Git is a source code version control system. Such a system is most
useful when you work in a team, but even when you're working alone,
it's a very useful tool to keep track of the changes you have made to
your code.

In this class, you are required to use Git for doing your homework
assignments (we call them labs). You will use Git not only for coding
your labs, but also for retrieving the skeleton code I provide,
submitting your code, and downloading solutions.

This tutorial covers not only the basic git operations that you need,
but also the workflow between you, me, and the TAs--from my
preparation of an assignment all the way to the grading of your
submissions by the TAs. Even if you are already familiar with git,
you may find the description of the workflow interesting.

Set EDITOR environment variable

Type "echo $EDITOR". If the shell does not respond with the name of
your editor--vim, emacs, or nano--add the following line at the end of
the .bashrc file in your home directory:

    export EDITOR=your_choice_of_editor

Log out and log in again. Type "echo $EDITOR" to make sure that your
modification to .bashrc has taken effect.

Configure your git environment

Tell git your name and email:

    git config --global user.name "Your Full Name"
    git config --global user.email your_uni@columbia.edu

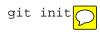git stores this information in ~/.gitconfig

Creating a project

Let's create a new directory, ~/tmp/test1, for our first git project.

    cd
    mkdir tmp
    cd tmp
    mkdir test1
    cd test1

Put the directory under git revision control:

    git init

If you type ll (I'll assume that ll is an alias for ls -alF), you will
see that there is a .git directory. The git repository for the
current directory is stashed in the .git directory.

Let's start our programming project. Write hello.c with your editor:

```
#include <stdio.h>
int main()
{
    printf("%s\n", "hello world");
    return 0;
}
```

Compile and run it:

```
gcc hello.c
./a.out
```

Let's see what git thinks about what we're doing:

```
git status
```

The git status command reports that hello.c and a.out are "Untracked"
We can have git track hello.c by adding it to the "staging" area (more
on this later):

```
git add hello.c
```

Run git status again. It now reports that hello.c is "a new file to
be committed." Let's commit it:

```
git commit
```

Git opens up your editor for you to type a commit message. A commit
message should succinctly describe what you're committing in the first
line. If you have more to say, follow the first line with a blank
line, and then with a more through multi-line description.

For now, type in the following one-line commit message, save, and exit
the editor.

```
Added hello-world program.
```

Run git status again. It now reports that only a.out is untracked.
It has no mention of hello.c. When git says nothing about a file, it
means that it is being tracked, and that it has not changed since it
has been last committed.

We have successfully put our first coding project under git revision
control.

Modifying files

Modify hello.c to print "bye world" instead, and run git status. It
reports that the file is "Changed but not updated." This means that

the file has been modified since the last commit, but it is still not
ready to be committed because it has not been moved into the staging
area. In git, a file must first go to the staging area before it can
be committed.

Before we move it to the staging area, let's see what we changed in
the file:

    git diff

Or, if your terminal supports color,

    git diff --color

The output should tell you that you took out the "hello world" line,
and added a "bye world" line, like this:

    -    printf("%s\n", "hello world");
    +   printf("%s\n", "bye world");

We move the file to the staging area with git add command:

    git add hello.c

In git, "add" means this: move the change you made to the staging
area. The change could be a modification to a tracked file, or it
could be a creation of a brand new file. This is a point of confusion
for those of you who are familiar with other version control systems
such as Subversion.

At this point, "git diff" will report no change. Our change--from
hello to bye--has been moved into staging already. So this means that
"git diff" reports the difference between the staging area and the
working copy of the file.

To see the difference between the last commit and the staging area,
add "--cached" option:

    git diff --cached

Let's commit our change. If your commit message is a one-liner, you
can skip the editor by giving the message directly as part of the git
commit command:

    git commit -m "changed hello to bye"

To see your commit history:

    git log

You can add a brief summary of what was done at each commit:

    git log --stat --summary

Or you can see the full diff at each commit:

    git log -p

And in color:

```
git log -p --color
```


The tracked, the modified, and the staged

A file in a directory under git revision control is either tracked or
untracked. A tracked file can be unmodified, modified but unstaged,
or modified and staged. Confused? Let's try again.

There are four possibilities for a file in a git-controlled directory:

1) Untracked

    Object files and executable files that can be rebuilt are usually
    not tracked.

2) Tracked, unmodified

    The file is in the git repository, and it has not been modified
    since the last commit. "git status" says nothing about the file.

3) Tracked, modified, but unstaged

    You modified the file, but didn't "git add" the file. The change
    has not been staged, so it's not ready for commit yet.

4) Tracked, modified, and staged

    You modified the file, and did "git add" the file. The change has
    been moved to the staging area. It is ready for commit.

The staging area is also called the "index".

Other useful git commands

Here are some more git commands that you will find useful.

To rename a tracked file:

```
git mv  old-filename  new-filename
```

To remove a tracked file from the repository:

```
git rm  filename
```

The mv or rm actions are automatically staged for you, but you still
need to "git commit" your actions.

