**GIT**

GIT is a distributed revision control and source code management system with an emphasis on speed. GIT is free software distributed under the terms of the GNU General Public License version 2.

**Why GIT:**

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.

Distributed Version Control System 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.

**Advantage of GIT:**

* Free and open source
* Fast and small
* Implicit backup
* Security
* No need of powerful hardware
* Easier branching.

**Where we have to download:**

Based on our system specification we have download GIT tool form the below mentioned website

<https://git-scm.com/downloads>

**The Key Words we used in the GIT:**

**Repository**

A copy of a complete project folder is called **repository** that lives on your computer. The repository contains the information like files, images and etc.,

### 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.

### 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.

**GIT GUI and GIT Bash**

We can use this tool in the following ways

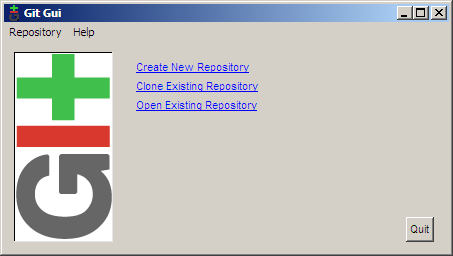
* 1. Git GUI – Graphical User Interface mode (Default Mode)
  2. Git Bash – Comment Line Mode will be open for the corresponding repository.

**Steps to access Git Bash.**

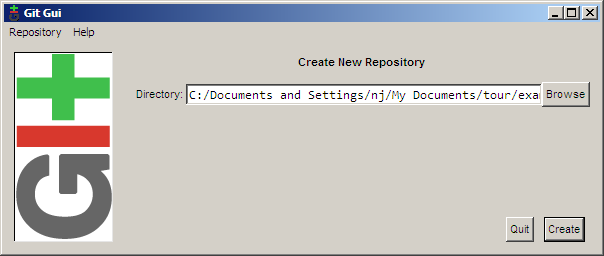
Start -> Git GUI -> Repository -> GUI Bash here

**Creating New Repository**

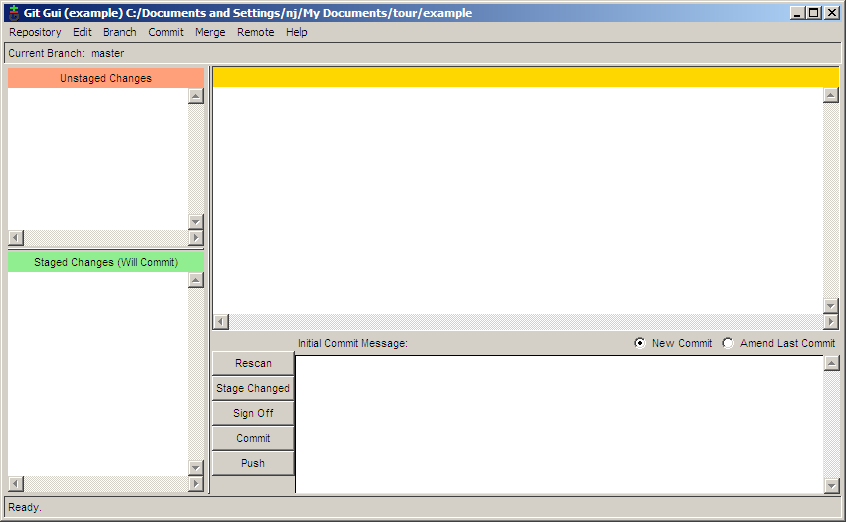
To create a repository, first create the folder you want the project to live under. Next, right click on the folder and choose Git GUI Here.



Choosing Create New Repository brings us to the next dialog.



Fill in the path to your new directory and click Create. You will then be presented with the main interface of git gui, which is what will be shown from now on when you right click on your folder and click Git GUI Here.



**Working Directory and Staging Area in GIT:**

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.

