Using Your Linux Account

Mattox Beckman

January 23, 2012

Contents

1   Introduction

Welcome!

As part of this course, you were given an account on my machine `dijkstra.cs.iit.edu`, or `dijkstra` for short. The machine runs Linux. The purpose of this document is to give you the information you need to use your account effectively.

You may also have been given an account on `bitbucket`. This is a second account, and we will discuss that in section 5.

2   First Login

If you don't want to use `dijkstra`, you don't have to; you are welcome to use your own machine. This will be easiest if you have a linux machine. If you have a Mac, you should be able to do everything here as well, since it runs a Unix based OS. Windows users may have a more difficult time. It is possible to install the tools you need to run natively under Windows, but with that amount of effort, you might as well have learned Linux instead. To use Linux under Windows, the easiest way is to use the Ubuntu Windows Installer. Ask your favorite search engine for the link.

An unfortunate thing about Linux is that you have to learn quite a bit at the very beginning in order to get started. But once you have gained some proficiency, you will be very productive.

2.1   Initial Account Information

When you first get an account on the machine, you will get an email like this one.

```
1  This email is to inform you of your linux account.
2
3  The machine is named dijkstra.cs.iit.edu.
4
5  Your username is USERNAME.
6
7  Your password will arrive in a message just after or before this one.
8  The subject will be 'dijkstra', and the body of the message will be
9  the password, a string of 8 hexadecimal digits.
10
11 That's not the most secure way to deliver a password, so you will need
12 to log onto your account and change it within the next few days, or
13 else the account will be suspended.
14
15 Enjoy!
16 - Mattox's account creation program
```

The word USERNAME will be replaced by something that presumably is related to your actual name. If you have an IIT email address, then your username is most likely the same.

You will then get another email with the subject "dijkstra". In it will be something like this:

```
1  dea0667c
```

This is your initial password. The account creation program generated it randomly. It will only be good for one use.

2.2   Secure Shell

To log on, you will need to use the *secure shell protocol*, also called SSH. There are several ways to do this.

The fastest way is to use a Java-enabled web browser. Simply go to the URL http://dijkstra.cs.iit.edu/ssh. On that page is a Java applet that implements secure shell.

For those of you who use Windows, there are programs such as Putty or Powerterm that implement SSH. There are also Java-based emulators out there. If you search for one, be sure to find one that uses protocol 2.

If you use Macintosh, Cygwin, or Linux, you can use the command line. You will type something like

```
1  ssh username@dijkstra.cs.iit.edu
```

Let's try it!

2.3   First Connection

When you connect for the first time, you will need to change your password. The screen will look something like this:

```
 1 │ username@dijkstra.cs.iit.edu's password:
 2 │ You are required to change your password immediately (root enforced)
 3 │ Linux dijkstra.cs.iit.edu 2.6.24-19-generic #1 SMP Fri Jul 11 23:41:49 UTC 2008 i686
 4 │
 5 │ The programs included with the Ubuntu system are free software;
 6 │ the exact distribution terms for each program are described in the
 7 │ individual files in /usr/share/doc/*/copyright.
 8 │
 9 │ Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
10 │ applicable law.
11 │
12 │ To access official Ubuntu documentation, please visit:
13 │ http://help.ubuntu.com/
14 │ WARNING: Your password has expired.
15 │ You must change your password now and login again!
16 │ Changing password for username.
17 │ (current) UNIX password:
18 │ Enter new UNIX password:
19 │ Retype new UNIX password:
20 │ passwd: password updated successfully
21 │ Connection to dijkstra.cs.iit.edu closed.
```

You will type in a password four times. Note well, *you will not see the cursor move or get any feedback as you type in your password*. This is standard behavior on unix systems.

The first time you type in your password is on line 1. This is for your initial login. The password is set for single-use, so the machine will automatically run the password change program. This program requires you to type in your old password, which you will do on line 17. On the next line you will create a new password, and then on line 19 you will type it again to verify that there were no mistakes.

Once that is done, you will be rudely logged out. Simply log in again, this time with your new password.

2.4   First Real Login

The first time you log in with your new password, you will get the welcome screen. It will have a bunch of stuff on it that you can ignore for the moment—we'll explain it more later. It should look something like this.

```
 1 │ Last login: Fri Sep  5 09:41:28 2008 from adsl-68-20-176-127.dsl.chcgil.ameritech.net
 2 │
 3 │ KeyChain 2.6.8; http://www.gentoo.org/proj/en/keychain/
 4 │ Copyright 2002-2004 Gentoo Foundation; Distributed under the GPL
 5 │
 6 │  * Initializing /home/students/all/username/.keychain/dijkstra.cs.iit.edu-sh file...
 7 │  * Initializing /home/students/all/username/.keychain/dijkstra.cs.iit.edu-csh file...
 8 │  * Initializing /home/students/all/username/.keychain/dijkstra.cs.iit.edu-fish file...
 9 │  * Starting ssh-agent
10 │  * Adding 1 ssh key(s)...
```

```
11  Identity added: /home/students/all/username/.ssh/id_rsa
12  (/home/students/all/username/.ssh/id_rsa)
13
14  username@dijkstra:~$
```

2.5 You're in!

You now have access to your Linux account. The next three sections tell you what you need to know to make good use of it. Section 3 introduces some basic Linux commands. Feel free to skip this section if you are already familiar with Linux or Unix. Section 4 explains[1] secure shell secret keys. These give you more flexibility logging in, and also give you access to the version control system, git, which is explained in section 5. The remaining sections are optional, but will help you do more advanced things.

3 Basic Linux Commands

The paradigm of Unix and its variants is the *command line interface* (CLI). You interact with your computer using an interpreter called a *shell*. There are many different shells you can choose from to meet your needs. The one we will use in class is called `bash`.[2]

In the previous section you saw how to log on. At the bottom of the screen you saw something like this:

```
1  username@dijkstra:~$
```

This line is called the *prompt*. Prompts are highly customizable; the one on my own computer shows the time and date, and the name of the computer I'm on at the moment. The one you see here shows the name of the user (`username` in this case), the short name of the machine (`dijkstra`), and the current directory[3] (`~`, which is an abbreviation for the user's home folder).

