

Section 1 - Git

G1.1 About Git

Git is a free *version control system* (VCS) that can be downloaded and run on your own computer as a stand-alone application. You can use it to track changes to your own projects, which would not necessarily involve any online activity (besides the original download). In this scenario, you would be using **Git** to organize and track changes you make to projects so that you could go back in time to see what a project looked like at some previous stage.

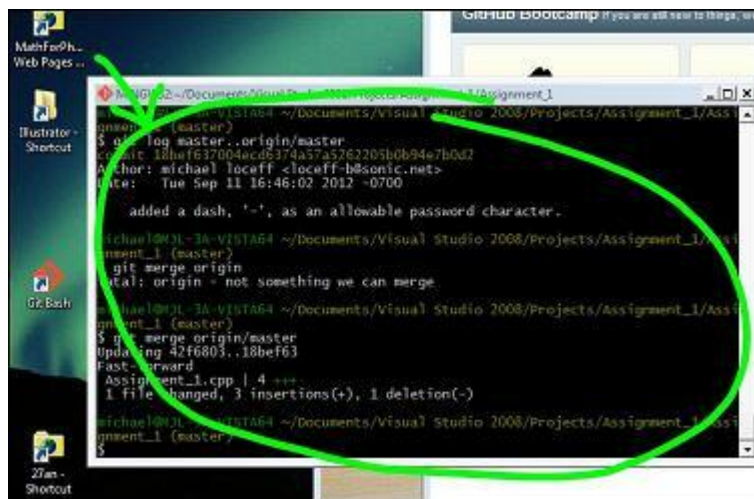
Another way **Git** can be used is through its networking options, which allow you to *push* and *pull* snapshots of your project to/from remote servers. When used in this way, **Git** gives you two additional benefits:

1. Your work is now backed-up on a remote server in case your local files are somehow removed or destroyed.
2. The remote server can be accessed by other members of a team which may be collaborating on the development, maintenance and debugging of the project.

In this scenario, a remote **Git** server (or *remote repository*), acts a little like a **Dropbox** account, except that it has the many tools needed for version tracking and collaborative development.

G1.2 The Console (or Terminal)

These days, the average person hardly ever sees a *console* -- those strange, often dark, rectangles of text used by programmers and sysadmins:



```
Michael@3A-VISTA64: ~/Documents/Visual Studio 2008/Projects/Assignment_1/Assignment_1
$ git log master..origin/master
commit 18b1f6370d4cc69374a57a5262205b0b94e7b0d2
Author: michael loceff <loceff-h@senic.net>
Date: Tue Sep 11 16:46:02 2012 -0700

    added a dash, '-', as an allowable password character.

Michael@3A-VISTA64: ~/Documents/Visual Studio 2008/Projects/Assignment_1/Assignment_1 (master)
$ git merge origin/master
fatal: origin - not something we can merge

Michael@3A-VISTA64: ~/Documents/Visual Studio 2008/Projects/Assignment_1/Assignment_1 (master)
$ git merge origin/master
Updating 42f6803..18bef63
Fast-forward
 Assignment_1.cpp | 4 +++
 1 file changed, 3 insertions(+), 1 deletion(-)

Michael@3A-VISTA64: ~/Documents/Visual Studio 2008/Projects/Assignment_1/Assignment_1 (master)
$
```

In fact, even we programmers often avoid them since we have tons of GUI-based IDEs like Eclipse and Visual Studio to help us get the job done. Yes, **Git** and **GitHub** have GUIs for every platform, too. However, we are not going to use a GUI in this short overview, because there are certain concepts that are easy to miss if you don't experience the original, console-flavored version of the **Git** tools. So if you are wondering whether the **Git** experience is always as *texty* as it will seem in the next three pages, the answer is "no, but be happy." This is good for you.

G1.3 Installing Git

Step 1: Get Git.

Whether you are developing on your own locally, or working remotely with or without a team, you will need to download the **Git** application on your system. One way to do that is by using this link, created and hosted by **GitHub**:

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

Install it using all the default choices. They are the safest for your computer and most compatible for collaboration.

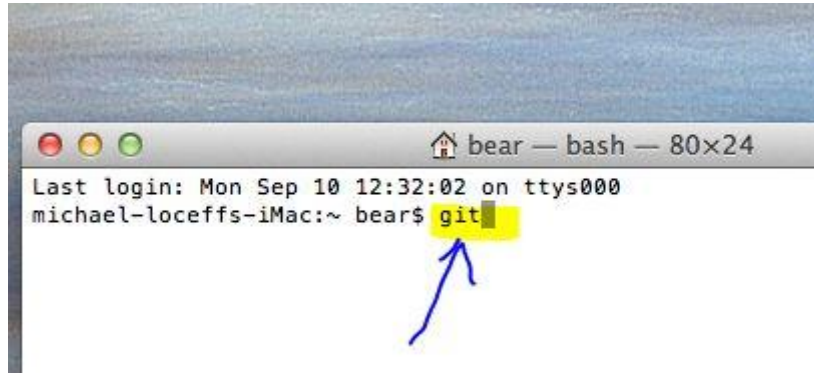
Step 2: Open a Git Terminal Window and Confirm The Install

On a Mac

Launch your **Terminal** program (under **Apps** → **Utilities**, but you should add it to your **Dock** for easy access).

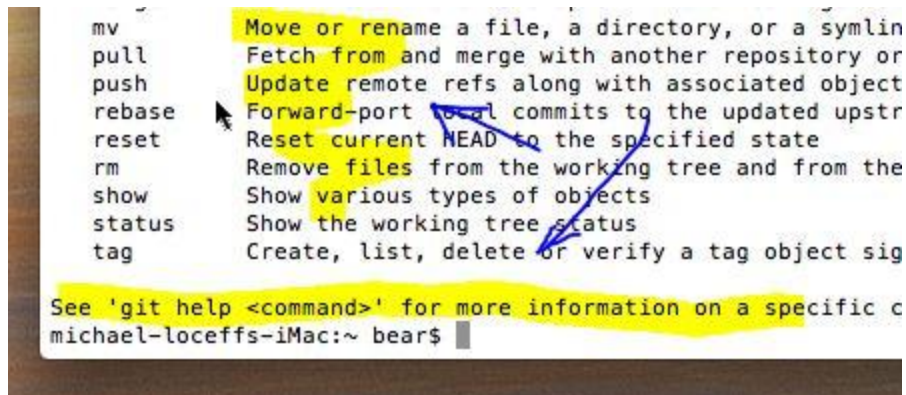


Once open, type "**git**" at the prompt:



A screenshot of a macOS Terminal window titled "bear — bash — 80x24". The window shows the login history "Last login: Mon Sep 10 12:32:02 on ttys000" and the prompt "michael-loceffs-iMac:~ bear\$". The word "git" has been typed at the prompt and is highlighted in yellow. A blue arrow points from the "git" text to the list of Git commands in the following block.