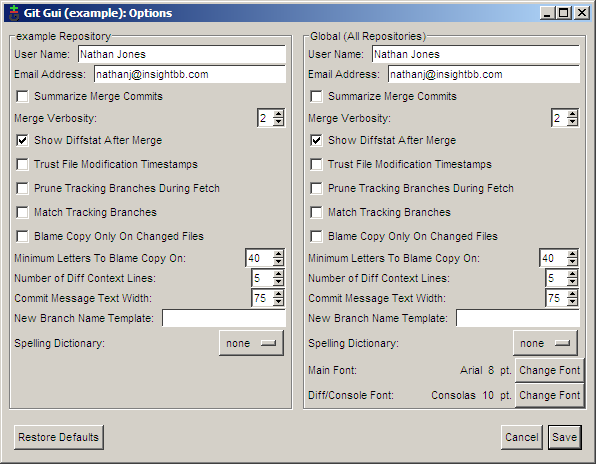
**Step 1**: You modify a file from the working directory.

**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.

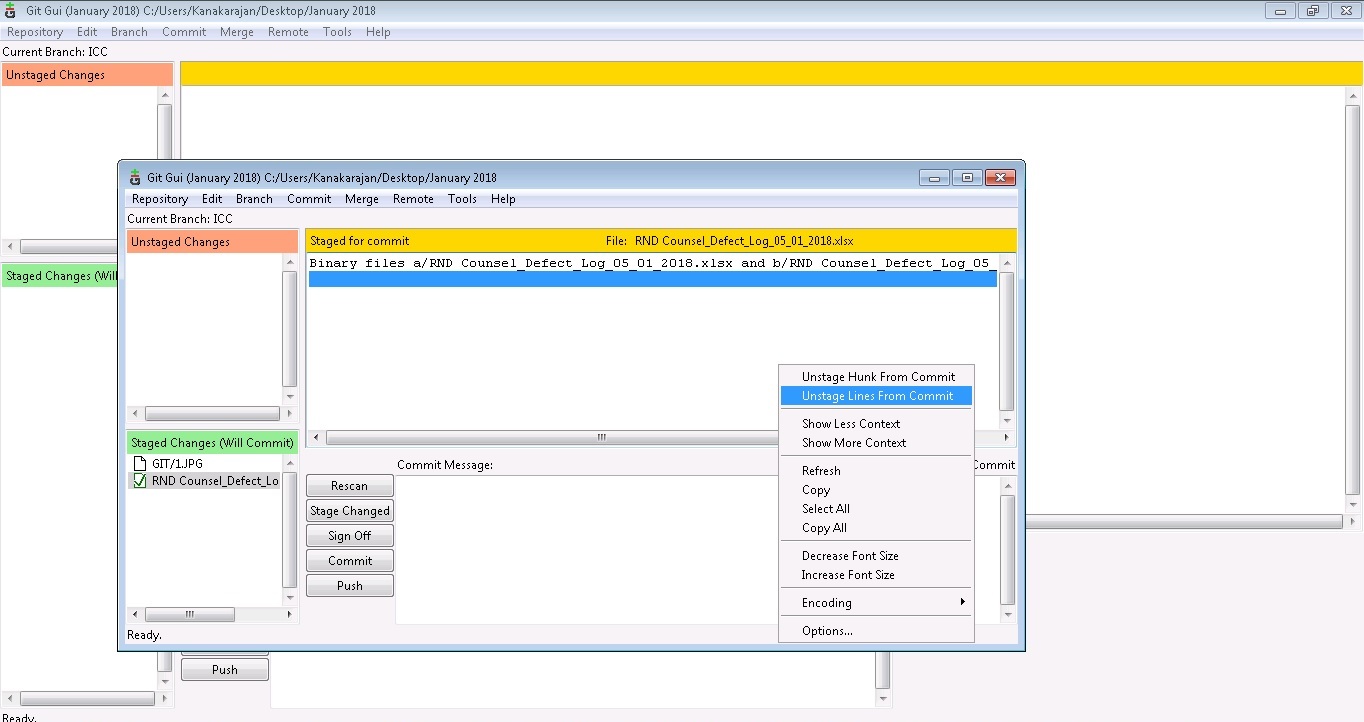


Now that the repository has been set up, you will need to tell git who you are so that commit messages will have the correct author. To do this, choose *Edit → Options*.



