

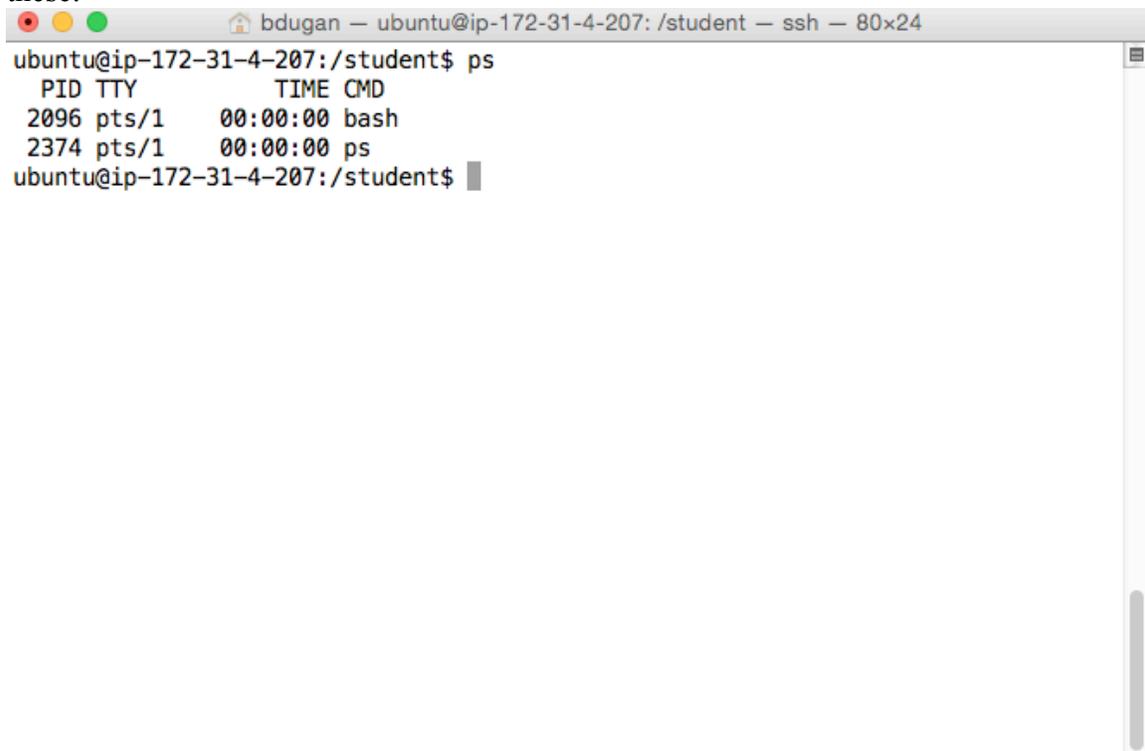
## AWS – running first C++ program

This document describes how edit, compile, link, and run your first linux program using:

- Terminal – a command line interface to your linux os
- Emacs – a popular ide for linux developers
- g++ - a C++ compiler
- Make – the code dependency manager for linux developers

These screen

Type the shell command “ps” which stands for “process status”. This will give a list of processes that are under the control of the shell you just created. You should see two of these:

A terminal window screenshot showing the output of the 'ps' command. The terminal title bar reads 'bdugan - ubuntu@ip-172-31-4-207: /student - ssh - 80x24'. The prompt is 'ubuntu@ip-172-31-4-207:/student\$'. The output of 'ps' is a table with columns 'PID', 'TTY', 'TIME', and 'CMD'.