If correctly installed, you will see something like this:



A screenshot of a macOS Terminal window showing the output of the "git" command. The output lists various Git commands and their descriptions, with several lines highlighted in yellow. A blue arrow points from the "git" command in the previous block to the "rebase" command in this list. The list includes: "mv" (Move or rename a file, a directory, or a symlink), "pull" (Fetch from and merge with another repository or), "push" (Update remote refs along with associated object), "rebase" (Forward-port local commits to the updated upstr), "reset" (Reset current HEAD to the specified state), "rm" (Remove files from the working tree and from the), "show" (Show various types of objects), "status" (Show the working tree status), and "tag" (Create, list, delete or verify a tag object sig). Below the list, a line says "See 'git help <command>' for more information on a specific c". The prompt "michael-loceffs-iMac:~ bear\$" is visible at the bottom.

You can close **Terminal** now that we have confirmed that **Git** was correctly installed.

On Windows

You should open **Git Bash** (which will be on your desktop if your followed my advice and accepted the defaults during installation).



If correctly installed, you will see something like this:

```
grep      Print lines matching a pattern
init      Create an empty git repository or reinitialize
log       Show commit logs
merge     Join two or more development histories together
mv        Move or rename a file, a directory, or a symlink
pull      Fetch from and merge with another repository
push      Update remote refs along with associated objects
rebase    Forward-port local commits to the updated upstream
reset     Reset current HEAD to the specified state
rm        Remove files from the working tree and from the
show      Show various types of objects
status    Show the working tree status
tag       Create, list, delete or verify a tag object
See 'git help <command>' for more information on a specific
michael@MJL-SA-VISTA61:~$
```

The terminal window title bar shows 'cs_2a_assignment_soln_8.t'.

You can close **Git Bash** now that we have confirmed that **Git** was correctly installed.

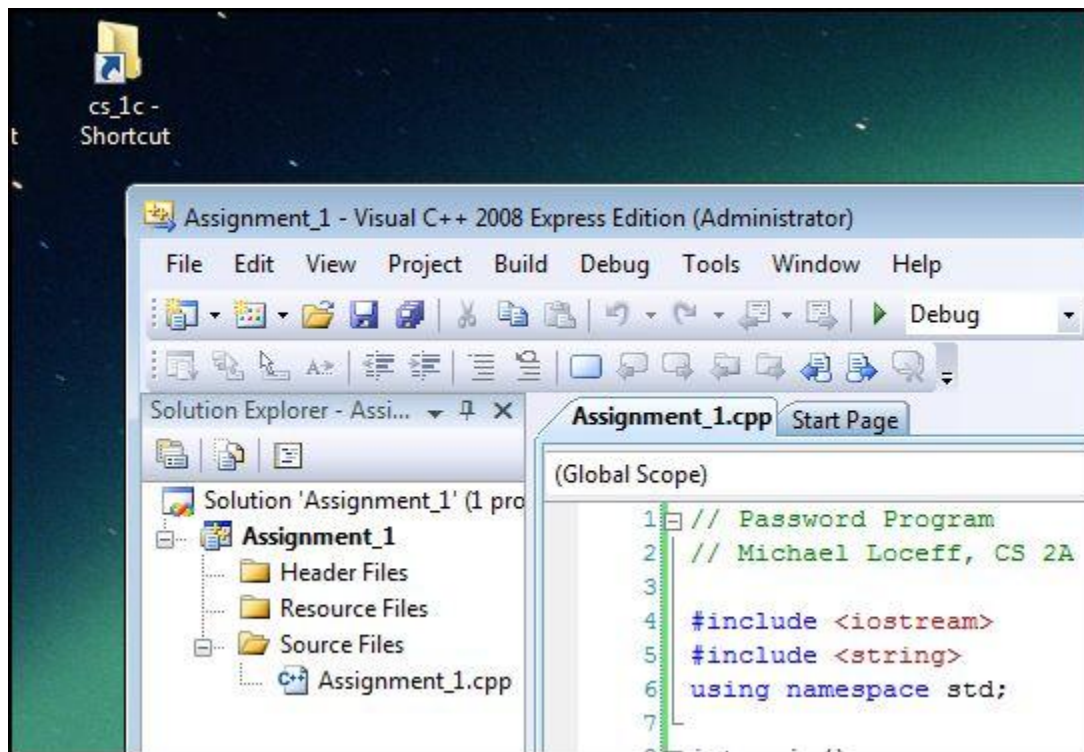
Section 2 - Using Git to Track a Project

G2.1 The Project Folder

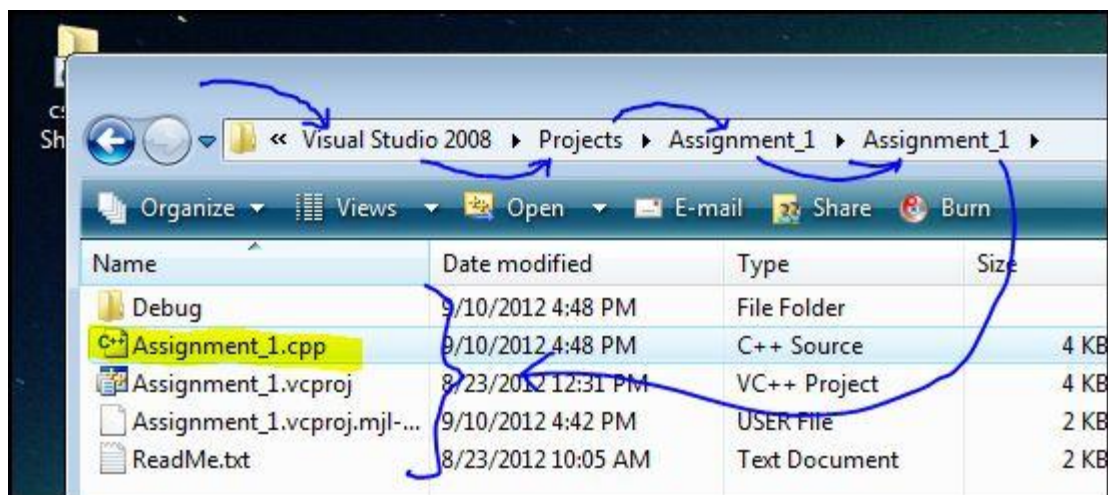
By using an IDE, we are often shielded from the details of our project. One of these details is the knowledge about a project's location within our computer's file system. Where is it? To answer that question, read the section below that pertains to your operating system and IDE.