The last character of the prompt contains an important piece of information. If it is a `%` or `$`, then you are logged into a user account. If it is a `#`, then you are logged into the super-user account. The super-user is also called *root*. This kind of user has full administrative rights on the machine. You should not expect to get to see this prompt on `dijkstra`!

3.1 Example Files

Your account starts off with nothing in it, which is kind of boring, so to get started, type the following commands. Remember that `%` is the prompt; don't type that. Do type in the space and the period that are at the end, though!

```
1  username@dijkstra:~$ cp -r /home/examples/linux .
```

Don't worry about what this command does just yet; we will explain it soon.

---

[1]Well, mentions, actually.

[2]Some history: one of the early shells was called `bsh`, for Bourne shell. So the GNU people named this one the "Bourne Again Shell," though in fact it is based on `ksh` (Korn shell) and `csh` (C shell). Unix programmers seem to like puns.

[3]On Windows machines, directories are often called *folders*.

3.2 Listing Files

Probably the most-used command is `ls`. Standing for *list*, it lists the files in the current directory. Try it now. On your account, it will probably look something like this:

```
1 username@dijkstra:~$ ls
2 linux
```

There may be other files; you can ignore them for now. The `linux` file is a actually a directory, the equivalent of a folder in Windows.

3.3 Directories

When you are working on the command line, there is a concept called the "current directory" or "working directory". This directory is the target of all commands that you type, and when talking about it you will say that you are "in" that directory, as if it were a physical location. You can find out what directory you are in by using the `pwd` (print working directory) command:

```
1 username@dijkstra:~$ pwd
2 /home/students/all/username
```

In this example account, the user happens to be in the directory `/home/students/all/username`. It is really five directories nested within each other. The top-level directory is called simply `/`, and is pronounced *root*. Within the root directory is a subdirectory called `home`, and within that is another subdirectory called `students`. The directories `all` and `username` are similarly nested.

To change to a new directory, you use the command `cd` (for *change directory*). To change to the `linux` directory, type `cd linux`. When you type `pwd`, it shows you the new location.

```
1 username@dijkstra:~$ pwd
2 username@dijkstra:~$ cd linux
3 username@dijkstra:~/linux$ pwd
4 /home/students/all/username/linux
```

Notice how the prompt changes as you switch directories.

There are a few special directory names.

- `.`, a single period, stands for the current directory.

- `..`, two periods, stands for the parent directory. At the moment, `.` is `/home/students/all/username/linux`, and `..` is `/home/students/all/username`.

- `~`, is the home directory. This is the main directory in which your files are placed.

If you type `cd ~`, that will bring you back to your home directory. As a shortcut, `cd` by itself will do the same thing.

For the next section, go ahead and `cd` to the `linux` directory. One habit most programmers have: as soon as you change to a directory, get a listing to see what's in it.

```
1 username@dijkstra:~$ cd linux
2 username@dijkstra:~/linux$ ls
3 Hello.java  f1  f2  test1.txt  test2.txt
```

3.4   Customizing `ls`

You can customize the behavior of `ls` (and in fact, just about every command) by using *switches* or *command-line options*. A switch begins with one or two dashes. A one-dash switch usually has just one letter after it, while two-dash switches are more verbose. Here are some examples of one-dash switches for `ls`, and what they do.

- -F Puts a * after executable files and a / after directory names.

```
1 username@dijkstra:~/linux$ ls -F
2 Hello.java  f1/  f2/  test1.txt  test2.txt
```

  This makes it very easy to tell which files are directories, etc.

- -a "All". Filenames that begin with a . are usually hidden. This tells `ls` to show everything.

```
1 username@dijkstra:~/linux$ ls -a
2 .  ..  .hidden.txt  Hello.java  f1  f2  test1.txt  test2.txt
```

  Most files that begin with . are configuration files of some sort. Another common feature of configuration files is that they usually end with the characters `rc` (for "resource"). The `.pinerc` file is an example. In Unix, configuration information is scattered about multiple files. While this does mean you have to hunt for the proper file to configure a program (usually, for a program `foo`, the user configuration file will be called `.foorc`), you avoid the unpleasantness that occurs in Windows when something corrupts the registry.

  In this directory, there are three hidden files. The . and .. are the current and parent directories, respectively.

- -l "Long". Give more information about the file.

```
1 username@dijkstra:~/linux$ ls -l
2 total 12
3 -rw-r--r--  1 username users 117 Jan 22 20:08 Hello.java
4 drwxr-xr-x  2 username users 112 Jan 22 20:11 f1
5 drwxr-xr-x  2 username users 112 Jan 22 20:11 f2
6 -rw-r--r--  1 username users  68 Jan 22 20:09 test1.txt
7 -rw-r--r--  1 username users 123 Jan 22 20:10 test2.txt
```

  The first column contains the permission information. The `username` field is the user who owns it, and the `users` field is the group that owns the file. (In Unix, files belong to both a user and a group.) The very first letter sometimes has a `d`: this indicates a directory. The next column is the size of the file, followed by the date and the file name. For example, `test1.txt` is 68 bytes long.

  You can combine switches, too.

```
1 username@dijkstra:~/linux$ ls -laF
2 total 16
3 drwxr-xr-x  4 username users 224 Jan 22 20:11 ./
4 drwxr-xr-x  3 username users 128 Jan 22 20:28 ../
5 -rw-r--r--  1 username users  31 Jan 22 20:10 .hidden.txt
6 -rw-r--r--  1 username users 117 Jan 22 20:08 Hello.java
```

Mattox Beckman   **Page 7**   Illinois Institute of Technology

```
 7 drwxr-xr-x  2 username users 112 Jan 22 20:11 f1/
 8 drwxr-xr-x  2 username users 112 Jan 22 20:11 f2/
 9 -rw-r--r--  1 username users  68 Jan 22 20:09 test1.txt
10 -rw-r--r--  1 username users 123 Jan 22 20:10 test2.txt
```

- You can ask to list certain files directly:

```
1 username@dijkstra:~/linux$ ls test1.txt
2 test1.txt
```

- ...and directories...

```
1 username@dijkstra:~/linux$ ls f1
2 test1.txt  test2.txt
```

- You can also use *wildcards*. To get all the files ending in `.txt` you can use `*.txt`

```
1 username@dijkstra:~/linux$ ls *.txt
2 test1.txt  test2.txt
```

3.5 Directory Management

You've seen `cd` and `pwd`. There are commands to allow you to create and remove directories, also.

