Computer Science 381
Programming Unix in C
The College of Saint Rose
Winter Immersion 2016

Lab 1: C and Unix Introduction

Due: Monday, December 21, 2015

In this first lab, you will be introduced to the basics of C and Unix.

You may ask your instructor and classmates for help as you complete this lab, but the work you submit must ultimately be your own. If you are completely unfamiliar with Unix, don't hesitate to ask questions! On the other hand, if you have some experience, don't hesitate to help a classmate!

If you choose not to work on the Macintosh systems in Science Center 469A, you can safely ignore the references to and tasks related to the Macintosh-specific items.

Preliminaries

Before you begin work on this lab, you should make sure you can log into the Macs in Science Center 469A (these should accept your regular username and password) and that you have an account on and remember your password to our remote-access Linux system `mogul.strose.edu` (a separate account that needs to be set up if you have not used this system for a previous course).

Also, read over the description of the types of items you will encounter in our labs on the course home page.

Motivation and Background

Most computer users interact with modern operating systems using *graphical user interfaces (GUIs)*. Icons represent applications, files, and folders. Running programs present one or more graphical windows though which the user interacts with the keyboard, mouse, touch screen, or other devides. GUIs are nice, and have made computing accessible to a wide audience of users, but they are not always the most efficient way to interact with the computer. Many tasks can be accomplished much more efficiently by interacting with the system using *shell*, or *command line*, where commands are issued directly to the system by typing them at the keyboard.

When working at the command line, you will be presented with a prompt. This is your direct interface to issue commands to the operating system. When you type a command here, the shell will execute the command on your behalf, print out any results, then reissue the prompt.

We will focus here on the command line in Unix-like systems. Linux is the most popular current traditional Unix variant in use today, but all Macs running MacOS X are also Unix systems that "hide" their Unix from users, unless they know where to look. There are ways to interact with Microsoft Windows systems with a Unix command-line as well, using tools such as Cygwin. Other current and historical Unix systems include FreeBSD, OpenBSD, SunOS/Solaris, and IBM's AIX. Wikipedia has an article about the History of Unix. In the mobile world, iOS tries hard to make shell access impossible, but tools like Terminal Emulator for Android can provide at least a limited Unix command line on Android devices (which are based on Linux).

The command line is itself a program called a *shell*. There are many shell programs, but most Unix users now use either `bash` or `tcsh`. `bash` is an enhancement of the earlier `sh` shell, known as the "Bourne Shell", and its name is a play on that name: the **B**ourne **A**gain **SH**ell. Your default shell on most modern systems is likely to be `bash`. `tcsh` is an enhanced version of the `csh`, or "C Shell". You can run the same commands from any shell, but the control structures (loops, conditionals, etc.) have shell-specific syntax. These differences will not be important to us for now, and where relevant, our course materials will assume you are at a `bash` command line.

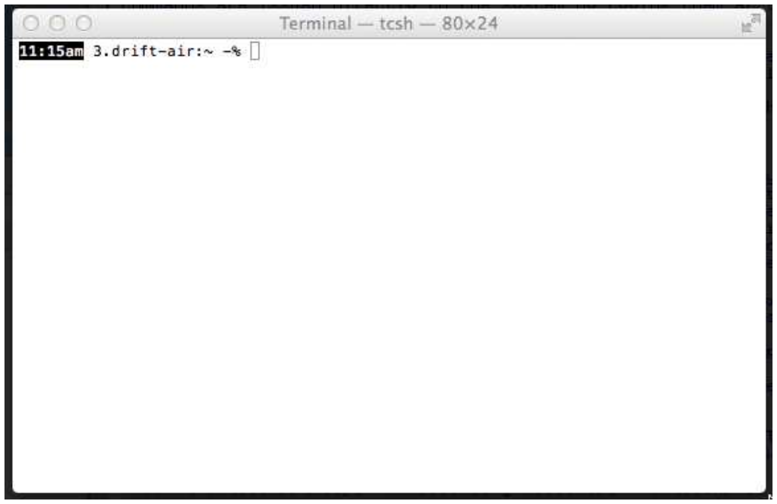
Accessing The Command Line

For many of our tasks this semester, it will not matter which Unix variant you use. For this lab, we need to make sure everyone can get to the Unix command line on at least the two systems "officially supported" in this course: Linux on the virtual server `mogul.strose.edu`, and the Mac OS `Terminal` on the Science Center 469A Mac systems.