Location of Project Folder On Windows/VC++

Here is a project I'm calling **Assignment_1**, as viewed from our IDE:



We can't quite tell where this lives on our system, so a little investigation is in order. On my computer, here is where I find the project's payload:



In other words, if one uses the default settings when installing Visual Studio, your project **Assignment_1** will be in the following path:

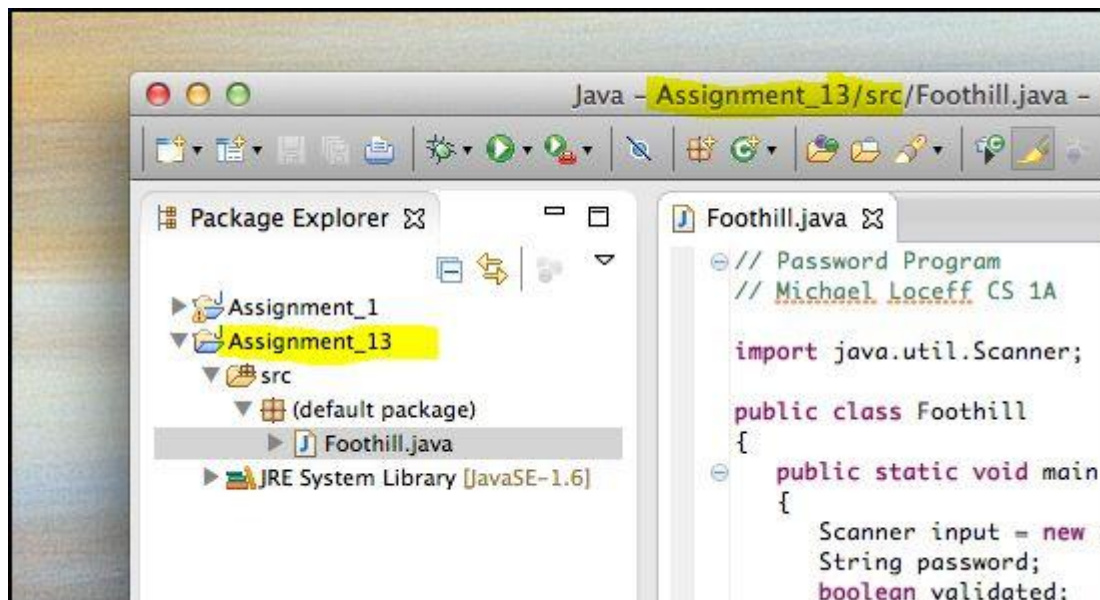
```
C:\Users\<YOU>\Documents\Visual Studio  
2008\Projects\Assignment_1\Assignment_1
```

Notice that there are two folders with the name **Assignment_1**. It is inside the second, or inner, **Assignment_1** folder that our project files actually live.

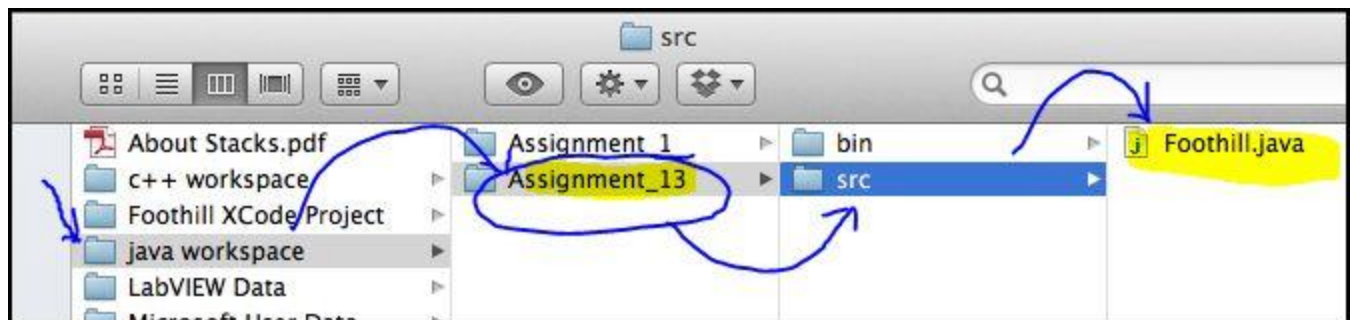
Find the location of any one of your VC++ projects and, make a note of the folder location, because you'll need it for the next step. It should be similar to mine, above.

Location of Project Folder On Mac/Eclipse

Here is a project I'm calling **Assignment_13**, as viewed from our IDE:



We can't quite tell where this lives on our system, so we search on our desktop. On my computer, here is where I find the project's payload:



In other words, if one uses the default settings when installing Eclipse, your project **Assignment_13** source files will be in the following path:

```
<YOU>/Documents/"java workspace"/Assignment_13/src
```

Notice that "**java workspace**" will be just "**workspace**" if you used the defaults when you installed and ran **Eclipse** the first time. Or, it might be "**c++ workspace**", or anything else you named it, if you were using C++ and named it such. Furthermore, the quotes surrounding "**java workspace**" will be needed later, if you have a folder that, like mine, has a space in its name. That's why you see them around "**java workspace**" but none of the other folders.

Find the location of any one of your Eclipse projects and, make a note of the folder location, because you'll need it for the next step. It should be similar to mine, above.

G2.2 Configuring Git

You will use the *git config* command now to create an identity that will be used to distinguish your changes from someone else's (like another team member). The following two commands (with your info used in place of Dr. Sheldon Cooper's) will accomplish this. Ignore the lines starting "**michael...**". These are just part of the **Bash** prompt on Windows. The command is typed after the \$.

```
michael@MJL-3A-VISTA64 ~  
$ git config --global user.name "sheldon cooper"  
  
michael@MJL-3A-VISTA64 ~  
$ git config --global user.email coopsheldonfox@ucla.edu  
  
michael@MJL-3A-VISTA64 ~  
$
```

There are other configuration options that we can ignore for now. These are the two important ones.

G2.3 The Repository

Navigating to the Project Folder

Using the command **cd** to change directory, we can navigate to our project folder. Start your **Git** console (using **Bash** on windows or **Terminal** on a Mac), and execute the command **cd ~**, which starts us off at our user home directory. I often follow this with the **pwd** command which tells me exactly where I am. This prevents me from getting lost:

```
$ cd ~
```

```
michael@MJL-3A-VISTA64 ~  
$ pwd  
/c/Users/michael
```

```
michael@MJL-3A-VISTA64 ~  
$
```

Now, using a series of **cd** commands, we follow the path toward our project. You can do this in one long command, but I will be taking one step-at-a-time to reduce the chance of a big mistake.

```
michael@MJL-3A-VISTA64 ~  
$ cd Documents
```

```
michael@MJL-3A-VISTA64 ~/Documents  
$ cd "Visual Studio 2008"
```

```
michael@MJL-3A-VISTA64 ~/Documents/Visual Studio 2008  
$ cd Projects
```

```
michael@MJL-3A-VISTA64 ~/Documents/Visual Studio 2008/Projects  
$ cd Assignment_1
```

```
michael@MJL-3A-VISTA64 ~/Documents/Visual Studio 2008/Projects/Assignment_1  
$ cd Assignment_1
```

```
michael@MJL-3A-VISTA64 ~/Documents/Visual Studio  
2008/Projects/Assignment_1/Assi  
gnment_1  
$
```

If you issue an **ls** command (list files and folders), you will see the files that you knew were there all along:

```
$ ls  
Assignment_1.cpp                                Debug  
Assignment_1.vcproj                             ReadMe.txt  
Assignment_1.vcproj.mjl-3a-vista64.michael.user
```