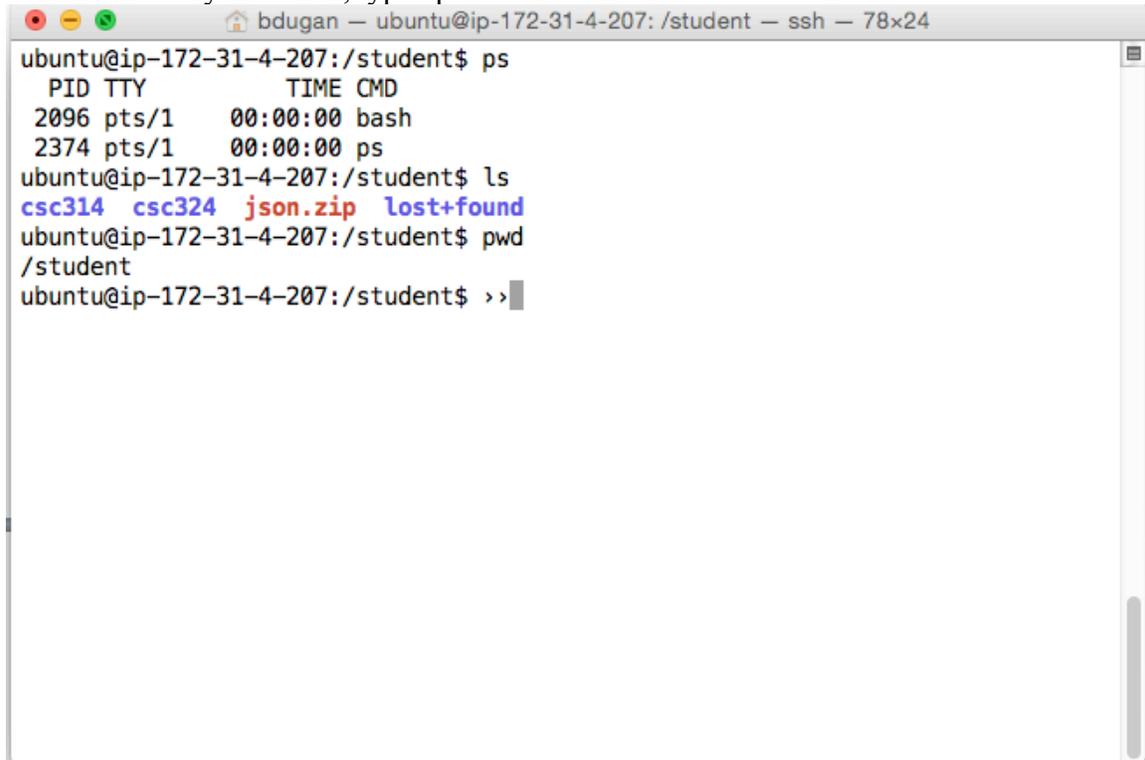
PID	TTY	TIME	CMD
2096	pts/1	00:00:00	bash
2374	pts/1	00:00:00	ps

The prompt returns to 'ubuntu@ip-172-31-4-207:/student\$'.

The first process is “bash”, that’s the name of the shell you are running inside the terminal. The second process is the “ps” command itself..

Now type “ls”. This stands for “list directory contents”. It is similar to the DOS “dir” command. When you type “ls” without any arguments, you are listing files that are in shell’s current directory. Notice filenames and directories appear in different colors. I have a directory called “os” which appears in blue. I have a file called “first.db” which appears in white. I have some other files which appear in red.

At the moment, the files I'm looking at are in my home director which happens to be the default directory you are placed in when you log into the system. To find out your current directory in a shell, type "pwd". You should see "/student >":

A terminal window titled "bdugan — ubuntu@ip-172-31-4-207: /student — ssh — 78x24". The terminal shows the following commands and output:

```
ubuntu@ip-172-31-4-207:/student$ ps
  PID TTY          TIME CMD
 2096 pts/1    00:00:00 bash
 2374 pts/1    00:00:00 ps
ubuntu@ip-172-31-4-207:/student$ ls
csc314  csc324  json.zip  lost+found
ubuntu@ip-172-31-4-207:/student$ pwd
/student
ubuntu@ip-172-31-4-207:/student$ >>
```

Now it's time to create your own directory so we can get started programming. Type in the command:

```
mkdir csc390
```

Now type "ls" to verify that the directory exists.

Now change to this new directory:

```
cd csc390
```

Type "ls" and you will see that there are no files in this directory.

