar_t *my_wstrchr(wchar_t *ws, wchar_t wc)
+{
+
+
while (*ws) {
if (*ws == wc)
+
return ws;
+
++ws;
+ }
+ return NULL;
+}
+
+ size_t my_wstrlen(const wchar_t *s)
++>>>>>>>9d201a9c61bc4713f4095175f8954b642dae8f86
{
const wchar_t *p = s;
```

As both Tom and Jerry changed name of the same function Git doesn't know what to do that is why it asks user to resolve conflict manually.

Tom decides to keep function name suggested by Jerry, but he keeps comment as it is; which is added by him. After removing conflict markers git diff will look like this.

```
[tom@CentOS src]$ git diff
```

Above command producess following result.

```
diff --cc src/string_operations.c
diff --cc src/string_operations.c
index 52bec84,163a779..0000000
--- a/src/string_operations.c
+++ b/src/string_operations.c
@@@ -1,8 -1,17 +1,18 @@@
#include <stdio.h>
#include <wchar.h>
+ wchar_t *my_wstrchr(wchar_t *ws, wchar_t wc)
+{
+
while (*ws) {
+
if (*ws == wc)
+
return ws;
+
++ws;
+ }
+ return NULL;
+}
+
+/* wide character strlen fucntion */
- size_t my_wc_strlen(const wchar_t *s)
+ size_t my_wstrlen(const wchar_t *s)
{
const wchar_t *p = s;
```

As Tom modified files, he has to commit these changes first and after that he can pull changes.

```
[tom@CentOS src]$ git commit -a -m 'Resolved conflict'
[master 6b1ac36] Resolved conflict

[tom@CentOS src]$ git pull origin wchar_support.
```

Tom has resolved conflict, now pull operation will succeed.

CHAPTER

20

Different Platforms

GNU/Linux and Mac OS uses **line-feed (LF)** or new line as line ending character while Windows uses**line-feed and carriage-return (LFCR)** combination to represent line ending character.

To avoid unnecessary commits because of these line ending differences we have to configure Git client to write the same line ending to the Git repository.

For Windows system we can configure Git client to convert line endings to **CRLF** format while checking out and convert them back to **LF** format during commit operation. Below setting will do needful.

```
[tom@CentOS project]$ git config --global core.autocrlf true
```

For GNU/Linux or Mac OS we can configure Git client to convert line endings from **CRLF** to **LF** while performing checkout operation.

```
[tom@CentOS project]$ git config --global core.autocrlf input
```

CHAPTER

21

Online Repositories

GitHub is a web-based hosting service for software development projects that use 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.

Create GitHub repository

Go to github.com. If you have already **GitHub** account, then login using that account or create new one. Follow the steps from github.com website to create new repository.

Push operation

Tom decides to use **GitHub** server. To start 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
```

Abover command will produce following result.

```
Hello, World !!!
```

After verifying his code, he initialized directory with 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 **GitHub** repository URL as a remote origin and push his changes to the remote repository.

Note: We already discussed all these steps in chapter 4 under create bare repository section.

```
[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 namd and password. After successful authentication, operation will suceed.

Above command will produce 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 command discussed in this chapter with **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 new directory in his home directory and clones **GitHub** repository by using git clone command.

```
[jerry@CentOS]$ pwd
/home/jerry

[jerry@CentOS]$ mkdir jerry_repo

[jerry@CentOS]$ git clone https://github.com/kangralkar/test_repo.git
```

Above command produces 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 verified directory contents by executing ls command.

```
[jerry@CentOS]$ ls
test_repo

[jerry@CentOS]$ ls test_repo/
hello.c
```