On a Mac, using an Eclipse/Java environment, the listing would look a little different because we would have initialized the repo from our **Assignment_13** project's **src** directory which has only one **.java** file in it (based on my single-file project):

```
michael-loceffs-iMacsrc bear$ ls  
Foothill1.java  
michael-loceffs-iMac:src bear$
```


Creating the Repository

The project folder is where the *repository*, or *repo*, will live. That's why we went there. We are going to create a *repo* by issuing the **git init** command, and thus create a new subdirectory named **.git**, within the project folder. **Git** uses the **.git** subdirectory for directing traffic in our project evolution -- it stores all the secret, but vital, data about our project's history. Let's issue the **git init** command, followed by an **ls** to list the files afterward.

```
$ git init
Initialized empty Git repository in c:/Users/michael/Documents/Visual Studio
2008/Projects/Assignment_1/Assignment_1/.git/

michael@MJL-3A-VISTA64 ~/Documents/Visual Studio
2008/Projects/Assignment_1/Assi
gnment_1 (master)
$ ls
Assignment_1.cpp          Debug
Assignment_1.vcproj      ReadMe.txt
Assignment_1.vcproj.mjl-3a-vista64.michael.user
```

That's interesting.

We don't see the new **.git** directory. The dot at the start of **.git**, makes it a hidden entity. If you issue the **ls** command with the **-a** (all) option, you will see it:

```
$ ls -a
.      Assignment_1.cpp          Debug
..     Assignment_1.vcproj      ReadMe.txt
.git   Assignment_1.vcproj.mjl-3a-vista64.michael.user
```

Side Note - The commands **ls**, **cd**, **pwd**, etc. are all Unix commands. So you are learning a little Unix as you study this tutorial.

Another detail you may have noticed is that the parenthetical, **(master)**, is displayed after the directory name (end of line that starts "**michael@MJL...**" two screen shots above). It means that this is going to the *master branch* which is the main project effort. Any experimental or beta efforts will sprout from this branch and get their own branch names, distinct from master (like **beta_3_47**, **hotfix_1**, **hotfix_2**, etc.). After all, the purpose of **Git** is to enable a lot of people to do a lot of things to a project, which will result in many different versions and experiments. There will always be a **master** branch which represents the main direction of the project.

G2.4 The Status of the Repo

Untracked vs. Tracked Files

Every file in the directory of the *repository* -- which is your project folder -- has a *status*. The most boring status is *Untracked*. These are files that we simply don't care about for tracking, but are still needed by our IDE. They may be important for our program -- the executables, IDE files, etc., but they are not human-readable and/or they just don't matter for tracking purposes. Put another way, these untracked files can usually be recreated from the more important source files which *are* tracked. So, if we give someone else the tracked files (via a **Git** checkout, for example) they will be able to reproduce all the other, untracked files.

The files we *want to track* are **.cpp** files for C++, **.java** files for Java, **.html** and **.css** files for web design, **.txt** files that have documentation, and so on.

When we create a new *repo*, no files are tracked. We can see this by issuing a **git status** command:

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       Assignment_1.cpp
#       Assignment_1.vcproj
#       Assignment_1.vcproj.mjl-3a-vista64.michael.user
#       Debug/
#       ReadMe.txt
nothing added to commit but untracked files present (use "git add" to track)
```

To start tracking a file or files, we use the **git add** command. We can add an individual file or a group of files, using the wildcard character, *:

```
$ git add ReadMe.txt
$ git add *.cpp
```

(The wildcard, also has its origin in Unix.) Nothing visible happens, but when we ask for a **git status**, we see the difference:

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   Assignment_1.cpp
#       new file:   ReadMe.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       Assignment_1.vcproj
#       Assignment_1.vcproj.mjl-3a-vista64.michael.user
#       Debug/
```

Those files moved up to the section called **Changes to be committed**. We only need to do this once. The files **Assignment_1.cpp** and **ReadMe.txt** are now going to be *tracked* for the weeks, months or years that the project lives (unless we manually remove them from the tracked files).

What happens if some of the files that we want to track, like **Foothill.java** or **math_tools.cpp**, are inside subfolders deeper than the directory we set as the repo home? That is not a problem. We don't have to explain such minutia to **Git**. If we want to add all **.cpp** or all **.java** files to those being tracked, we do so by using the same sort of command. For the Java project, when we do a **git add *.java**, **Git** would find the file, even if it were further down the directory tree.

Staged Files and "commit"

This brings us to another category of files: *staged*. When files were *added* in the last section, they were not placed in the repository. So far nothing from our project is in the *repo*. We moved them into a *staging area*, which means that they will be moved into the *repository* at the next **commit** action. A **commit** is an action that tells **Git** we are sure we want to make our recent changes more permanent; we want to officially add them to the *repo*. Doing so means that we will be able to return to the state of the project as it was in the staging area at the time of the **commit**. Sometimes this commit involves a change to only one line or a small bug-fix to the code. Other times, as in the current situation, it means we are adding an entire file full of new code that wasn't there before. Let's do it.

```
$ git commit -m "The original commit for the password project"
[master (root-commit) 3cb52d4] The original commit for the password project
2 files changed, 136 insertions(+)
create mode 100644 Assignment_1.cpp
create mode 100644 ReadMe.txt
```

The **-m** option indicates that we want to add a comment describing the significance of this commit. The string that followed it will be in the tracking documentation. If you don't use the **-m** options, a text editor will be opened for you, and you'll be able to put all sorts of documentation into that editor. When you close the editor, the commit action will proceed, automatically.

Let's look at the status now.

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       Assignment_1.vcproj
#       Assignment_1.vcproj.mjl-3a-vista64.michael.user
#       Debug/
nothing added to commit but untracked files present (use "git add" to track)
```

We see that the two *staged* files are not present in the status. Where are they? They are not in the untracked section, since they are being tracked. But neither are they in the staging area. They are checked-in safely, and everything is calm and quiet. This means that our current files in the VC++ or Eclipse IDE, which are being tracked, are identical to those in the repository. We can sleep well.

Updating the Project and Repository

What happens if we further edit one of the files? Say we make a change to the program. I'm going to change a line to require a slightly longer password (this is a small detail deep in the code of the password program we are tracking). Here is the line before my change:

```
// test for reasonable length
if (length < 6 || length > 15)
{
    cout << "Password must be between 6 and 15 characters.\n";
    continue;
}
```

and now I will make some changes, strengthening the requirement for passwords: they must be at least 7 characters long:

```
// test for reasonable length
if (length < 7 || length > 15)
{
    cout << "Password must be between 7 and 15 characters.\n";
    continue;
}
```

(I should be using a symbolic constant for the 6 and 7, so I don't have to change it in two places. -1 point for me!) I compile it, run it, fix a bug I introduced, recompile and run and I'm satisfied.

I save the project and quit my compiler. Now, let's go back to the **Git** console and check on the status:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   Assignment_1.cpp
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       Assignment_1.vcproj
#       Assignment_1.vcproj.mjl-3a-vista64.michael.user
#       Debug/
no changes added to commit (use "git add" and/or "git commit -a")
```