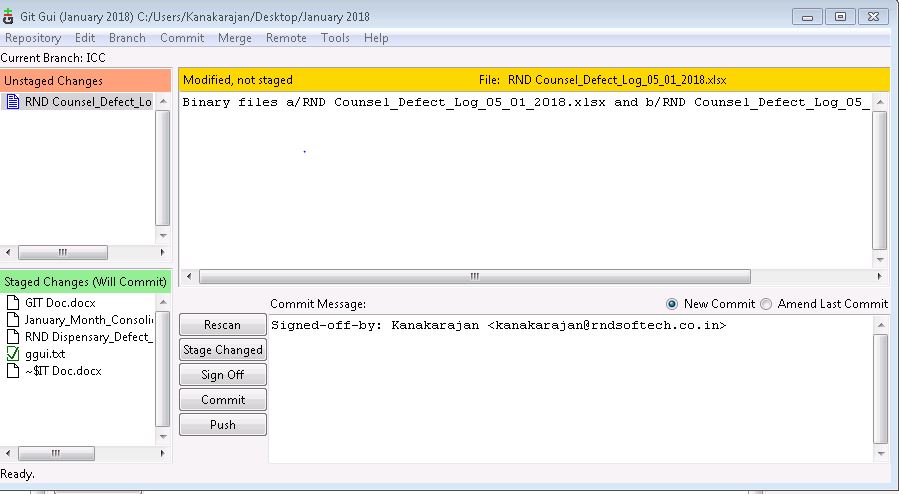
In the options dialog, there are two versions of each preference. On the left side of the dialog are options that you want for this repository only, while the right side contains the global options which apply to all repositories. The defaults for these options are sensible so just fill in the user name and email for now. If you have a favorite font, you may want to set it now as well.

Moving files from unstaged to staged state

Select the file (displaying on the unstaged state) you want to move to the staged state. The selected file content will be displayed on the Staged for comments screen. Right click the Staged for comments screen and select the Unstated Lines from Comment options

## Committing

Now that the repository has been created, it is time to create something to commit. For this example, I added a file called RND Dispensary\_Defect\_Log\_08\_01\_2018\_Retest.exl to the GIT Gui. Once the user Click on the Stage changed the added file moved from the unstated to staged changes. After that user has entered the enter Comments in Commit message part that what he has done on. Then Click Commit button.



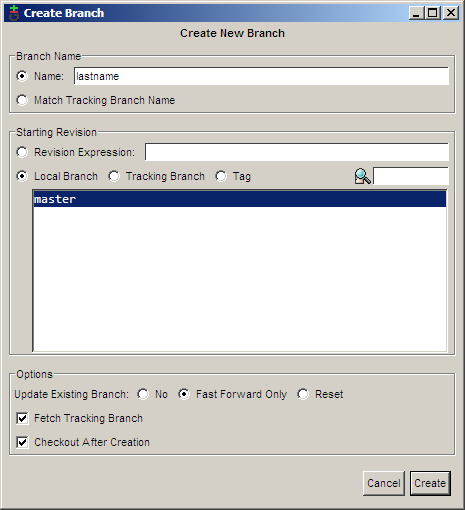
**Branching**

Suppose if we want to start adding new features for our next big version of the program. But, we also want to keep a stable, maintenance version of the program to fix bugs on. To do this, we will create a branch for our new development. To create a new branch in git GUI, choose Branch → Create. The big feature that I would like to add is to ask the user for their last name, so I am calling this branch last name. The default options in the Create Branch dialog are all fine, so just enter the name and click Create.

Steps to create Branching

Get GUI ->Branch -> Create -> Enter the Branch Name -> Click Create

Git Bash -> Git Checkout –B LastName.