MacOS X `Terminal`

Once logged into a Science Center 469A Mac system, you can access the command line in a few ways – we'll start with the `Terminal` application.

You can launch the `Terminal` application from the Finder under "`Applications`", then "`Utilities`", where you will find "`Terminal`". Or, just use the Spotlight (the magnifying glass in the upper right corner of the desktop) and type "`Terminal`" and click its icon to launch. You should see a new window pop up that looks something like this:



```
11:15am 3.drift-air:~ -%
```

Your prompt might look different (mine's `tcsh` and I've customized it with some extra information) but there should be some sort of prompt.

Give your shell a command to make sure it work. Try typing "`who am i`" and make sure it produces some output.

Remote Login to Linux

We can only access our virtual Linux server, `mogul.strose.edu`, by connecting to it over the Internet. This is done using a remote login protocol called *secure shell (ssh).* mogul runs a secure shell *server* which can accept login requests from secure shell *clients*.

One such client is the `ssh` program available from the command line on our Macs. If your username on `mogul.strose.edu` is `jcool`, issue the command

```
ssh mogul.strose.edu -l jcool
```

at the terminal prompt. Log in with your `mogul.strose.edu` password. You should be presented with a prompt that looks something like:

```
[jcool@mogul ~]$
```

and mogul is now ready to accept your commands. Try the same "`who am i`" command there to make sure it will respond to your commands.

You can also connect from Windows systems with ssh clients such as PuTTY.

Passing Command Line Parameters

We have only used a couple of Unix commands so far, but some of them have consisted of multiple "words" typed at the command prompt. In a Unix system, the first is always the name of the program to run (more on this later), and the others are the *command-line parameters* – values that are given to the program when it starts up that can help control exactly what it does.

In the case of `ssh`, it requires at least that one command-line parameter to specify the name of the server on the network to which you wish to connect. The other parameters tell `ssh` what username to use to establish that connection. Note that the `-l` and the username that follows are optional: if omitted, `ssh` will use the username associated with the shell from which it is issued. So if your usernames match between the Mac and mogul, you can omit those parameters.

The X Window System

Most of our Unix discussion will focus on commands that are completely text-based: you type commands at a shell, the program interacts with you by you typing into the shell and reading its output in the terminal. That's not to say Unix systems do not have the ability to use GUIs, however. Some of our tasks, most importantly editing of files, will be made easier by using programs that have a GUI.

The *de facto* standard for Unix GUIs is the *X Window System*, often also called *X11*. (Side note 1: a very early version of this course was based on a similar one that was once taught at Williams College, entitled *C, Unix, and the Joy of X*.) (Side note 2: the "11" here is the major version number of the system, though I can't imagine there will ever be an "X12", since it's been "X11" for the 25 or so years I've been using it.) X has a great advantage over many windowing systems in that the program whose windows are displayed on a screen do not necessarily need to be running on the same computer to which the screen is attached. We will take advantage of this shortly to run programs on mogul, which has no graphical screen attached, and display the user interface for those programs on the Macs in Science Center 469A.

First, we need to launch the X server on our Mac, which will translate graphics events from programs that use X11 graphics to Mac native graphics calls. In the Spotlight, type "`XQuartz`" and run that program. Once `XQuartz` has launched, you should be able to run X-based command-line programs from your Mac command shell. A good, simple test is to try the `xeyes` command. If a pair of eyeballs that follow your mouse pointer come up, X is running properly. If you get an error message or simply no window, something went wrong. Assuming you don't want to keep those eyeballs there, type `Ctrl-c` in your terminal to kill the program. `Ctrl-c` generates an interrupt signal that your shell sends to the program running within it, which normally results in that program terminating. If this works, the eyeballs should go away.

X Window System servers also exist for Microsoft Windows, and if you plan to work remotely on mogul from a Windows system, you will probably want to install one of them. Here is a link to one such tool and how to use it with PuTTY. Your mileage may vary. In our course materials, the assumption is that you are sitting in front of a Mac in Science Center 469A.