We can see that there are "changes not staged for commit." This is yet another state (and and the last one today) that one of our project files can be in: ***modified but not staged***. It simply means we are working on this file, but we are, for one reason or another, not ready to ***stage*** it for a ***commit***. Maybe we are working on a problem and have not resolved it yet. Or perhaps we are experimenting with a new feature and we have not completed the minimal functionality needed to warrant staging it. At any rate, once we are ready to stage this file, we issue the **git add** command again:

```
$ git add Assignment_1.cpp

michael@MJL-3A-VISTA64 ~/Documents/Visual Studio
2008/Projects/Assignment_1/Assi
gnment_1 (master)
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   Assignment_1.cpp
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       Assignment_1.vcproj
#       Assignment_1.vcproj.mjl-3a-vista64.michael.user
#       Debug/
```

The status shows that **Assignment_1.cpp** is now staged, waiting for a commit. Let's do that now:

```
$ git commit -m "Stenghtened length requirement for password to length 7."
[master 42f6803] Stenghtened length requirement for password to length 7.
 1 file changed, 2 insertions(+), 2 deletions(-)
```


Now if we check the status, all is quiet again.

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       Assignment_1.vcproj
#       Assignment_1.vcproj.mjl-3a-vista64.michael.user
#       Debug/
nothing added to commit but untracked files present (use "git add" to track)
```

Incidentally, we can ask **Git** to stop pestering us about the untracked files by creating and editing an **.ignore** file in our repository folder. But we only have time in this section for the bare essentials. As you can imagine, we have not begun to scratch the surface.

- How do we recover a previous state of the project?
- How do we remove a file from the staging area?
- How do we overwrite our mess, if we have made some unproductive changes, with a clean copy from the *repo*?
- How do we *branch* our repo so as to allow a beta version of the software while continuing to allow a master branch to hold the current version for hot bug fixes?
- How do we *merge* different contributors' work?
- What GUI apps are available to help automate some of these tedious command-line tasks?
- How do we *pull* and *push* our repo from/to a remote server?

For the purpose of this brief intro we will not be able to answer these questions today. However, I will touch on the answer to a couple of them in the next section about **GitHub**, the remote public repository used for secure tracking and collaboration.

Section 3 - GitHub and Remote Collaboration

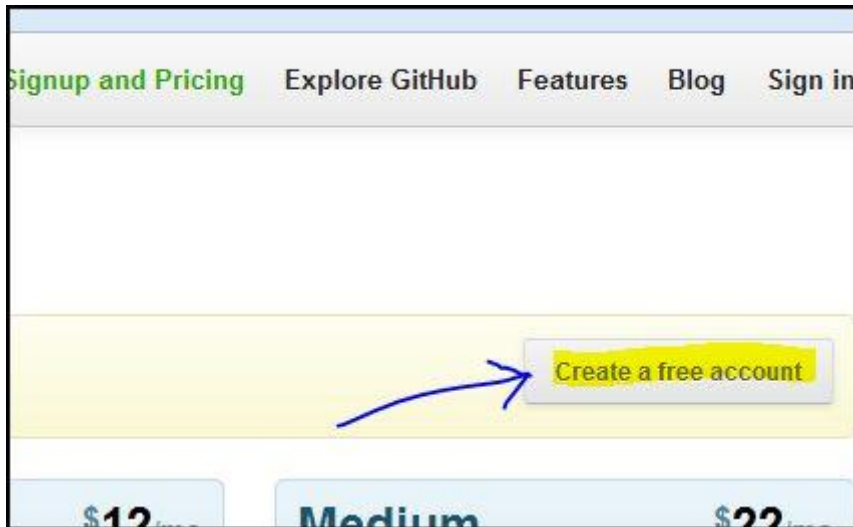
Unless you are already working for a company that has its own remote VCS, the best way to gain experience to remote tracking and project collaboration is by using the free public access offered by GitHub. The only thing to remember when you get a free **GitHub** account is that the code on it is publicly viewable and downloadable. (If you want privacy, you have to pay.) Others cannot modify your code on the server without your permission, but they can view, download, compile and modify it on their systems. In fact, this is the whole point.

G3.1 Signing Up

Go to

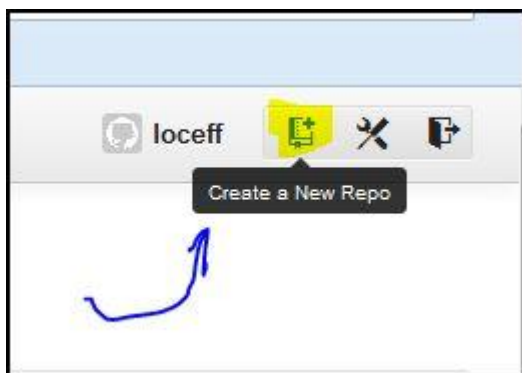
<https://github.com/>

and create your own account:

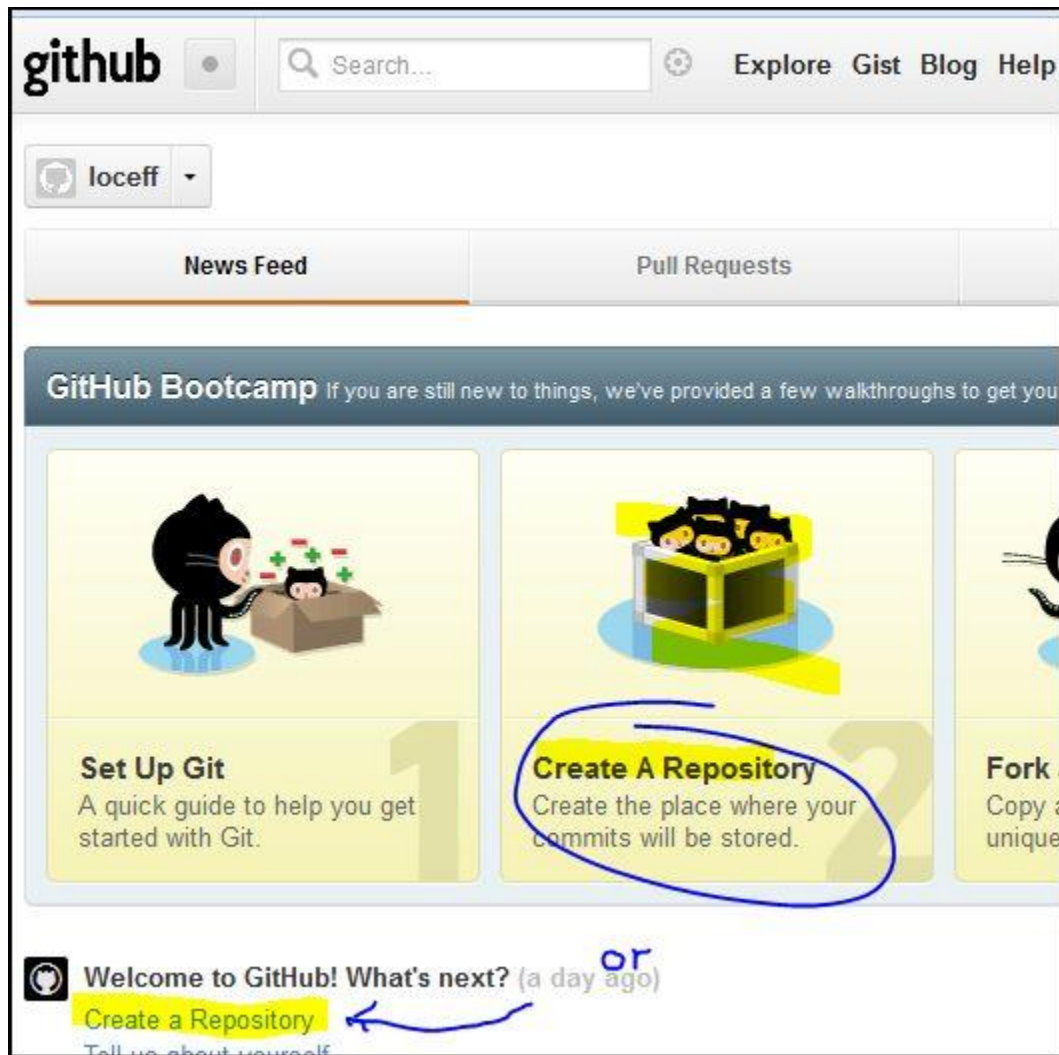


G3.2 Creating a Remote Repository

Next, you will create and name a repository for your project. While this is intended to hold the project we started in the last section, its name can be different. You can have multiple repositories, one for each project that you control (including projects that you *clone* from others and work on as a separate effort). For now, just create a single *repo* to get started. I'm creating one for the password program that I showed you in the class modules of the beginning C++ course.



By the way, you'll see some tutorials along the way. Below you find a little help on creating a repository and even setting up **Git** on your system.



Give the new **repo** a name (no spaces, please), and set it to be **public** so others, or even you on a system which is not yet set up, can **clone** it.

The screenshot shows the GitHub 'Create repository' form. The browser address bar displays 'https://secure1.anthe...'. The form has two main sections: 'Owner' and 'Repository name'. The 'Owner' dropdown is set to 'loceff'. The 'Repository name' text box contains 'password-project' and is highlighted in yellow. A green checkmark is visible to the right of the text box. Below this, a hint says 'Great repository names are short and memorable. Need inspiration? H...'. The 'Description (optional)' text box contains 'A learning example for CS students at Foothill College'. The 'Visibility' section has two radio buttons: 'Public' (selected and highlighted in yellow) and 'Private'. Below 'Public' is the text 'Anyone can see this repository. You choose who can commit.' Below 'Private' is the text 'You choose who can see and commit to this repository.' The 'Initialize this repository with a README' checkbox is checked. Below it is the text 'This will allow you to git clone the repository immediately.' and a dropdown menu for 'Add .gitignore: None'. At the bottom is a green 'Create repository' button, which is also highlighted in yellow. Three blue arrows point to the 'Repository name' text box, the 'Public' radio button, and the 'Create repository' button.

Latest Headlines <https://secure1.anthe...>

Owner **Repository name**

loceff ✓

Great repository names are short and memorable. Need inspiration? H...

Description (optional)

☒ **Public** Anyone can see this repository. You choose who can commit.

☐ **Private** You choose who can see and commit to this repository.

☒ **Initialize this repository with a README**
This will allow you to git clone the repository immediately.

Add .gitignore: **None**

Create repository

You don't have to do any of the optional things at the bottom of the window since you can take care of those on your local system if you wish. Once you click **Create repository**, you will see a screen with some advice for your local **Git** console. The first part tells you how to create a new **repo**, which you already did, so skip it. But the second part explains how to set your **Git** environment to know about and connect to this new online **repo**; it tells you the exact commands to issue in order to make a connection between your local project and the remote **GitHub repo** you just created.

Quick setup — if you've done this kind of thing before

Setup in Windows or **HTTP** **SSH** `https://github.com/loceff/password-project.git`

We recommend that every repository has a **README**, **LICENSE**, and **.gitignore**

Create a new repository on the command line