Now that I am on the lastname branch, I can make my new modifications:

#include <stdio.h>

#include <string.h>

int main(int argc, char \*\*argv)

{

char first[255], last[255];

printf("Enter your first name: ");

fgets(first, 255, stdin);

first[strlen(first)-1] = '\0'; /\* remove the newline at the end \*/

printf("Now enter your last name: ");

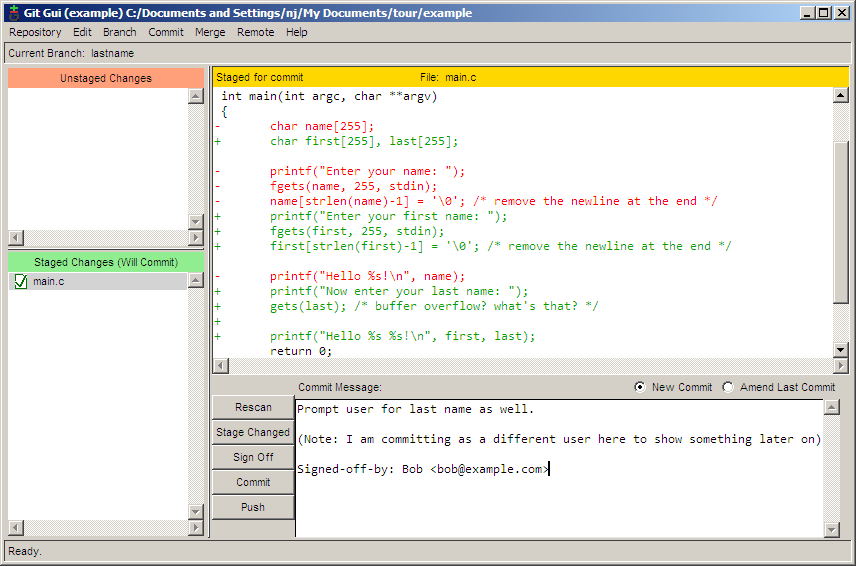
gets(last); /\* buffer overflow? what's that? \*/

printf("Hello %s %s!\n", first, last);

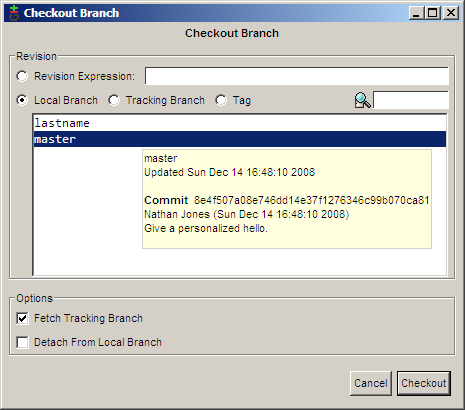
return 0;

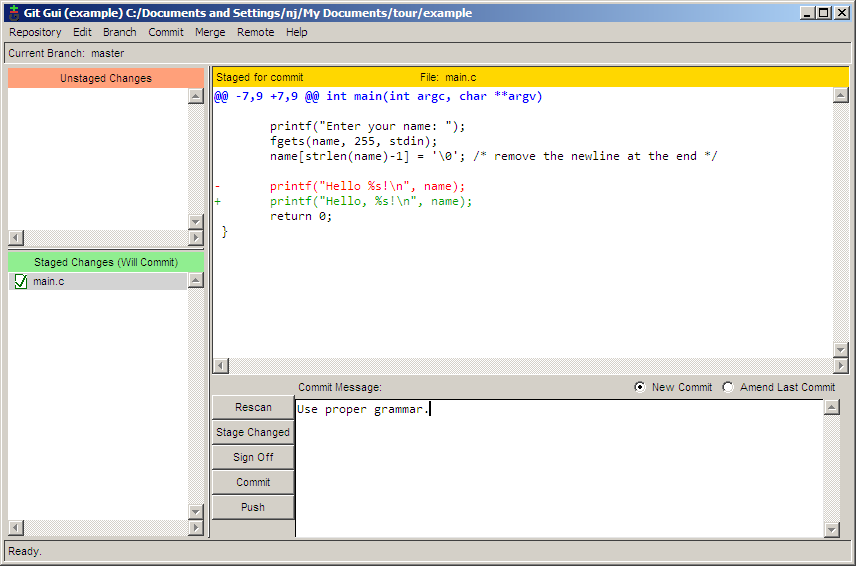
}

And then I can commit the change. Note here that I am committing using a different name. This is to show off something later. Normally you would always use the same name when committing.