The Emacs Editor

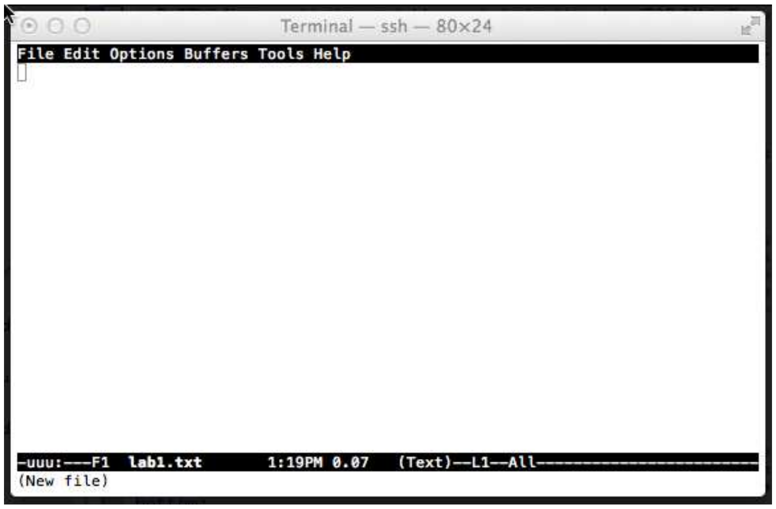
Emacs (`emacs` from the Unix command line) is a powerful text editor, which is very good for programming in a language like C and for general plain-text editing. You will need to become familiar with it.

Locate your existing command shell or create a new command shell logged into mogul. We will will use the `emacs` command in that shell to run Emacs to create your `lab1.txt` file that will contain your answers to this week's lab questions. For at least this lab, you are to create this file in your home directory on mogul.

In a shell on mogul, start Emacs with

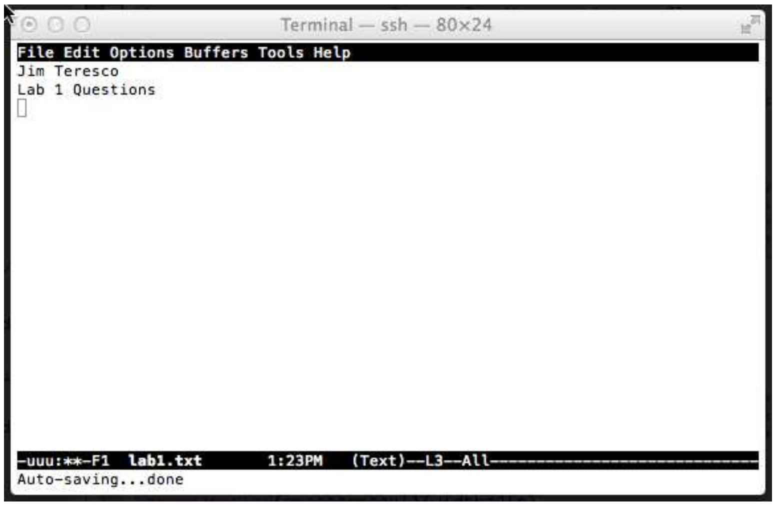
```
emacs lab1.txt
```

Emacs should start up, and present you with a text-based menu across the top (which we will purposely ignore), a large area where you can edit the file, and two lines of status information across the bottom:



```
-uuu:---F1  lab1.txt       1:19PM 0.07    (Text)--L1--All----------------------
(New file)
```

Type your name and "Lab 1 Questions" in the Emacs window that is editing the file `lab1.txt`:



```
Jim Teresco
Lab 1 Questions

-uuu:**-F1  lab1.txt       1:23PM   (Text)--L3--All---------------------------
Auto-saving...done
```

Some things to notice here:

- Two of the dashes in the status bar at the bottom have become $\star$'s. This tells you that the file you are currently editing has been modified since it was last saved. To remedy this, type `Ctrl-x` then `Ctrl-s`, the `emacs` "save buffer" command. Hopefully those $\star$'s will become $-$'s again, and you'll get a message at the bottom of the window that the file has been written.
- You can click in the window, but the `emacs` cursor does not move. Mouse events are not being delivered to `emacs`, so it cannot respond to them. To move the cursor, we can use the arrow keys (and other keystrokes we'll learn soon). Similarly, we cannot click on the text-based menu across the top of the window. While it is possible to navigate those menus with the keyboard, you don't want to do this. Much of the real power of Emacs (and of Unix in general) comes from the ability to issue commands from the keyboard without navigating menus or even using a pointing device at all!