- `mkdir`, for *make directory*. You can create a new directory with this command. Actually, you can create as many as you feel like typing at the same time.

```
1 username@dijkstra:~/linux$ mkdir t1 t2 t3
2 username@dijkstra:~/linux$ ls -F
3 Hello.java  f1/  f2/  t1/  t2/  t3/  test1.txt  test2.txt
```

One important note about directory names: if the name begins with `/`, then the directory is assumed to start with the root directory. Otherwise, the name is assumed to be relative to the current directory. For example...

```
 1 username@dijkstra:~/linux$ cd t1
 2 username@dijkstra:~/linux/t1$ pwd
 3 /home/students/all/username/linux/t1
 4 username@dijkstra:~/linux/t1$ cd t2  # This should not work.
 5 -bash: cd: t2: No such file or directory
 6 username@dijkstra:~/linux/t1$ cd ../t2  # Now try this.
 7 username@dijkstra:~/linux/t2$ pwd
 8 /home/students/all/username/linux/t2
 9 username@dijkstra:~/linux/t2$ cd /home/students/all/username/linux/t3
10 username@dijkstra:~/linux/t3$ pwd
11 /home/students/all/username/linux/t3
```

Note that the # character, when the user types it, indicates the beginning of a comment in most shells.

- **rmdir**, for *remove directory*. This will erase a directory, if it is empty. If it's not empty, it will give an error message.

```
1  username@dijkstra:~/linux$ pwd
2  /home/students/all/username/linux
3  username@dijkstra:~/linux$ ls -F
4  Hello.java  f1/  f2/  t1/  t2/  t3/  test1.txt  test2.txt
5  username@dijkstra:~/linux$ rmdir t1
6  username@dijkstra:~/linux$ ls -F
7  Hello.java  f1/  f2/  t2/  t3/  test1.txt  test2.txt
8  username@dijkstra:~/linux$ rmdir f1
9  rmdir: 'f1': Directory not empty
10 username@dijkstra:~/linux$ ls -F
11 Hello.java  f1/  f2/  t2/  t3/  test1.txt  test2.txt
12 username@dijkstra:~/linux$ ls f1
13 test1.txt  test2.txt
```

3.6   Aliases

Tired of typing `ls -F` every time? The `alias` command can save you some typing.

```
1  username@dijkstra:~/linux$ ls
2  Hello.java  f1  f2  t2  t3  test1.txt  test2.txt
3  username@dijkstra:~/linux$ alias ls="ls -F"
4  username@dijkstra:~/linux$ ls
5  Hello.java  f1/  f2/  t2/  t3/  test1.txt  test2.txt
```

If you try to use `ls`, it will automatically expand it out to `ls -F`. It only happens for things at the beginning of the command line, though, so `mkdir ls` will *not* be expanded to `mkdir ls -F`.

Use the `alias` command by itself to list all your aliases. When I did that on my account, this is what I got:

```
1  username@dijkstra:~/linux$ alias
2  alias l.='ls -d .* --color=tty'
3  alias ll='ls -l --color=tty'
4  alias ls='ls -F'
5  alias vi='vim'
```

As you can see, sometimes there are other aliases that come already defined when you log in.

The `unalias` command will remove an alias.

```
1  username@dijkstra:~/linux$ unalias ls
2  username@dijkstra:~/linux$ alias
3  alias l.='ls -d .* --color=tty'
4  alias ll='ls -l --color=tty'
5  alias vi='vim'
```

3.7 File Management

There are similar commands to manage your files.

- `cat`, for *concatenate*. This command seems strange at first. It reads in whatever files you tell it on the command line, and then prints them to `stdout` (standard output), usually your screen. If you don't give it the name of any files, it takes the input from `stdin` (standard input), usually your keyboard. If you leave both out, then it will copy everything you type.

```
1  username@dijkstra:~/linux$ cat
2  hello
3  hello
4  world
5  world
6  ^C
```

The first `hello` and first `world` were typed by the user, the others were printed by the shell.

```
1  username@dijkstra:~/linux$ cat Hello.java
2  public class Hello {
3      public static void main(String[] args) {
4          System.out.println("Hello, world!\ n");
5      }
6  }
```

To create a file using this, you can *redirect* standard output using the `>` operator.

```
1   username@dijkstra:~/linux$ cat > foo
2   hello
3   world
4   ^D
5   username@dijkstra:~/linux$ cat foo
6   hello
7   world
8   username@dijkstra:~/linux$ cat foo foo
9   hello
10  world
11  hello
12  world
```

The `^D` (control-d) symbol is the end-of-file character in Unix. The last example shows why this command is called "concatenate".

You can also redirect standard input if you want, using the `<` operator, or the `|` operator. Here are some examples. Note that `-` is a special file name. It means either standard input *or* standard output, depending on the context.

```
 1 username@dijkstra:~/linux$ cat - > fruits
 2 durian
 3 pear
 4 apple
 5 cherry
 6 banana
 7 ^D
 8 username@dijkstra:~/linux$ cat foo fruits
 9 hello
10 world
11 durian
12 pear
13 apple
14 cherry
15 banana
16 username@dijkstra:~/linux$ cat < fruits > foo
17 username@dijkstra:~/linux$ cat foo
18 durian
19 pear
20 apple
21 cherry
22 banana
23 username@dijkstra:~/linux$ sort < fruits
24 apple
25 banana
26 cherry
27 durian
28 pear
29 username@dijkstra:~/linux$ cat fruits | sort
30 apple
31 banana
32 cherry
33 durian
34 pear
```

The `sort` command is a standard Unix command. It takes lines from standard input, sorts them, and puts them onto standard output. The last two examples show `sort` in action. Note well that `sort` does not change the original file: `fruits` still has its original unsorted data inside! Question, what do you think `sort < fruits > sfruits` will do?