Type "pwd" to verify that you are in the csc390 directory:

```
bdugan — ubuntu@ip-172-31-4-207: /student/csc390 — ssh — 78x24
ubuntu@ip-172-31-4-207:/student$ ls
csc314 csc324 json.zip lost+found
ubuntu@ip-172-31-4-207:/student$
ubuntu@ip-172-31-4-207:/student$ pwd
/student
ubuntu@ip-172-31-4-207:/student$ mkdir csc390
ubuntu@ip-172-31-4-207:/student$ ls
csc314 csc324 csc390 json.zip lost+found
ubuntu@ip-172-31-4-207:/student$ cd csc390
ubuntu@ip-172-31-4-207:/student/csc390$ ls
ubuntu@ip-172-31-4-207:/student/csc390$ pwd
/student/csc390
ubuntu@ip-172-31-4-207:/student/csc390$ >
```

Now it's time to bring up the editor you'll be using for your programs in this class. Type the following command:

```
emacs
```

Let's start by creating the files you'll need for your "Hello World" program. To create a file, make sure that the editor is active by clicking on it and type the following sequence:

```
<CTRL>-x <CTRL>-f
```

Hold the <CTRL> key down and at the same time press the "x" key. Then hold the <CTRL> key down and press the "f" key. You should see something like this:

```
bdugan — ubuntu@ip-172-31-4-207: /student/csc390 — ssh — 78x24
File Edit Options Buffers Tools Minibuf Help
Welcome to GNU Emacs, one component of the GNU/Linux operating system.

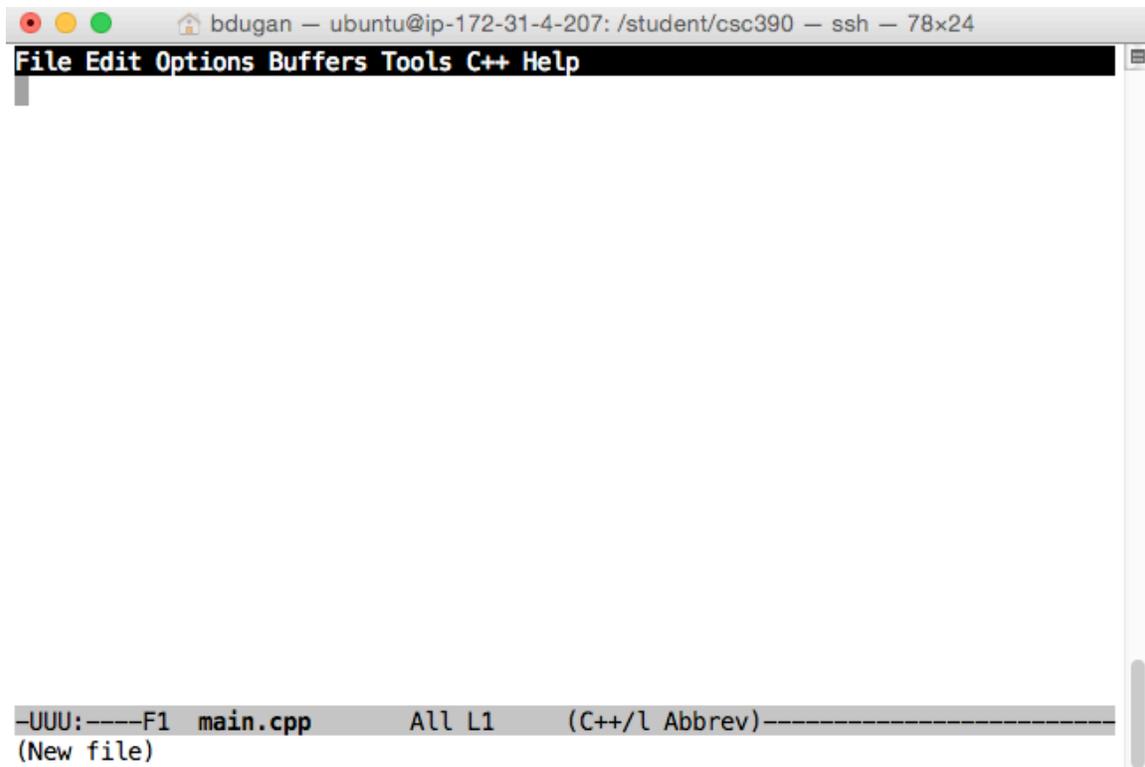
Get help          C-h (Hold down CTRL and press h)
Emacs manual      C-h r          Browse manuals   C-h i
Emacs tutorial    C-h t          Undo changes     C-x u
Buy manuals       C-h RET        Exit Emacs       C-x C-c
Activate menubar  M-`
(`C-' means use the CTRL key. `M-' means use the Meta (or Alt) key.
If you have no Meta key, you may instead type ESC followed by the character.)
Useful tasks:
Visit New File    Open Home Directory
Customize Startup Open *scratch* buffer

GNU Emacs 23.4.1 (x86_64-pc-linux-gnu, GTK+ Version 2.24.18)
of 2013-06-17 on komainu, modified by Debian
Copyright (C) 2012 Free Software Foundation, Inc.

GNU Emacs comes with ABSOLUTELY NO WARRANTY; type C-h C-w for full details.
Emacs is Free Software--Free as in Freedom--so you can redistribute copies
of Emacs and modify it; type C-h C-c to see the conditions.
Type C-h C-o for information on getting the latest version.
-UUU:%%--F1 *GNU Emacs* Top L1 (Fundamental)-----
Find file: /student/csc390/
```