- There are other items in the status bar, some of which we'll discuss later.

Unless making just a few quick edits, we do want to run Emacs in its GUI mode. So let's first exit the Emacs you were working in by typing `Ctrl-x` then `Ctrl-c`. You should be back to your shell prompt.

We will now connect to mogul using an extra option that will allow it to display our Emacs session back to the X server running on the Mac. As long as `X`Quartz is still running, the command:

```
ssh -Y mogul.strose.edu -l jcool
```

will connect to mogul as before, but will allow any command that tries to create graphical windows to display them back on your Mac. The `-Y` flag instructs ssh to set up an X11 tunnel to display programs running on the remote computer on your local screen.

Once you have your prompt, start emacs again:

```
emacs lab1.txt
```

This time, Emacs should launch in a separate window, and you'll be able to use the mouse to select from menus at the top of the window and to move the cursor around.

One problem, though, is that the `emacs` program is still occupying the command line – we don't have our prompt back to be able to do anything else. To do so, we will type `Ctrl-z` in the terminal. That will issue a stop interrupt to the program and give us the prompt back. Unfortunately, it also stops our instance of `emacs` from running, so it will not respond to the keyboard or mouse. To get it to respond again without reclaiming its control of the command prompt, we can then issue the command `bg`. This will tell the most-recently stopped program (in this case, `emacs`) to start executing in the background. Now, you should be able to use the Emacs window as well as continue to type more commands in the terminal.

That's a very common thing to do with X-based Unix programs, and it's normally combined into a single step. If we start Emacs instead with

```
emacs lab1.txt &
```

The `&` at the end will tell our shell to start the program but put it right into the background. This will create an Emacs window that is responsive to keyboard and mouse events, and will immediately reissue a prompt to allow the shell to continue to be used. You can use the `&` to start any Unix command in the background, but it is most commonly used to start X-based programs that will be running for a long time. This is our first example of process control that is at the heart of what makes Unix systems so powerful.

Now it's time to learn some Emacs commands. Launch another Emacs session, either on the Mac or on mogul, where you can type some text and then identify the function of and experiment with these Emacs commands:

```
C-x C-s    C-x C-c    C-x C-f    C-x C-w    C-g    C-a    C-e
C-d        C-_        C-v        M-v        C-s    C-r    M-%
C-k        C-y        M-gg       C-x u
```

`C-` before a key means hold down `Ctrl` and hit that key. `M-` indicates the "Meta" key, which on most systems is `Esc`. To issue a Meta command, hit the `Esc` key, release it, then hit the key(s) for the command you wish to issue. Use the keystrokes rather than the menus. It will save you time later if you spend some extra time getting used to the keyboard commands now! Note: for some of these commands, a very small buffer (that is, the contents of the file you are editing) will not allow you to see what they do. So create a file with several screens full of text before you go too far.

? Question 1:
Give a brief description of each of the Emacs commands above in `lab1.txt` (3 points).

Directory Structure

It is always important, but especially so when working with the Unix command line, to know where the files in various directories (often called "folders" on Macintosh and Windows systems because of how they are visually represented in GUIs) you might be using are actually stored, and where and how those are accessible.

On the Macs in our labs, your home directories are (unfortunately) local to each station. Any files you save there are only on that computer and are not guaranteed to remain for a later session. If you want to save files on your college network space, you will need to "mount it". To do so, from any Finder window, choose the "SAN2" or similar entry on the left, and find your name in the big list of folders that comes up. Double click to connect and then make sure you save your files to the volume that you mount.

On `mogul.strose.edu`, we find a more standard Unix style environment. Each user has a *home directory* where only that user has permission to read and write files. Your home directory is the initial *current directory* or *working directory* when you first log in.

The working directory is where the program will look for files unless instructed to do otherwise. You'll hear Unix users asking a question like "What directory are you in?" and the answer to this is your working directory.

In Unix environments, *paths* to files and directories are the sequence of directories that should be traversed to locate a given file or directory. Paths that start with a "/" are *absolute paths*, where the search starts at the system-wide *root directory* (represented as "/"). Other paths are *relative paths*, and begin their search at the working directory.

The command `pwd` will instruct the shell to print your working directory.

? Question 2:
What is your home directory on `mogul.strose.edu`? (use `pwd`)

Note: lab questions are worth 1 point each unless otherwise specified.