- `cp`, for *copy* It takes two arguments, the source file and the destination file.

```
 1 username@dijkstra:~/linux$ cp fruits bats
 2 username@dijkstra:~/linux$ cat bats
 3 durian
 4 pear
```

```
5  apple
6  cherry
7  banana
```

- mv, for *move*. There are two versions of this command. One moves one or more files to a new directory; the other renames a file. If the last argument is a directory name, then all the rest of the arguments are moved into it. Note: directories can be moved this way, too! If the last argument is a file name, then the first argument is renamed as the second argument.

```
1  username@dijkstra:~/linux$ mkdir t1
2  username@dijkstra:~/linux$ mv fruits t1
3  username@dijkstra:~/linux$ ls t1
4  fruits
5  username@dijkstra:~/linux$ mkdir t2
6  username@dijkstra:~/linux$ mv t1 t2
7  username@dijkstra:~/linux$ ls t1
8  t1: No such file or directory
9  username@dijkstra:~/linux$ ls t2
10 t1
11 username@dijkstra:~/linux$ ls t2/t1
12 fruits
13 username@dijkstra:~/linux$ ls -R t2
14 .:
15 t1
16
17 ./t1:
18 fruits
19 username@dijkstra:~/linux$ mv bats cows
20 username@dijkstra:~/linux$ cat cows
21 durian
22 pear
23 apple
24 cherry
25 banana
```

- rm, for *remove*.

  This one is dangerous. It erases things. You will *not* be able to get them back afterwards.

```
1  username@dijkstra:~/linux$ cat - > foo
2  hi
3  ^D
4  username@dijkstra:~/linux$ cat foo
5  hi
6  username@dijkstra:~/linux$ rm foo
```

```
7  username@dijkstra:~/linux$ cat foo
8  cat: cannot open foo
```

If you are scared, the `-i` option will cause `rm` to ask for confirmation first.

```
1  username@dijkstra:~/linux$ rm -i cows
2  rm: remove cows (yes/no)? n
3  username@dijkstra:~/linux$ ls cows
4  cows
```

Use `alias` to redefine `rm` if you want this all the time.

If this is not dangerous enough for you, the `-r` option makes it recursive. It will destroy a file, or if it is a directory, it will destroy the directory and everything within it.

```
1  username@dijkstra:~/linux$ rm -r t2
2  username@dijkstra:~/linux$ ls t2
3  t2: No such file or directory
```

3.8   Man Pages

The commands we showed have a lot of other options. There is a command called `man` (for "manual") that will give you the instructions. So, `man ls` will give you all the other options for `ls`, and `man bash` will give you the manual for the shell, perhaps disappointing the radical feminists in the audience. There are manual pages for all kinds of things, not just commands. For example, `man ascii` will give you the ASCII table, and `man printf` will give you the programming instructions for the C function `printf`.

3.9   Shell Variables

The `bash` shell, like most shells, has variables; it is a real programming language. (Note: not all programming languages have variables! Ask if you're curious.)

A shell variable begins with `$`. There are a few predefined variables. Two of the most important are `$HOME` and `$PATH`. To examine them, you can use the `echo` command, which is the "print" statement of shell languages.

```
1  username@dijkstra:~/linux$ echo Testing
2  Testing
3  username@dijkstra:~/linux$ echo Home variable is $HOME
4  Home variable is /home/students/all/username
5  username@dijkstra:~/linux$ echo Path is $PATH
6  Path is /usr/kerberos/bin:/usr/local/bin:/bin:
7  /usr/bin:/usr/X11R6/bin:/home/students/all/username/bin
```

The `$HOME` variable contains the directory name for your home directory. The `$PATH` directory lists all the directories in which are commands that could be executed. The directories are separated by colons.

There are two ways to set a variable. You can use the syntax `X=y` to set variable `$X` to `y`. Note that we leave off the `$` for `X` here—what do you think would happen if we did not?

Mattox Beckman   **Page 13**   Illinois Institute of Technology

```
1  username@dijkstra:~/linux$ Answer=42
2  username@dijkstra:~/linux$ echo $Answer
3  42
```

These variables are kept in a structure called the *environment*. Other programs that you run have the ability to examine the environment, which makes using these environment variables a handy way to control the behavior of certain programs.

Variables that you define do not normally get exported to programs that you run, though. So a program would be able to use $HOME and $PATH, but not $Answer. To change that, use the export command.

If you've already defined the variable, (like $Answer), just use

```
1  username@dijkstra:~/linux$ export Answer
```

Again, leave off the $ this time or else weird things will happen. You can export and define a variable simultaneously:

```
1  username@dijkstra:~/linux$ export MetaQuestion="What is the answer to the ultimate question?"
2  username@dijkstra:~/linux$ echo $MetaQuestion
3  What is the answer to the ultimate question?
4  username@dijkstra:~/linux$ export foo="The answer is $Answer"
5  username@dijkstra:~/linux$ echo $foo
6  The answer is 42
7  username@dijkstra:~/linux$ export Answer=54
8  username@dijkstra:~/linux$ echo $foo
9  The answer is 42
```

You can do lots of things with variables. There are for-loops, while-loops, arrays, and mathematical functions. Typically, if you are using a lot of those, then you should start thinking of using a language like python or ruby instead.

3.10   Editors

Notepad is for making small lists and reading the README files that come with software. But even Notepad is better than using cat to write programs. Word processors are not at all designed to handle programs. There really is no substitute for having the right tool for the job. One of your most important tools, *even more important than which language you pick*, will be your editor. There are three editors available on dijkstra that are fairly popular. I'll list them in the order that I like them.

- vim, or "VI Improved", is my current favorite. Based on vi, it is programmable, and has an extensive library. It emphasizes being able to navigate files quickly, and is especially well suited for people who know how to touch-type. Its syntax highlighting abilities are unmatched.

  Use the vim command to get started. The first page has a tutorial, or just run the command vim-tutor.

  Advantages: special modes for certain kinds of programming; highly customizable and programmable, modal interface, a huge library of customizations written for it, tends to start (a lot) faster than emacs.

  Disadvantages: Steep learning curve, you have to use the escape key a lot to switch between command-mode and insert mode, and it will become a religion and you will be compelled to fight for its honor when compared with emacs.

- emacs, started in 1976 (!) and has continued to grow and evolve. There are two popular variants, FSF emacs (from the Free Software Foundation) and xemacs (developed in part at the University of Illinois at Urbana). The FSF version is called emacs and the other is called xemacs. Most users will not notice a difference between the two, other than that xemacs uses a bit more graphics.

  You can start it from your shell; pass it the -nw option to tell it not to open a graphic interface. MS-Windows cannot understand the graphic X protocol used by Unix.

  The first page will tell you how to start the tutorial.