Sometimes you make some changes to a file, but regret it, and want to
go back to the version last committed. If the file has not been
staged yet, you can do:

```
git checkout -- filename
```

If the file has been staged, you must first unstage it:

```
git reset HEAD  filename
```

There are two ways to display a manual page for a git command. For
example, for the "git status" command, you can type one of the
following two commands:

```
git help status
man git-status
```

Lastly, "git grep" searches for specified patterns in all files in the
repository. To see all places you called printf():

```
git grep printf
```


Cloning a project

You created a brand new project in the test1 directory, added a file,
and modified the file. But more often than not, a programmer starts
with an existing code base. When the code base is under git version
control, you can *clone* the whole repository. This is in fact what
you will do to start your lab assignments from my skeleton code.

Let's move up one directory, clone test1 into test2, and cd into the
test2 directory:

```
cd ..
git clone test1 test2
cd test2
```

Type ll to see that your hello.c file is cloned here. Moreover, if
you run "git log", you will see that the whole commit history is
replicated here. "git clone" not only copies the latest version of
the files, but also copies the entire repository, including the entire
commit history. After cloning, the two repositories are
indistinguishable.

Let's make some changes--and let's be bad. Edit hello.c to replace
"printf" with "printf%^&", save and commit:

```
vim hello.c
git add hello.c
git commit -m "hello world modification - work in progress"
```

Now run "git log" to see your recent commit carrying on the commit
history that was cloned. If you want to see only the commits after
cloning:

```
git log origin..
```

Of course you can add -p and --color to see the full diff in color:

```
git log -p --color origin..
```

Let's make one more modification. Fix the printf, and perhaps change
the "bye world" to "rock my world" while we're there.

```
vim hello.c
git add hello.c
git commit -m "fixed typo & now prints rock my world"
```

Run "git log -p --color origin.." again to see the two commits you
have made after cloning.

Generating a patch set

You can save the full details of everything you did after cloning with
the following command:

```
git format-patch --stdout origin > mywork.mbox
```

If you open mywork.mbox with your editor (be careful not to modify the
file), you will see that the file contains the full diffs of all
commits you made after cloning. A diff is also called a "patch". The
mywork.mbox file, therefore, contains a set of patches, one for each
commit after cloning.

When you were looking at the mywork.mbox file in your editor, you
might have noticed that each patch kinda looks like an email message.
That's because they are. In fact, the mywork.mbox file is in the UNIX
mailbox format (hence the extension .mbox), so you can view it using
an email application.

You can use mutt, a terminal-based email app, to open this file as if
it were your mailbox. Type 'q' to exit out of mutt.

```
mutt -f mywork.mbox
```

If you want to read the file with the diffs in color, run the
following command. Use arrow keys to scroll and type 'q' to exit.

```
cat mywork.mbox | colordiff | less -R
```

This patch file is in fact what you submit when you complete your lab
assignment. You will first clone my repository that contains some
skeleton code, add and modify files to complete the lab, and then
generate and submit a patch file that contains all your commits since
you cloned my repository. I will supply a script that automates the
patch generation and submission.

Applying the patch set

The TAs will apply your patch to my repository (the one you started
with by cloning it) to recreate your repository at the time of your
submission, and grade your work.

Let's go through that process. Move up one directory and clone test1
into test3:

```
    cd ..
    git clone test1 test3
    cd test3
```

Run "git log" to verify that you have the commits made in test1, but
not the ones made in test2.

Now let's bring in the commits made in test2 by applying the patch
file that you generated in test2:

```
    git am ../test2/mywork.mbox
```

You should see the following messages:

```
    Applying: hello world modification - work in progress
    Applying: fixed typo & now prints rock my world
```

Run "git log" to verify that now you have all the commits you made
both in test1 and test2.

Adding a directory into your repository

After the lab deadline, I will publish the solution by adding a
"solution" subdirectory to my repository. Let's simulate that
process. Go into the original test1 directory, and make the solution
subdirectory and create two files in it:

```
    cd ../test1
    mkdir solution
    cd solution

    cp ../hello.c .
    echo 'hello:' > Makefile
```

Type ll to see that two files--Makefile and hello.c--have been created
in the solutions directory. (hello.c was copied from the parent
directory, and Makefile was created directly on the command line using
the echo command. BTW, the Makefile contains a single line, "hello:".
Can you see why this is a legitimate Makefile?)

Now, move out of the solution directory, and git add & git commit the
solution directory:

```
    cd ..
    git add solution
    git commit -m "added solution"
```

Note that "git add solution" stages all files in the directory.

Pulling changes from a remote repository

Once you hear that the lab solution is available, you will want to
retrieve it and take a look. You do that by "pulling" the changes in
```

my repository into your repository.  Let's pull the changes we just
made in test1 into test2:

```
cd ../test2/
git pull
```

The "git pull" command looks at the original repository that you
cloned from, fetches all the changes made since the cloning, and
merges the changes into the current repository. You now have the
solution right in your repository.

Learning more about git

This tutorial covers everything you need to do your lab assignments.

Git is an extremely powerful tool and a beautifully designed piece of
software. If you want to learn more about it, start with the official
git tutorial:

```
man gittutorial
```

There are pointers to further documentations at the end of the
tutorial.

The documentation page of the Git web site has many links as well:

```
http://git-scm.com/documentation
```