emacs is prompting you to find a file in the subdirectory “~/operatingsystems”. That “~” is a special character which in unix parlance represents your home directory... it is the equivalent of “/home/<username>” or in my case “/home/bdugan”. We don’t want to find a file, we want to create a new one, so type:

```
main.cpp <ENTER>
```



Notice that emacs has entered “cc-mode” and placed a C++ drop down item in the menu bar at the top of the editor. Emacs is ready for you to start writing some C++ code. The cursor is now inside the main window of the editor and you are ready to start writing your first program!

Most of the editing keys on your keyboard will work in emacs as you’d expect them too. For example, when you press the <BACKSPACE> key, you will delete the character to the left of the cursor and move the cursor back one space.

Emacs also has some built in keyboard sequences that do the same thing:

- <CTRL>-f – move cursor forward one character
- <CTRL>-b – move cursor backward one character
- <CTRL>-d – delete character at cursor position, slide text to right of cursor position one character to left
- <CTRL>-n - move cursor one line down
- <CTRL>-p – move cursor one line up
- <CTRL>-g – whenever you get in trouble, type this a couple of times to get back to the main editing buffer

Why would you EVER want to learn these sequences? As it turns out there’s a pretty good reason. Emacs is ubiquitous. It runs on almost every computer hardware platform on the planet. Every computer has a slightly different keyboard. You may not be guaranteed, for example, that a computer has an <UPARROW> key! Most computers have the <CTRL> key and a “p” key however.

Let's write our first program:

```
#include <iostream.h>
#include <main.h>

int main(void)
{
    cout << "Hello World!" << endl;
    return 1;
}
```

It should look something like this in your editor:

A screenshot of a C++ IDE window. The title bar shows 'bdugan - ubuntu@ip-172-31-4-207: /student/csc390 - ssh - 78x24'. The menu bar includes 'File Edit Options Buffers Tools C++ Help'. The code editor contains the following code:

```
#include <iostream>
#include <main.h>

using namespace std;

int main(void)
{
    cout << "Hello World" << endl;
    return 1;
}
```

The status bar at the bottom shows '-UU-:\*\*\*-F1 main.cpp All L1 (C++/l Abbrev)-----'.

Save the file by issuing the command:

<CTRL>-x <CTRL>-s

One thing you should notice about the #include file names is that the “.h” extension on the header file is specified explicitly. This is a nuance of the g++ compiler which you’ll be using for the class. Don’t forget to put that extension on your header files.