? Question 3:
What is your home directory when you open a fresh Macintosh Terminal window?

Directory paths are not always obvious. For example, when you mounted your college network space on the Mac, the system chose a path to it. There are a few ways to find this, and here are two:

- The `df` ("disk full") command lists all of the storage devices currently available to a Unix system, including fixed, removeable, and network devices. If you issue this command, one of the entries likely would contain something like `/Volumes/jcool`, that shows the path to your network storage.
- Mac OS X has a number of unique ways it interacts with its Unix core. One convenient feature is that you can drag and drop icons representing files or folders into a terminal window, and it will paste in the path to that folder or file. When you mounted the network storage space, it should have created an icon on your desktop. Drag that icon to a shell and it should show the path.

? Question 4:
What is the path to the directory where you mounted your college network volume?

You can also list the contents of your working directory with the command `ls`.

? Question 5:
What output do you see when you issue the `ls` command on `mogul.strose.edu`? (Describe it in a sentence, you need not copy and paste the whole thing.)

Other important operations to navigate and modify the directory structure are changing your working directory (`cd`), creating a new directory (`mkdir`), and removing a directory (`rmdir`).

Create a directory in your account for your work for this course (`cs381` might be a good name), and a directory within that directory for this assignment (`lab1` might be a good name).

? Question 6:
Change your working directory to the one you just created and issue the `pwd` command. What does this show as your working directory?

In your shell window and in your home directory (note: you can always reset your working directory to be your home directory by issuing the command `cd` with no parameters), issue this command:

```
    uname -a > linux.txt
```

This will execute the command `uname -a`, which prints a variety of information about the system you are on, and "redirects" the output, which would normally be printed in your terminal window, to the file `linux.txt`.

Output Capture:
`linux.txt` for 1 point(s)

(Note that these "Output Capture" tasks are not asking you to paste the contents of the file into your `lab1.txt` document, but to include the file itself in your submission – more on that in the submission section at the end of the lab.)

Look at the contents of the file `linux.txt` with the command:

```
cat linux.txt
```

Do the same in a Mac terminal window, saving the output of `uname -a` in a file called `mac.txt`.

Output Capture:
`mac.txt` for 1 point(s)

Question 7:
What do you think the information in `linux.txt` and `mac.txt` means?

Question 8:
If your current directory is your home directory on mogul, what is the relative path to your `linux.txt` file? What is the absolute path?

Unix Commands

The Unix command line is a confusing and frustrating place to work unless you know the commands. We will work with many of them throughout the course, but for now, your task is to familiarize yourself with some of the most common.

Identify the function of and experiment with these Unix commands (a few of which you have already used):

```
ls     cd     cp     mv     rm     mkdir   pwd
man    chmod  cat    more   grep   head    tail
ln     find   rmdir  wc     diff   scp     touch
```

Question 9:
Give a one sentence description of each command. (5 points)

Using appropriate commands from the above list, move the `linux.txt` and `mac.txt` files you created in your home directory into the directory you created on mogul for your work for this assignment. Note that one of these involves a local file move, the other a remote copy.

Show that this has worked by issuing the following command from inside of your course directory (but not inside the directory for this assignment):

```
ls -laR > ls.out
```

Then move the file `ls.out` into the directory for this assignment.

Output Capture:
`ls.out` for 3 point(s)

Using the Unix manual, your favorite search engine, or in discussion with your classmates, determine the answers to these questions:

Question 10:
How do you change your working directory to be "one level up" from the current working directory? (Give the command.)

Question 11:
Give two or three different ways to change your working directory to be your home directory. All likely involve the `cd` command, but will take different parameters.

The C Programming Language

C is a widely-used, general purpose language, well-suited to low-level systems programming and scientific computation. Few languages have maintained popularity for as long as C has.

We will initially study it assuming you have Java experience, focusing on the features that make C significantly different from Java. Fortunately, Java borrowed much of its syntax from C, so it is not difficult for a Java programmer to read most C programs.

C++ is a superset of C (that is, any valid C program is also a valid C++ program, just one that doesn't take advantage of the additional features of C++). C++ adds object-oriented feautures. In this course, we will look only at C, not C++.

A Very Simple C Program

We will begin by seeing how to compile and run a very simple C program (`hello.c`) in a Unix environment. Of course, this program prints a slight variation on the traditional "Hello, World!" message.