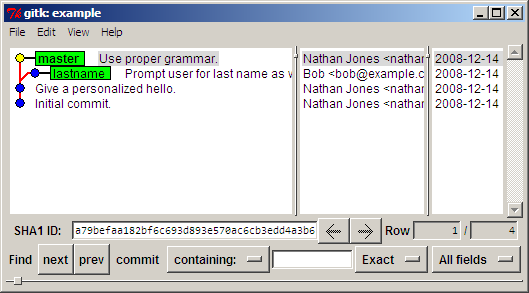


Meanwhile, a user informs us that not displaying a comma when directly addressing someone is a serious bug. In order to make this bug fix on our stable branch, we must first switch back to it. This is done using *Branch → Checkout*.



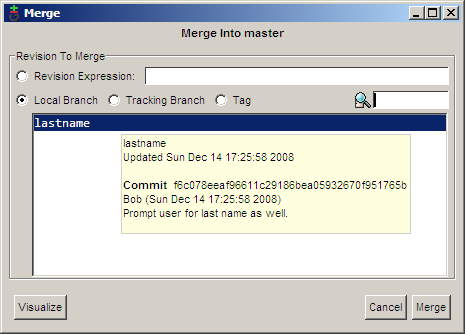
Now we can fix our major bug.

If we choose *Repository → Visualize All Branch History*, we can see how our history is shaping up.

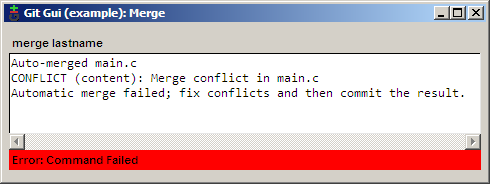


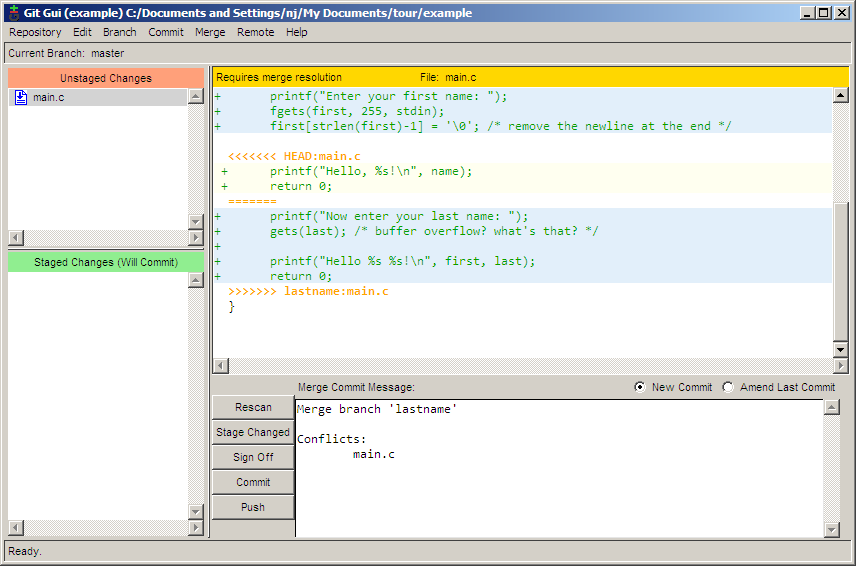
## Merging

After days of work, we decide that our lastname branch is stable enough to be merged into the master branch. To perform the merge, use *Merge → Local Merge*.