  Advantages: special modes for certain kinds of programming; highly customizable and programmable, simple interface, a huge library of customizations written for it.

  Disadvantages: Steep learning curve, you have to use a lot of control key sequences to do things, and it will become a religion and you will be compelled to fight for its honor when it is compared with vim.

- joe, nano, etc.

  These are very simple editors, but they do the job. They are for the lazy programmer who is content with second-rate tools and practices, but still doesn't want to bear the shame of programming with notepad.

  Advantages: very small, short learning curve.

  Disadvantages: Not programmable, very little power, not as readily available as emacs and vim, and nobody cares enough about them to fight for their honor.

3.11   Startup Files

If you get tired of setting the PATH and redefining your aliases each time you log in, you can edit a file called .bash_profile and place the commands you want in it. This file is run every time you log in. Here is what's in mine. It sets the PATH so that our special version of Java will be found, and aliases ls.

```
username@dijkstra:~/linux$ cat .bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH
unset USERNAME
```

From reading the comments in this code, you can see that there is a another file called .bashrc, in which it is intended that aliases go.

```
username@dijkstra:~/linux$ cat .bashrc
# .bashrc
```

```
3
4  # User specific aliases and functions
5
6  alias ls="ls -F"
7
8  # Source global definitions
9  if [ -f /etc/bashrc ]; then
10          . /etc/bashrc
11  fi
```

These two files were created when your account was set up. I have added the "ls" alias to my .bashrc file.

4   Secure Shell Keys

Secure shell (SSH) is a means of communicating between systems. You use it to log on to dijkstra, but you can also use it to issue remote commands. If you want to be able to log into dijkstra without needing a password, or access your bitbucket account without a password, you will need this.

A full explanation of SSH and its key system is beyond the scope of this document. There are some good tutorials online that explain the whole system. Ask your favorite search engine, or check the resources page of the class web site.

To make things easy, your account has been set up with an initial password-less secure shell key. If you are familiar with SSH you can replace it with your own key. If not, you should be able to use the system as-is.

When you log in, the keychain program will show something like this:

```
1  Last login: Fri Sep  5 17:26:34 2008 from localhost
2
3  KeyChain 2.6.8; http://www.gentoo.org/proj/en/keychain/
4  Copyright 2002-2004 Gentoo Foundation; Distributed under the GPL
5
6   * Found existing ssh-agent (10886)
7   * Adding 1 ssh key(s)...
8  Identity added: /home/students/all/username/.ssh/id_rsa (/home/students/all/username/.ssh/id_rsa)
9
10  username@dijkstra:~$
```

If you see that, you are ready for the next section. If not, then something has gone wrong. Contact Professor Beckman or your TA to have it checked out.

5   The Git Version Control System

Some accounts will be given something called a *Git repository*. Git is a *version control system*; it's job is to track files in a project, recording the changes and allowing multiple users to collaborate without stepping on each other.

The files Git tracks live in a *repository*. A repository looks like a normal directory containing whatever files belong to your project. But at the top-level directory of the project, there is a hidden directory called .git. This hidden directory contains meta-data about your files, such as what changes have been made, and who made the changes. Be careful not to modify the files in the .git directory!

Git is a *distributed* version control system. Everyone working on the project will have a repository, each one identical to the others. When a change is made, the programmer making the change can send the changes to the other users. This is in contrast to systems such as Subversion, which have a master repository that serves as the "official" version of the project.

We will be using `bitbucket` to host your repositories. You should have received an email from them inviting you to use your new repository. The repository name is usually a combination of the course rubric, the year, the month, and your email id.

5.1 Creating Your Repository

To create your copy of the repository, you need to copy the server repository to your account. This is done via the `git clone` command. The command takes one or two arguments. The first is the location of the remote repository. Here's how to find your repository's URL.

If you log onto your `bitbucket` account, you will enter the *dashboard*. On the right you will see a list of your repositories. The one for the class will be named something like `cs331-2012-01-username`. Click it. (We will pretend that this is the name of your repository from here on out. If you are reading this for a different course in a future semester (hello future people!) then yours will be different.)

This will bring you to the *overview* tab. There is an option called *clone this repository* near the top. You have two choices. One is to use SSH. We won't explain that here. The other choice is to use HTTPS. Select that option, and below you will see a line like

```
git clone https://mattoxbeckman@bitbucket.org/mattoxbeckman/cs331-2011-08-username.git
```

In your `dijkstra` account, `cd` to your home directory, and type that command. It will ask you for a password. *This is your `bitbucket` password, not your `dijkstra` password!*

If all goes well, it will look something like this:

```
username@dijkstra:~$ git clone https://mattoxbeckman@bitbucket.org/mattoxbeckman/cs331-2011-08-username.git
Initialized empty Git repository in /home/students/all/username/cs331-2011-08-username.git/
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
username@dijkstra:~$ ls
cs331-2012-08-username
```

Once this is done, you will see a directory called `cs331-2012-08-username`. This is your repository.

5.2 The Workflow

In a typical work cycle, you will first *pull* any changes from the server repository into your repository, perform any edits, *commit* the changes to your repository, and when you are done for the day, *push* the changes back to the server repository. There are other commands you may need, and we will introduce some of them here.

5.2.1 Begin with `git pull`

The first thing to do is `cd` to the directory and run `git pull`. This fetches any changes from the server and copies them into your repository. You never know when the professor or a classmate (if you are working in teams) will update something; this makes sure you are using the most recent version of the files when you do your own editing.

```
username@dijkstra:~/cs331-2012-08-username$ git pull
Already up-to-date.
```

In this case, nothing has been changed on the server, so there's nothing to see.

5.2.2 Adding a File: `git add`

Let's add a file. Call it `Foo.java`.

```
username@dijkstra:~/cs331-2012-08-username$ cat - > Foo.java
public static class Foo {
    public void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
^D
```

The final `^D` is Control-D, the end-of-file character in unix.
Now let's ask Git what our status is.

```
username@dijkstra:~/cs331-2012-08-username$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       Foo.java
nothing added to commit but untracked files present (use "git add" to track)
```

This says that Git noticed a new file, but that Git has not been told to track changes to it. We want to track changes, so we do that with the `git add` command:

```
username@dijkstra:~/cs331-2012-08-username$ git add Foo.java
```

The output is usually minimal. Now let's ask about the status again:

```
username@dijkstra:~/cs331-2012-08-username$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   Foo.java
#
```

Now it says that `Foo.java` is a new file.