OK. Now create the “main.h” file and add the following comment:

```
// Nothing goes here yet...
```

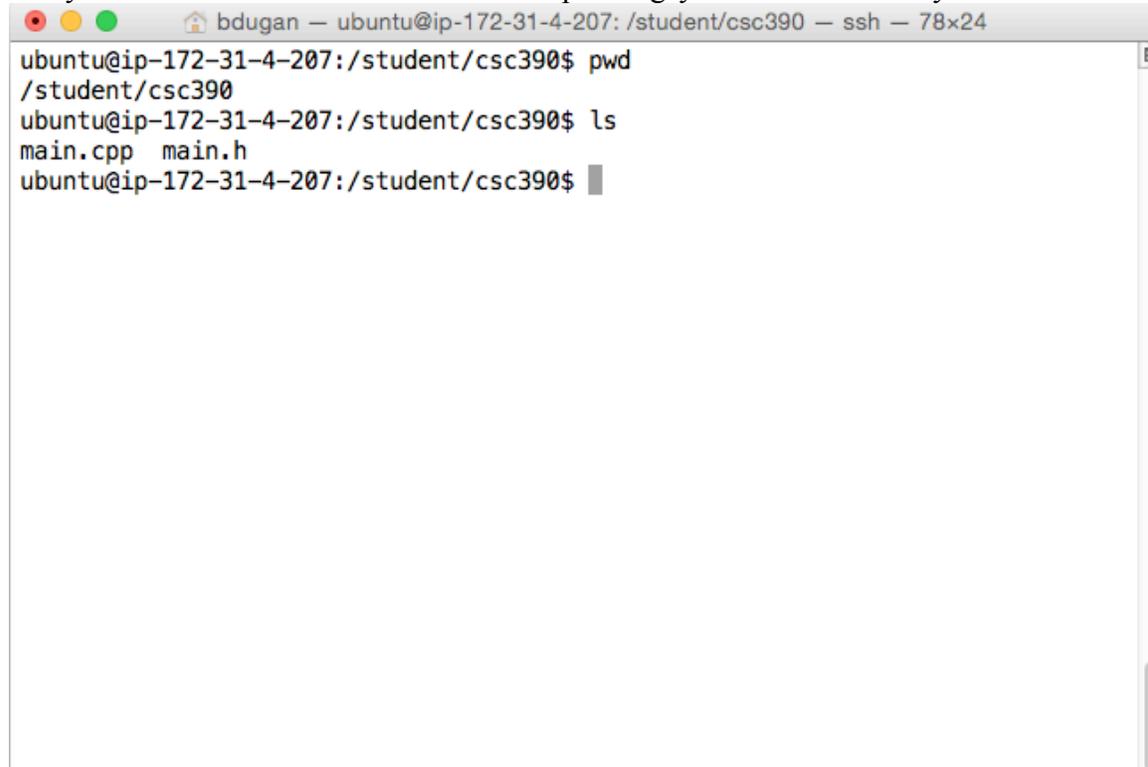
Then save the file.

## Compile, Link, and Run

Now you are ready to compile, link, and run your program.

First, we'll do this by hand. Then we'll automate everything using a makefile.