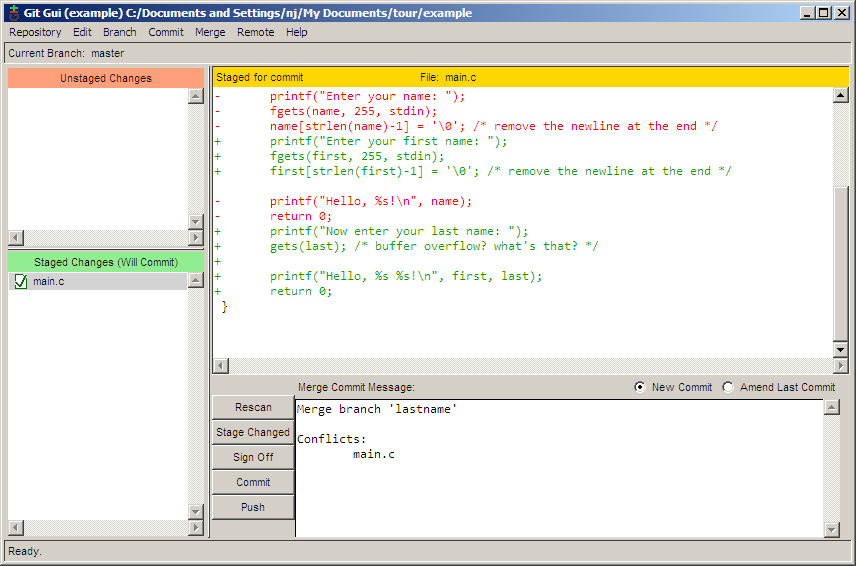


Because the two different commits made two different modifications to the same line, a conflict occurs.



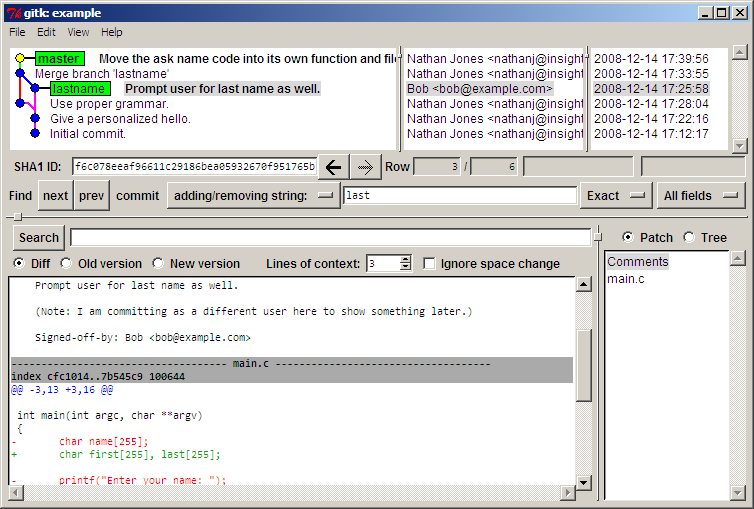


This conflict can be resolved using any text editor. After resolving the conflict, stage the changes by clicking the file icon and then commit the merge by clicking the Commit button.



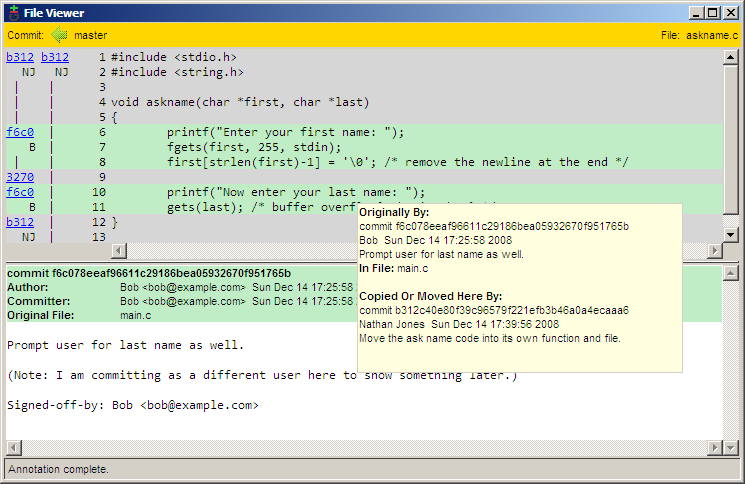
## Viewing History

The history of the repository can be viewed and searched by choosing *Repository → Visualize All Branch History*. In the next screenshot, I am trying to find which commit added the last variable by searching for all commits which added or removed the word last. Commits which match the search are bolded, making it quick and easy to spot the desired commit.