5.2.3 Committing Changes with `git commit`

Git knows that the file has been changed because the hidden `.git` directory has a reference version of all your files. When you typed the `git add` command above, you told Git that you intended to add `Foo.java` to it. The act of copying changes to the `.git` directory is called *committing*.

To perform a commit, you will use the `git commit` command.

There are many options you can give it, the most common is `-m`. This option allows you to specify a log message explaining what you changed. Git will not allow you to perform a commit without a log message. Here is what a commit looks like.

```
username@dijkstra:~/cs331-2012-08-username$ git commit -m "Added Foo.java."
Created commit 2efdf73: Added Foo.java.
 1 files changed, 5 insertions(+), 0 deletions(-)
 create mode 100644 Foo.java
```

Every commit gets a unique identifier. In this example, the commit ID was `2efdf73`. Note, the server repository does not yet see any of these changes! They have only been made in your account.

5.2.4 Using `git log` and `git diff` to Review History

If you want to see the list of recent changes, you can use `git log`.

```
username@dijkstra:~/cs331-2012-08-username$ git log
commit 2efdf73822025af197e94a9e4144fc4721b11720
Author: User Name <username@dijkstra.cs.iit.edu>
Date:   Mon Sep 22 16:02:16 2008 -0500

    Added Foo.java.

commit 452393d9efa243c8e74e10e98df7c1e04a68e0f5
Author: Mattox Beckman <mattox@dijkstra.cs.iit.edu>
Date:   Mon Sep 22 14:19:02 2008 -0500

    Added initial file.
```

In this example, there were two commits. The first one was done by the professor, the second one done by `username`.

Another command that is useful is `git diff`, where "diff" is short for *difference*.

Suppose you want to know the difference between the two commits. You can use the commit IDs as arguments to `git diff` to find out. Most of the time, you only need the first six characters of the ID.

```
username@dijkstra:~/cs331-2012-08-username$ git diff -r 452393 2efdf7
diff --git a/Foo.java b/Foo.java
new file mode 100644
index 0000000..90147d2
```

```
 5 │ --- /dev/null
 6 │ +++ b/Foo.java
 7 │ @@ -0,0 +1,5 @@
 8 │ +public static class Foo {
 9 │ +    public void main(String[] args) {
10 │ +        System.out.println("Hello, world!");
11 │ +    }
12 │ +}
```

The + indicates that a line was added, and the - indicates that a line was deleted. If a line was changed, `git diff` will indicate that by showing the old version as deleted and the new version as added.

If you type `git diff` by itself, it compares the last commit to the files in the directory. Currently they are the same, so there will be no output.

5.2.5   Making Changes

Let's modify our `Foo.java` to see what Git does. Let's change the greeting to "Hi, world!" using your favorite editor.

When you are done, running `git status` will tell you that Git noticed the change.

```
1 │ username@dijkstra:~/cs331-2012-08-username$ git status
2 │ # On branch master
3 │ # Changed but not updated:
4 │ #   (use "git add <file>..." to update what will be committed)
5 │ #
6 │ #       modified:   Foo.java
7 │ #
8 │ no changes added to commit (use "git add" and/or "git commit -a")
```

Git knows that `Foo.java` has been changed, but you have not told Git that you intend to keep the changes. If you run `git commit` right now, nothing will happen. So you can recognize it, here is what it looks like:

```
1 │ username@dijkstra:~/cs331-2012-08-username$ git commit
2 │ # On branch master
3 │ # Changed but not updated:
4 │ #   (use "git add <file>..." to update what will be committed)
5 │ #
6 │ #       modified:   Foo.java
7 │ #
8 │ no changes added to commit (use "git add" and/or "git commit -a")
```

As you can see, it simply re-ran `git status` for you.

At this point, `git diff` will return something interesting.

```
1 │ username@dijkstra:~/cs331-2012-08-username$ git diff
2 │ diff --git a/Foo.java b/Foo.java
3 │ index 90147d2..ba3d9b3 100644
```

```
4   --- a/Foo.java
5   +++ b/Foo.java
6   @@ -1,5 +1,5 @@
7    public static class Foo {
8        public void main(String[] args) {
9   -        System.out.println("Hello, world!");
10  +        System.out.println("Hi, world!");
11       }
12   }
```

If you type `git diff -r 452393`, leaving off the second commit ID, it will tell you the difference between that commit and the current version of the files. Notice the differences in the output.

```
1   username@dijkstra:~/cs331-2012-08-username$ git diff -r 452393
2   diff --git a/Foo.java b/Foo.java
3   new file mode 100644
4   index 0000000..ba3d9b3
5   --- /dev/null
6   +++ b/Foo.java
7   @@ -0,0 +1,5 @@
8   +public static class Foo {
9   +    public void main(String[] args) {
10  +        System.out.println("Hi, world!");
11  +    }
12  +}
```

To tell Git that you want these changes committed, you have to run `git add` again. This might seem like a pain, but it is useful because sometimes many files will change at once, and you might want to commit the changes in smaller groups. Re-adding the files gives you a mechanism to do that.

```
1   username@dijkstra:~/cs331-2012-08-username$ git add Foo.java
2   username@dijkstra:~/cs331-2012-08-username$ git commit -m "Changed hello to hi."
3   Created commit 5fccd8d: Changed hello to hi.
4    1 files changed, 1 insertions(+), 1 deletions(-)
```

Now you have three commits in your repository.

5.2.6   Updating the Server with `git push`

At this point, your three commits are in your repository, but not in the server's repository. To fix this, you will want to run `git push`.

```
1   username@dijkstra:~/cs331-2012-08-username$ git push
2   Counting objects: 7, done.
3   Compressing objects: 100% (6/6), done.
4   Writing objects: 100% (6/6), 694 bytes, done.
```

```
5  Total 6 (delta 1), reused 0 (delta 0)
6  To csXXX-git@dijkstra.cs.iit.edu:cs331-2012-08-username.git
7     452393d..5fccd8d  master -> master
```

The details will be different, and it will again ask for your `bitbucket` password. But once that is done you can click the *source* tab on `bitbucket` and see your files!