```
touch README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/loceff/password-project.git
git push -u origin master
```

Push an existing repository from the command line

```
git remote add origin https://github.com/loceff/password-project.git
git push -u origin master }
```

G3.3 Pushing to the Repo

Setting the Remote Repo

Meanwhile, back at the homestead, we can reopen our **Git** console (Terminal, Bash, etc.) and move to the directory of our project. Once there, we can first verify that we have no remote server set by issuing the git remote command:

```
$ git remote
```

```
michael@MJL-3A-VISTA64 ~/Documents/Visual Studio 2008/Projects/Assignment_
gnment_1 (master)
$
```


All that happened was that we got another (long) prompt which means there is no remote server set. (Remember, that long line "michael ..." is part of my prompt. I will cut it out of some of the following console snippets). We heed the advice of **GitHub** and execute a **git remote add** command. This will do two things:

1. tell our local **Git** repository where the remote server is, and
2. give the remote server a short name, typically "**origin**."

Notice that **GitHub**'s suggested instruction which I bracketed in blue, above, provide the exact URL that the command will need, so they are very useful. We do it:

```
$ git remote add origin https://github.com/loceff/password-project.git
```

After this, a repeat of the **git remote** command verifies that something is now there:

```
$ git remote
origin
```

Using the *verbose* option, **-v**, tells us the association between the remote alias and the full URL:

```
michael@MJL-3A-VISTA64 ~/Documents/Visual Studio 2008/Projects/Assignment_
gnment_1 (master)
$ git remote -v
origin https://github.com/loceff/password-project.git (fetch)
origin https://github.com/loceff/password-project.git (push)
```

The Initial Push

We are ready to push our repository information to "origin", our remote host. This is the purpose of the **push** command. The meaning of the word **master** is to tell **Git** which of our **branches** we want to **push**; we only have one (**master**) and don't even know how to create others, yet, so it seems superfluous, but it's needed:

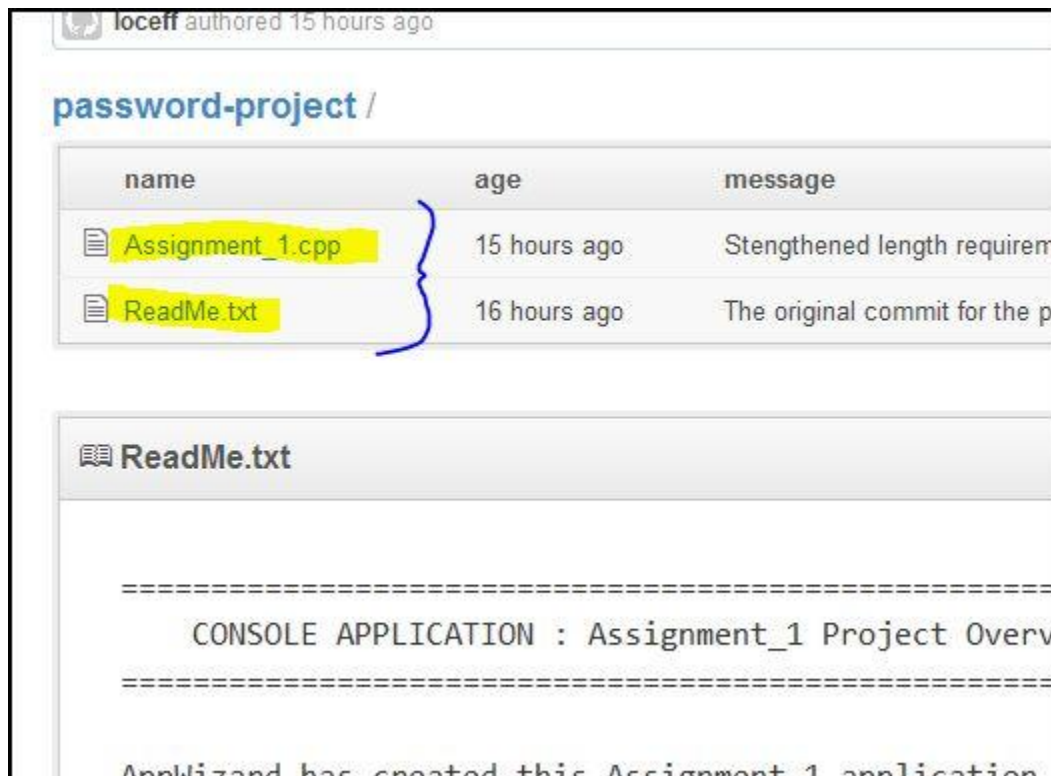
```
$ git push -u origin master
Username for 'https://github.com': <probably your email>
Password for <your account >: <type password>
Counting objects: 7, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 2.03 KiB, done.
Total 7 (delta 1), reused 0 (delta 0)
To https://github.com/loceff/password-project.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

In this case, we had to provide an ID and password for the **push**. Had we set up some security parameters, we could have avoided this step, not something by which we want to be distracted at this time.

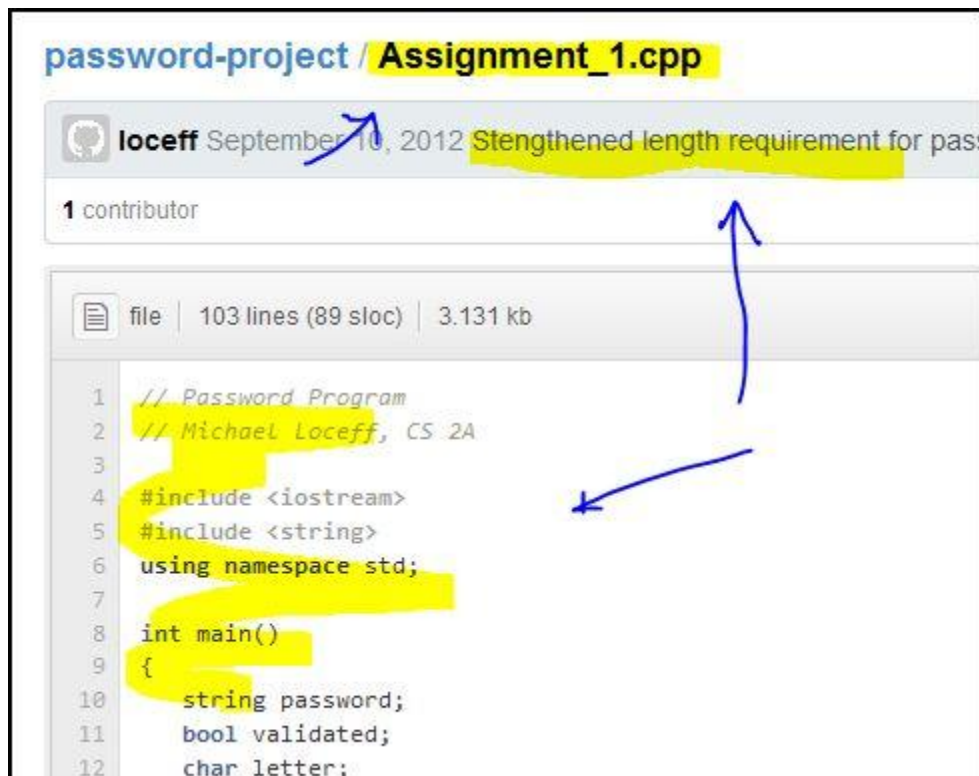
We can go back to our **GitHub** web page and see that the tracked files are there. Click on the project.



We see a list of familiar files -- very promising:



And if we click on the **.cpp** file, we can view its contents.



G3.4 Fetching the Repo

During the normal course of project workflow, you will be sitting down to your system and fetching the current snapshot of the repo as it is at that moment, thus assuring that you will be working on the most current version. This is known as a **fetch**. (Sometimes you will hear it called a **pull**, which is a more powerful **fetch** that does a couple things at once. While we will talk of **pushes** and **pulls**, informally, sometimes we say **pull** when we will actually be doing the more cautious **fetch**. I won't demonstrate the **pull** command in this tutorial -- it is essentially a **fetch** + **merge**.)

The command is **git fetch origin**:

```
michael@MJL-3A-VISTA64 ~/Documents/Visual Studio 2008/Projects/Assignment_1/Assignment_1 (master)
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       Assignment_1.vcproj
#       Assignment_1.vcproj.mjl-3a-vista64.michael.user
#       Debug/
nothing added to commit but untracked files present (use "git add" to track)
```

A status showed us that nothing changed, which is no surprise since we just did a push and no one else made any changes to the repo.

Merging

But if someone else had pushed changes, we would see the following during the fetch:

```
$ git fetch origin
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From https://github.com/loceff/password-project
 42f6803..18bef63  master    -> origin/master
```

A status will also confirm we have a mismatch between what we just fetched and our local files:

```
$ git status
# On branch master
# Your branch is behind 'origin/master' by 1 commit, and can be fast-
forwarded.
#
# Untracked files:
#   (use "git add ..." to include in what will be committed)
#
#       Assignment_1.vcproj
#       Assignment_1.vcproj.mjl-3a-vista64.michael.user
#       Debug/
nothing added to commit but untracked files present (use "git add" to track)
```

There are now two masters -- the one that is in *our* project, based on *our* latest changes, and the one I just fetched, which we are told is "ahead" of our ("*Your branch is behind origin/master by 1 commit*"). One way to see what changed is to ask for a **log** of the difference between the two. Let's use the **git log** command, and tell it to show us the log notes between my master (the older snapshot on my system), and the origin/master (the newer snapshot on **origin** that we have downloaded, but not merged into our project).