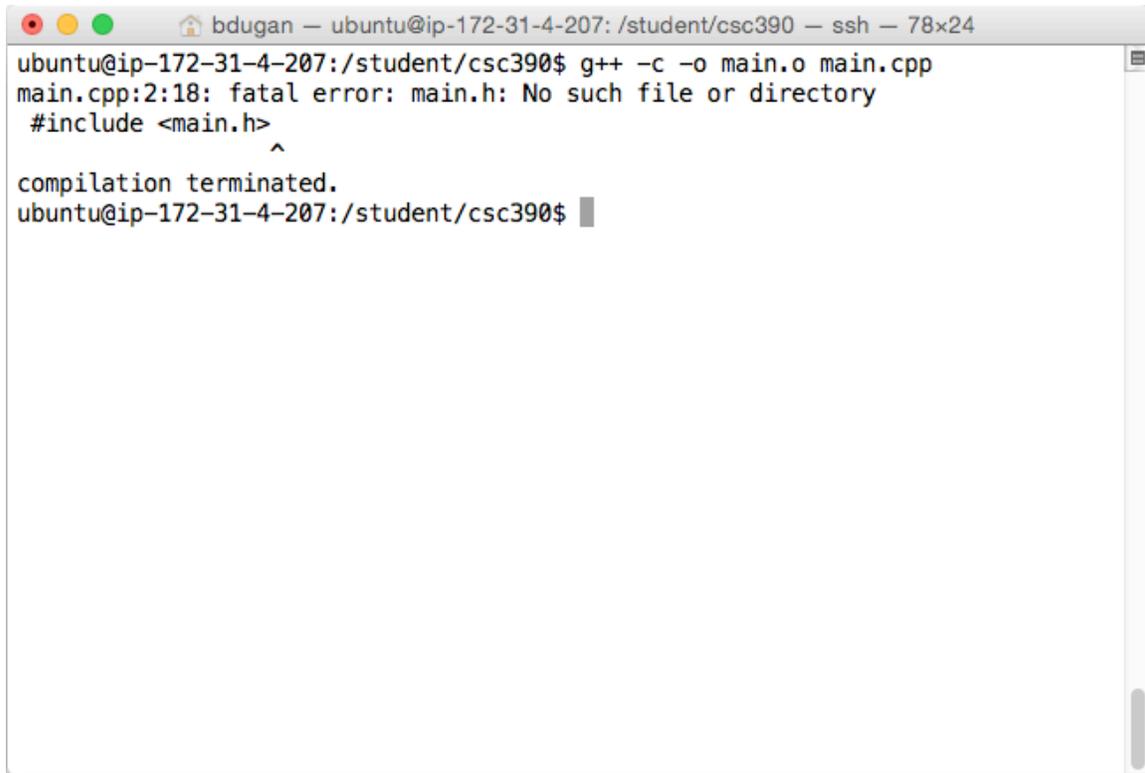
Leave the editor and click on the x-term window you created earlier. Type "ls" to verify that you have added two new files to the "operatingsystems" subdirectory:

A terminal window titled "bdugan — ubuntu@ip-172-31-4-207: /student/csc390 — ssh — 78x24". The terminal shows the following commands and output:

```
ubuntu@ip-172-31-4-207:/student/csc390$ pwd
/student/csc390
ubuntu@ip-172-31-4-207:/student/csc390$ ls
main.cpp main.h
ubuntu@ip-172-31-4-207:/student/csc390$
```

Now invoke the g++ compiler to compile your program:

```
g++ -c -o main.o main.cpp
```

A terminal window titled "bdugan — ubuntu@ip-172-31-4-207: /student/csc390 — ssh — 78x24". The terminal shows the command `g++ -c -o main.o main.cpp` being executed. The output is: `main.cpp:2:18: fatal error: main.h: No such file or directory`, followed by `#include <main.h>` with an arrow pointing to the opening angle bracket. Below that, it says `compilation terminated.` and the prompt `ubuntu@ip-172-31-4-207:/student/csc390$` is shown again.

```
ubuntu@ip-172-31-4-207:/student/csc390$ g++ -c -o main.o main.cpp
main.cpp:2:18: fatal error: main.h: No such file or directory
#include <main.h>
           ^
compilation terminated.
ubuntu@ip-172-31-4-207:/student/csc390$
```

What happened? It looks like the compilation failed because the compiler couldn't find the include file "main.h". You can fix this by adding "-I." to the compiler command:

```
g++ -c -I. -o main.o main.cpp
```