5.2.7   Retrieving Updates

Suppose now the professor tells you that the staff have changed some of your files. To get the changes, run `git pull` again.

```
1  username@dijkstra:~/cs331-2012-08-username$ git pull
2  Unpacking objects: 100% (3/3), done.
3  remote: Counting objects: 5, done.
4  remote: Compressing objects: 100% (3/3), done.
5  remote: Total 3 (delta 1), reused 0 (delta 0)
6  From csXXX-git@dijkstra.cs.iit.edu:username
7     5fccd8d..4ad72dd  master      -> origin/master
8  Updating 5fccd8d..4ad72dd
9  Fast forward
10  Foo.java |    1 +
11  1 files changed, 1 insertions(+), 0 deletions(-)
```

You will see from the screen that `Foo.java` has been updated. If you return later and can't remember, the command `git whatchanged` will also let you know which files were modified.

```
1  username@dijkstra:~/cs331-2012-08-username$ git whatchanged
2  commit 4ad72dd964b8796cd67338ea07d1610f1a08b1b9
3  Author: Mattox Beckman <mattox@dijkstra.cs.iit.edu>
4  Date:   Mon Sep 22 16:57:51 2008 -0500
5
6      Added a local variable.
7
8  :100644 100644 ba3d9b3... 8cb297c... M  Foo.java
```

The rest of the output has been truncated.

5.2.8   Getting More Details about What Changed with `git blame`

You can use `git diff` to see the differences, but if you want to see the changes in the context of the whole file, `git blame` is the command you want.

```
1  username@dijkstra:~/cs331-2012-08-username$ git blame Foo.java
2  2efdf738 (User Name      2008-09-22 16:02:16 -0500 1) public static class Foo {
3  4ad72dd9 (Mattox Beckman 2008-09-22 16:57:51 -0500 2)    int i;
```

```
4 2efdf738 (User Name        2008-09-22 16:02:16 -0500 3)      public void main(String[] args) {
5 5fccd8d8 (User Name        2008-09-22 16:21:45 -0500 4)          System.out.println("Hi, world!
6 2efdf738 (User Name        2008-09-22 16:02:16 -0500 5)      }
7 2efdf738 (User Name        2008-09-22 16:02:16 -0500 6) }
```

Each line of the code is shown. The first column is the commit ID of the last change affecting that line. The name of the user name, the date, and the line number precedes the code itself. We can see the professor changed line 2.

5.2.9   Rejection when Pushing

Sometimes you commit some changes and try to push the commits to the server only to find out you've been rejected. Suppose, for instance, you added an extra local variable.

```
1 username@dijkstra:~/cs331-2012-08-username$ git diff
2 diff --git a/Foo.java b/Foo.java
3 index 8cb297c..564731f 100644
4 --- a/Foo.java
5 +++ b/Foo.java
6 @@ -1,5 +1,5 @@
7  public static class Foo {
8 -    int i;
9 +    int i,j;
10     public void main(String[] args) {
11         System.out.println("Hi, world!");
12     }
13 username@dijkstra:~/cs331-2012-08-username$ git commit -am "Added j."
14 Created commit 6e3264e: Added j.
15  1 files changed, 1 insertions(+), 1 deletions(-)
```

(Note the `-am` flag. The `m` part is just the log message, but the `-a` part tells Git to automatically add any changed files to the commit. This saves you from having to run `git add` so many times.)

But now when you go to push, you get this output.

```
1 username@dijkstra:~/cs331-2012-08-username$ git push
2 To username@dijkstra.cs.iit.edu:username.git
3  ! [rejected]        master -> master (non-fast forward)
4 error: failed to push some refs to 'csXXX-git@dijkstra.cs.iit.edu:username.git'
```

This means that the server repository has newer information in it than yours. In order to maintain integrity, you need to do a pull first.

```
1 username@dijkstra:~/cs331-2012-08-username$ git pull
2 remote: Counting objects: 4, done.
3 remote: Compressing objects: 100% (3/3), done.
4 remote: Total 3 (delta 0), reused 0 (delta 0)
5 Unpacking objects: 100% (3/3), done.
```

```
6  From csXXX-git@dijkstra.cs.iit.edu:username
7     4ad72dd..17bc225  master      -> origin/master
8  Merge made by recursive.
9   Bar.java |    5 +++++
10  1 files changed, 5 insertions(+), 0 deletions(-)
11   create mode 100644 Bar.java
12 username@dijkstra:~/cs331-2012-08-username$ git push
13 Counting objects: 8, done.
14 Compressing objects: 100% (5/5), done.
15 Writing objects: 100% (5/5), 634 bytes, done.
16 Total 5 (delta 1), reused 0 (delta 0)
17 To csXXX-git@dijkstra.cs.iit.edu:username.git
18     17bc225..73ce837  master -> master
```

5.2.10   Conflicts

What if two people change the same file? As long as the changes are on different lines, Git will figure it out automatically. But suppose you and the professor modify the same line.

So, for `Bar.java` (which `git pull` just added) you change to

```
1 public static class Bar {
2    public void main(String[] args) {
3       System.out.println("Bye there, world!");
4    }
5 }
```

But while you were still editing, the professor changed it to:

```
1 public static class Bar {
2    public void main(String[] args) {
3       System.out.println("Good bye, world!");
4    }
5 }
```

You do a commit, and all looks fine,

```
1 username@dijkstra:~/cs331-2012-08-username$ git commit -am 'Changed string.'
2 Created commit 44dfa7a: Changed string.
3  1 files changed, 1 insertions(+), 1 deletions(-)
```

Now the push...

```
1 username@dijkstra:~/cs331-2012-08-username$ git push
2 To csXXX-git@dijkstra.cs.iit.edu:cs331-2012-08-username.git
3  ! [rejected]       master -> master (non-fast forward)
4 error: failed to push some refs to 'csXXX-git@dijkstra.cs.iit.edu:cs331-2012-08-username.git'
```

It was rejected, but you already know how to fix that, just run `git pull` first.

```
username@dijkstra:~/cs331-2012-08-username$ git pull
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 5 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From csXXX-git@dijkstra.cs.iit.edu:username
   73ce837..aeefe22  master      -> origin/master
Auto-merged Bar.java
CONFLICT (content): Merge conflict in Bar.java
Automatic merge failed; fix conflicts and then commit the result.
```

This does not look good. If you look at the message, it will tell you that `Bar.java` is the problem file. If you open it up, you will see this:

```
public static class Bar {
    public void main(String[] args) {
<<<<<<< HEAD:Bar.java
        System.out.println("Bye there, world!");
=======
        System.out.println("Good bye, world!");
>>>>>>> aeefe22b46ab455dab8ee37f0e711b6f737f8755:Bar.java
    }
}
```

The ID `HEAD` means the current files. So you see both versions of the file, yours and the professor's. All you have to do is edit the file so it looks the way you want it. Let's keep your version. Delete all the extraneous lines to get your original version back.