A few days later, someone looks through our code and sees that the gets function could cause a buffer overflow. Being the type to point fingers, this person decides to run a git blame to see who last modified this line of code. The problem is that Bob is the one who committed the line, but I was the one who last touched it when I moved the line into a different file. Obviously, I am not to blame (of course). Is git smart enough to figure this out? Yes, it is.

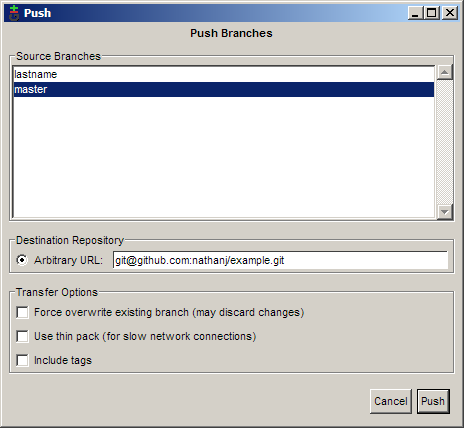
To run blame, select *Repository → Browse master's Files*. From the tree that pops up, double click on the file with the line in question which in this case is askname.c. Hovering the mouse over the line in question shows a tooltip message that tells us all we need to know.

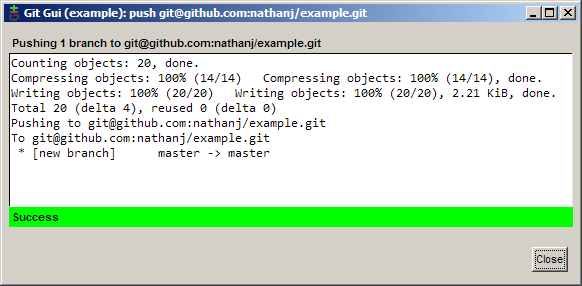


## Pushing to a Remote Server

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.

Steps: Git Gui -> *Remote → Push* will open up the push dialog. Typing in the commit address for the project and clicking Push will send the changes on their way.





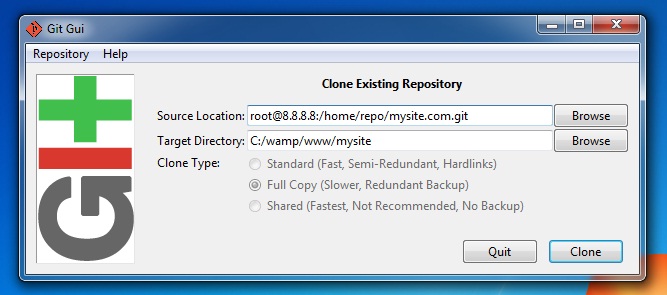
**Cloning Repository**

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.

Launch the GUI by clicking on Start > All programs > Git > Git GUI. We are cloning a repo, so choose the “Clone Existing Repository” option.

#### Cloning the Repo

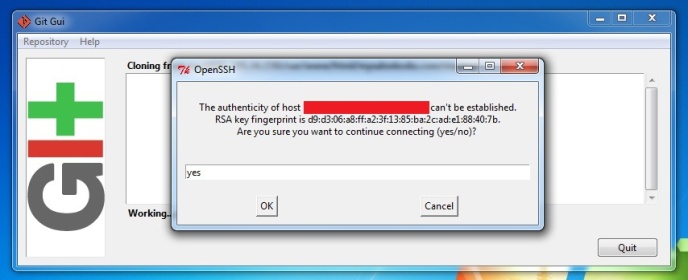
The window you get looks like the following:

[](http://twincreations.co.uk/wp-content/uploads/2013/05/git_clone.jpg)

#### The Git Path

When cloning from the repo, it is enough to tell Git the folder where the repo lives. If our repo lived at /home/repos/mysite.git/.git, we could just point to the /home/repos/mysite.git directory and Git would work it out. This applies to both bare and non-bare repos.

We click “clone”.

[](http://twincreations.co.uk/wp-content/uploads/2013/05/git_clone_rsa.jpg)

Git freaks out because it cannot authenticate the server. Just type “yes” to connect to the host (this is safe because we are requesting the address. Just make sure there are no typos).