See Example:
`/home/cs381/examples/hello`

For you to run this, you will need to copy the example to your own directory. Compiling the program will create new files, and you do not have necessary permissions in the file system to create those files in my shared area. Create a directory called `hello` under your directory for this lab and copy the C file into that directory.

Change to that directory and compile and run it:

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

? Question 12:
List the commands you executed to create the directory, copy the file, compile the program, and run the program.

Even in this simple program, there are several things worth noting as a beginning C programmer.

The command

```
gcc hello.c
```

is essentially just another program that can can run at the command prompt. We run a program named `gcc`, which is a free C compiler, part of the GNU Compiler Collection.

This example uses the `gcc` command in its simplest form, where it is used to compile a complete C program that is contained in a single file. In this case, we're asking `gcc` to compile a C program (the *source code*) found in the file `hello.c`. Since we didn't specify what to call the *executable* program produced, `gcc` produces a file `a.out`. The name is `a.out` for historical reasons, and stands for "assembler output".

This is analogous to a Java program consisting of one class (let's say it's the `public class Hello` in `Hello.java` in the same example directory as our `hello.c`) that has nothing but a `main` method. There is an important difference, however. In Java, when you compile, either by pressing a button in your IDE or at the command line with

```
javac Hello.java
```

the file produced is `Hello.class`, which needs to be run inside a Java Virtual Machine (JVM):

```
java Hello
```

It cannot run directly on the computer's hardware. The program `java`, the implementation of the JVM, runs directly on the hardware, but that program runs the Java program on our behalf.

Executables and Search Paths

But... when we compile the `hello.c` program, the `a.out` file produced is an actual executable program that runs on the hardware.

To understand how we run the program and why it's done that way, we need to understand how Unix shells run any program. Basically, to run a program we type its name. But the names it recognizes are only those programs that exist in a set of directories on the system called the *search path*.

The search path is simply a list of directory names, which are searched in the order they're specified for an executable program with the name that was typed at the shell prompt.

The search path is specified using an *environment variable*. Environment variables are used in Unix to provide information to a variety of programs. We can see the set of environment variables assigned to our shell with the `env` command. Run the command and redirect its output to a file `env.out`.

Output Capture:
`env.out` for 1 point(s)

In the file `env.out`, find the line that specifies the `PATH` environment variable. This is the list of directories where your shell will look for programs when you type a name at the prompt.

Using `ls`, look at the contents of some of the directories in your path. Can you find some of the commands you learned earlier in this lab?

So, if we want to figure out which actual executable file will run when we type a name, we can (as the shell would do), search each directory in our search path. The first one we encounter is the one that will execute. That's a lot of work. If we want to know which program will execute if we issue a particular command, we can use the `which` command to find out.

?Question 13:
Which executable file is run when you issue a `gcc` command on mogul?

So when we run one of our own programs, such as the `a.out` we generated from `hello.c`, we type its name. But if you do that on mogul, you will likely get an error message, even those `a.out` is in your working directory:

```
[terescoj@mogul ~]$ a.out
bash: a.out: command not found
```

The problem is that your working directory is not part of your search path! That's why when we ran the program above, we ran it with a slightly different command:

```
[terescoj@mogul hello]$ ./a.out
Hello, C World!
```

The "`./`" before the name tells our shell that we want to run the program in "`.`", which is the Unix shortcut for specifying our home directory. We could just as well give an entire absolute path to our program:

```
[terescoj@mogul hello]$ /home/terescoj/hello/a.out
Hello, C World!
```

We could have programs in our current directory execute without the "`./`" or absolute path, but having "`.`" in a search path is generally considered a bad idea.

We'll be writing lots of C programs, and we probably don't want all of our executables to be named `a.out`. We could certainly rename the ones we want to keep using the `mv` command. But let's just have `gcc` produce an executable with the name we want right way:

```
gcc -o hello hello.c
```

Here, the executable file produced is called `hello` because the `-o` command-line parameter is specified, which tells `gcc` that the next command-line parameter following the `-o` should be used as the output file name.

Details of our Simple Program

Finally, we examine the source code for our `hello.c` program.

At the top of the file, we have a big comment (the equivalent of the class comment in Java) describing what the program does, who wrote it, and when. Your programs should have something similar in each C file.