When you run `git status`, it will look like this.

```
username@dijkstra:~/cs331-2012-08-username$ git status
Bar.java: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       unmerged:   Bar.java
#       modified:   Bar.java
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Now you can commit and push.

```
username@dijkstra:~/cs331-2012-08-username$ git commit -am "Fixed conflict."
Created commit fe74447: Fixed conflict.
```

```
 3  username@dijkstra:~/cs331-2012-08-username$ git status
 4  # On branch master
 5  nothing to commit (working directory clean)
 6  username@dijkstra:~/cs331-2012-08-username$ git push
 7  Counting objects: 8, done.
 8  Compressing objects: 100% (4/4), done.
 9  Writing objects: 100% (4/4), 495 bytes, done.
10  Total 4 (delta 2), reused 0 (delta 0)
11  To csXXX-git@dijkstra.cs.iit.edu:cs331-2012-08-username.git
12     aeefe22..fe74447  master -> master
```

5.3   What next?

More information and tips will be added as time allows, but for now this will get you started. Please send me questions if something is confusing, badly worded, or missing, and it will show up in the next revision.

Until then, please browse the git documentation at `http://git.or.cz/index.html#documentation`.

6   Frequently Asked Questions

6.1   Secure Shell

6.1.1   I already have my own public key. Can I use that?

Yes, go ahead. Just place it into the `.ssh` directory on `dijkstra`. The key watcher will notice and copy it to the correct location.

6.1.2   I have a DSA key. Can I use that?

The script is not set up for DSA keys, but can be quickly if you want that. Let me know if that's what you want.

6.2   Connecting from Windows

6.2.1   How can I connect using Cygwin?

The command you need is

```
 1  $ ssh username@dijkstra.cs.iit.edu
```

If you will need to use windowed programs (such as Eclipse, X-Term, etc.) then you need to start an X session first. You can do that with the command

```
 1  $ startx
```

6.2.2   I got a message: `cannot open display:`. What does that mean?

Unix uses a window protocol called X. To use windowed programs remotely, you have to activate X on your machine. Under Cygwin, you can do this by typing `startx` at the command prompt.

6.3   Git

6.3.1   I developed my files on my own machine. Where do I put them on `dijkstra` in order to turn them in?

Assuming you have cloned the repository as described in section 5, you should put them in the directory `/cs331-2012-08-username/`

Then you need to run `git add` for each file you've added or changed, then `git commit` to store it in your repository, and then `git push` to turn it in.

For example, suppose you wanted to turn in the files `Tree.java` and `Readme.txt`, and the instructions for the lab say to put them in the `lab-7` directory.

First, you would create the `lab-7` directory in your repository.

```
username@dijkstra:~$ mkdir cs331-2012-08-username/lab-7
```

Then you would transfer the files over.
Next, you would type something like this.

```
username@dijkstra:~$ cd cs331-2012-08-username/lab-7
username@dijkstra:~/cs331-2012-08-username/lab-7$ git add Tree.java Readme.txt
username@dijkstra:~/cs331-2012-08-username/lab-7$ git commit -m "Turned in lab 7."
Created commit 4659414: Turned in lab 7
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 Readme.txt
 create mode 100644 Tree.java
username@dijkstra:~/cs331-2012-08-username/lab-7$ git push
Counting objects: 5, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 335 bytes, done.
Total 4 (delta 0), reused 0 (delta 0)
To csXXX-git@localhost:cs331-2012-08-username.git
   851cb5d..4659414  master -> master
```

Your output will be different than this, but it should look similar.

6.3.2   How can I tell if I pushed my files properly?

The fastest way is to re-clone your repository. First, make sure there's no directory called `test`. Then run the following command from your home directory.

```
username@dijkstra:~$ git clone csXXX-git@dijkstra.cs.iit.edu:cs331-2012-08-username.git test
```

Of course, replace `username` and `CSXXX` with the proper values. And whatever you do, *don't run this command from within another repository!* It won't break anything, but you will get very confused later, and then have to come to office hours for multiple help sessions to get this fixed. (Yes, I've seen this happen.) If you did it by mistake, don't panic, just `rm -r` the directory you just made and clone again in a different directory.

This will create a directory called `test`. In it will be the exact files that the instructional staff will see.

6.3.3   I run `git push` or `git pull` and it ask me for a password.

This means your secure shell key has not been activated. Try running the following commands:

```
1  username@dijkstra:~$ keychain
2  username@dijkstra:~$ source ~/.keychain/dijkstra.cs.iit.edu-sh
3  username@dijkstra:~$ ssh-add
```

If that doesn't work, or you get an error, you need to email me.

6.3.4   Isn't there a faster way to commit changed files?

Yes, if the file has been previously added to the repository, you can use `git commit -am "LOG MESSAGE"` to commit. The `-a` flag tells git to include all changed files. If the file has been added for the first time, you still need to run `git add` on it first.

6.4   Git Errors

6.4.1   I tried to `git push`, and it said that it was rejected.

See section 5.2.9 for a discussion of why this happens, but if you're in a hurry, try running `git pull` first.

6.4.2   I tried to `git pull` and it gave me the message `fatal: unable to create '.git/index': File exists`.

This usually means that a previous Git command was aborted somehow. Just to a `rm .git/index` and try again.

6.4.3   I tried to run `git pull`, and it told me that a file was unmerged.

This is usually caused by a conflict. See section 5.2.10.

6.4.4   I tried to run `git pull`, and it gave me the message `fatal: Entry 'a' would be overwritten by merge. Cannot merge`.

This means that you have a file called `a` that is not being tracked, but that the server repository is tracking it. Try removing `a` from your repository, and then do the pull again.