Where is this unmerged origin/master? We can't see it within our project directory: that's got our old files in it. Answer: it is in the .git subdirectory, packed away in Git's secret ninja format. It is actually local now, but not visible to us. In fact, aside from the **push** and **fetch**, all these **commits**, **adds**, and **merges** that we do in **Git** are done off-line. We can be in an airplane or at the cabin and continue to work "collaboratively."

Here the term **master** refers to my **master branch**, the only branch I am using on my local system. The term **origin/master** refers to the **master branch** on the **remote server**, which we are calling **origin**. So, let's issue a log command to see what the difference between the two masters is:

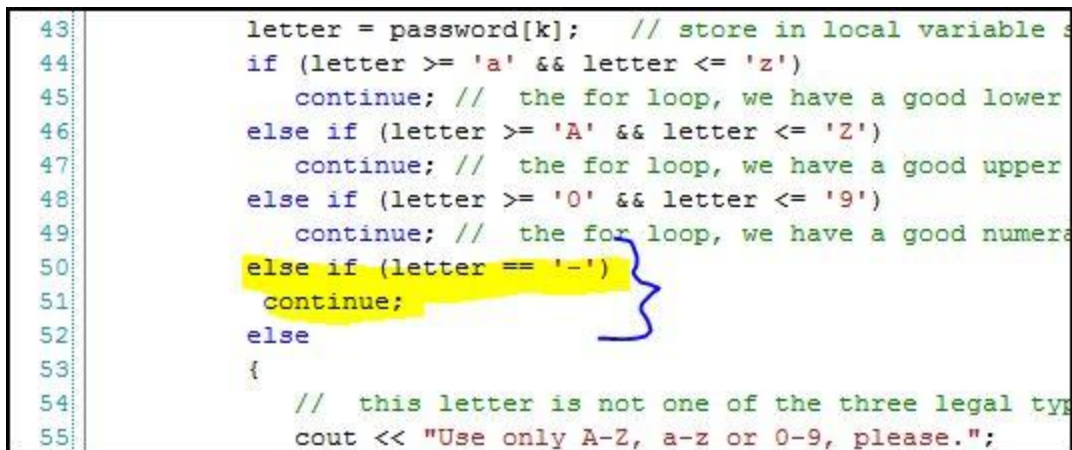
```
$ git log master..origin/master
commit 18bef637004ecd6374a57a5262205b0b94e7b0d2
Author: dr. sheldon cooper <coopshelfox@ucla.edu>
Date: Tue Sep 11 16:46:02 2012 -0700
```

added a dash, '-', as an allowable password character.

I see. Dr. Sheldon Cooper changed the source so as to make a dash legal for password characters. We can use the **diff** tool to see the exact lines, but for this tutorial, let's just accept it and merge that change into our repo so as to get our files updated. This done with the **merge** command:

```
$ git merge origin/master
Updating 42f6803..18bef63
Fast-forward
 Assignment_1.cpp | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

We have just merged the remote **origin/master** into our local files. Let's open the project to see if the change is there:



```
43 letter = password[k]; // store in local variable s
44 if (letter >= 'a' && letter <= 'z')
45     continue; // the for loop, we have a good lower
46 else if (letter >= 'A' && letter <= 'Z')
47     continue; // the for loop, we have a good upper
48 else if (letter >= '0' && letter <= '9')
49     continue; // the for loop, we have a good numerals
50 else if (letter == '-')
51     continue;
52 else
53 {
54     // this letter is not one of the three legal types
55     cout << "Use only A-Z, a-z or 0-9, please.";
```

Yes -- this was not part of our last source, so the new code has been incorporated into our **master** branch.

This is one of the main uses of a CVS like **Git**. It enables us move forward, collaboratively, in a systematic way. There are ways to return to previous snapshots of the project, undo changes, and be working on multiple versions of the same project in parallel.

G3.5 Cloning a Repo

If you were not the originator of the project, or you are on a computer that has no **repo** for the project yet, rather than doing a **git init**, you would execute a **git clone**. One way to do this is to

1. create a small ***Hello World!*** project,
2. get it to compile and run just to make sure you have all your IDE files in place and working,
3. delete the main source file for the ***Hello World!*** ("Foothill.java" or "Hello World Proj.cpp", say) because the files we "pull" down will contain all the sources we need, and these existing ones will just get in the way,
4. from our console (Terminal or Bash) window, navigate to the directory where our sources need to live, and
5. execute **clone** as shown below.

```
$ git clone git://github.com/loceff/password-project.git
```

This creates a new repo for us with the files already in place. We can fire up our IDE to make sure the files are visible, and try to compile and run them. Assuming all goes well, we are now in position to start developing. We can push our changes to the server and become part of the development team. (We need to have push privileges or have the account information of the remote repo in order for the pushes to work.)

G3.6 More Information

This has been a brief intro to version control systems using **Git** and **GitHub**. It took us on a side trip from our main topics, but it's important that you be aware of this institution, since it heavily influences software development these days. For more information you can look at the many (better?) tutorials on the web.

- For a solid **Git** tutorial, I recommend: <http://git-scm.com/book>
- There is a good ***Set Up Git*** tutorial as well as a useful FAQ at the **GitHub Help Center**: <https://help.github.com/>
- For the latest ***GUI GitHub clients*** for Mac, Windows, Eclipse, etc. go to <https://github.com/> and at the bottom of the page, see "Clients"

Section 3 - A GUI Input Example

4B.3.1 Your High School

Let's now retool our Grade School program as a baby GUI:

After the user answers the question and clicks **OK**, it displays something like this:

The program uses only what we have learned up to this point. Copy and run it in your own Java IDE:

```
import javax.swing.*;

public class Experiment_2
{
    public static void main(String[] args)
    {
        // declare some string variables
        String strUserAnswer;
        String strQuestion;
        String strFinalStatement;

        // Prepare a question for the user
        strQuestion = new String("What high school did you attend?");

        // Show the user the question and get a response
        strUserAnswer = JOptionPane.showInputDialog(strQuestion);

        // build up a final statement using the user's input
        strFinalStatement = "Really? You went to " + strUserAnswer + "? ";
        strFinalStatement = strFinalStatement + " You must know Don Fagenson!";

        // show the final output
        JOptionPane.showMessageDialog(null, strFinalStatement);
    }
}
```

Why "Baby"?

I called this a "baby" GUI. It's not a true GUI, because it works more like a console app than a graphical interface. A real GUI needs to have a window which *stays open* and which has various areas positioned within it: an input area, some labels, a result area, etc. We do not want windows disappearing and reappearing every time we take an action in a GUI, and that's what these baby GUIs have been doing. Such behavior is really nothing more than a console app masquerading as a GUI.

Note:

Do not submit a JOptionPane GUI as a homework assignment. I don't accept them. I am presenting them to you for your own personal use and as an intro to real GUIs.

I will present a couple optional C-modules in this course which describe how to build true GUIs. You will be allowed to substitute a GUI app version of one of your homework assignments in place of the console version.