As with Java, we need to tell C if there are libraries or other code that we will be using within this file. In Java, this is done with `import` statements, but nothing needs to be imported to use parts of some of Java's core API that fall under the `java.lang` package, like `System` and `Math`. In C, we need to inform the compiler for even things like basic input/output. In this case, our program uses a C library function called `printf` to print a message to the screen. For C library functions, the needed information is provided in *header files*, which usually end in `.h`. In this case, we need to include `stdio.h`. How do we know? Well, in this case, it's a header file included by nearly every C program, so you'll just get to know it. But in general, we can check the Unix manual with "`man 3 printf`" and see which header files are listed. We'll learn more about using the Unix manual to find out about C library functions and think more about the actual mechanism employed here later this semester.

Every C program starts its execution by calling the function `main`. The line

```
int main(int argc, char *argv[])
```

is the *function header* for `main`. It corresponds very nicely to the typical `main` method header in a Java application

```
  public static void main(String args[])
```

and plays the same role. The keyword `public` is not needed in C, as it has no notion of data protection like Java or C++. The `static` is not needed because all functions in C are essentially like `static` methods: they have a global scope and anyone can call them. C's `main` has an `int` return instead of `void`, since C uses the return value of the `main` function as a return code that the whole program provides to the operating system. The two command-line parameters are provided to `main`, traditionally declared as `argc`, the number of command-line parameters (including the name of the program itself), and `argv`, an array of pointers to character strings, each of which represents one of the command-line parameters. In this case, we don't use them, but they are often listed anyway as here (though they can be omitted if not used). These provide the same information as Java's array of `Strings`. As we will see soon, C arrays do not come equipped with a length attribute, so `argc` is needed to tell how many entries exist in the array `argv`, and string data is represented by a pointer to an array of `char`, hence the `char *`.

`printf` plays the role of Java's `System.out.print` and results in the string passed as a parameter to be printed to the screen. The `\n` results in a new line. We will see soon that the mechanism for constructing strings to print is quite different from that in Java.

A value of 0 returned from `main` generally indicates a successful execution, while a non-zero return indicates an error condition. So we return a 0. Many C compilers will also allow `main` to have a return type of `void` and no `return` statement, but the `int` return type is normally used.

In general, there is a lot of good news for Java programmers learning C. Much of the syntax of Java was borrowed from C, so a lot of things will look familiar. This includes the basics like `;`-terminated statements and code blocks enclosed in `{}` pairs, most of the arithmetic, boolean, and logical operators, and the names and syntax of control structures (loops and conditionals), and more. Much of our focus this semester will be on those places where important difference exist.

The biggest difference that is evident in this simple program is that there are no classes and methods, just *functions*, which can be called at any time. Any information a function needs to do its job must be provided by its parameters or exist in *global variables* – variable declared outside of every function and which are accessible from all functions.

Practice Program

Write your own C program named `helloloop.c`, much like the "Hello, World" example, but which prints some other message and prints it 10 times inside of a `for` loop. The C `for` loop is much like Java's `for` loop, except that the loop index variable needs to be declared before the loop. That is, a Java loop that looks like this:

```
for (int i=0; i<10; i++) {
  ...
}
```

would need to have the declaration of `i` outside of the loop:

```
int i;

// any other code that happens before the loop

for (i=0; i<10; i++) {
  ...
}
```

Make sure your program compiles and runs on either the Mac or mogul using `gcc`.

This program is worth 10 points.

Note: there are no formal "Programming Assignments" this week.

Submission

Please submit all required files as email attachments to *terescoj@strose.edu* by Monday, December 21, 2015. Be sure to check that you have used the correct file names and that your submission matches all of the submission guidelines listed on the course home page. In order to email your files, you will need to transfer them from mogul to the computer from which you wish to send the email. There are a number of options, including the `sftp` command from the Mac command line.

Grading

This lab is graded out of 35 points.

Grading Breakdown	
Question 1	3
Question 2	1
Question 3	1
Question 4	1
Question 5	1
Question 6	1
`linux.txt` output capture	1
`mac.txt` output capture	1
Question 7	1
Question 8	1
Question 9	5
`ls.out` output capture	3
Question 10	1
Question 11	1
Question 12	1
`env.out` output capture	1
Question 13	1
Practice program	10
Total	35