This command tells the compiler to compile main.cpp (via the "-c" argument) and produce an object file "main.o" as a result (via the "-o" argument). It also tells the compiler that in addition to any default directories, it should look in the current directory for include files (via the "-I." argument). Type "ls" to verify that you now have a "main.o" object file:

```
bdugan — ubuntu@ip-172-31-4-207: /student/csc390 — ssh — 78x24
ubuntu@ip-172-31-4-207:/student/csc390$ touch main.cpp
ubuntu@ip-172-31-4-207:/student/csc390$ g++ -c -o main.o main.cpp
main.cpp:2:18: fatal error: main.h: No such file or directory
   #include <main.h>
                 ^
compilation terminated.
ubuntu@ip-172-31-4-207:/student/csc390$ g++ -c -I. -o main.o main.cpp
ubuntu@ip-172-31-4-207:/student/csc390$ ls
main.cpp  main.h  main.o
ubuntu@ip-172-31-4-207:/student/csc390$
```

We can't run the program yet because we have an OBJECT file (main.o) not an EXECUTABLE file. To create an executable file, we need to run the linker:

```
g++ main.o -o main
```

This command tells the linker to link the object file main.o with all default libraries and produce an executable named "main" (via the -o parameter). Type "ls" again to show that main has been created:

```
bdugan — ubuntu@ip-172-31-4-207: /student/csc390 — ssh — 78x24
ubuntu@ip-172-31-4-207:/student/csc390$ touch main.cpp
ubuntu@ip-172-31-4-207:/student/csc390$ g++ -c -o main.o main.cpp
main.cpp:2:18: fatal error: main.h: No such file or directory
  #include <main.h>
                 ^
compilation terminated.
ubuntu@ip-172-31-4-207:/student/csc390$ g++ -c -I. -o main.o main.cpp
ubuntu@ip-172-31-4-207:/student/csc390$ ls
main.cpp  main.h  main.o
ubuntu@ip-172-31-4-207:/student/csc390$ g++ main.o -o main
ubuntu@ip-172-31-4-207:/student/csc390$ ls
main  main.cpp  main.h  main.o
ubuntu@ip-172-31-4-207:/student/csc390$
```

## ***PATH and SHELL VARIABLES***

Now let's run the program! At the shell command prompt type:

```
main <ENTER>
```

You should see something that looks like this:

```
bdugan — ubuntu@ip-172-31-4-207: /student/csc390 — ssh — 78x24
ubuntu@ip-172-31-4-207:/student/csc390$ touch main.cpp
ubuntu@ip-172-31-4-207:/student/csc390$ g++ -c -o main.o main.cpp
main.cpp:2:18: fatal error: main.h: No such file or directory
  #include <main.h>
                 ^
compilation terminated.
ubuntu@ip-172-31-4-207:/student/csc390$ g++ -c -I. -o main.o main.cpp
ubuntu@ip-172-31-4-207:/student/csc390$ ls
main.cpp main.h main.o
ubuntu@ip-172-31-4-207:/student/csc390$ g++ main.o -o main
ubuntu@ip-172-31-4-207:/student/csc390$ ls
main main.cpp main.h main.o
ubuntu@ip-172-31-4-207:/student/csc390$ main
No command 'main' found, did you mean:
  Command 'mail' from package 'mailutils' (universe)
  Command 'cain' from package 'cain' (universe)
  Command 'moin' from package 'python-moinmoin' (main)
  Command 'man' from package 'man-db' (main)
  Command 'rain' from package 'bsdgames' (universe)
  Command 'amin' from package 'orville-write' (universe)
main: command not found
ubuntu@ip-172-31-4-207:/student/csc390$
```

What happened? The shell returned an error stating that the command was not found. This happened because the shell did not find the command in list of directories it was supposed to search for commands. Your “operatingsystems” directory is not in this list. This list is stored in something called a shell environment variable. The name of the environment variable is called “PATH”. To see the list type:

```
echo $PATH
```

```
bdugan — ubuntu@ip-172-31-4-207: /student/csc390 — ssh — 78x24
ubuntu@ip-172-31-4-207:/student/csc390$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/l
ocal/games
ubuntu@ip-172-31-4-207:/student/csc390$
```

So the directories “/bin” and “/usr/bin” and “/bin”, etc. are searched by the shell for a match to the commands that you type at the shell prompt. A couple of interesting things to note here. First, the shell will match the command with the FIRST executable it finds in the list of directories provided in the PATH. Second, “operatingsystems” is not listed as one of the directories. Third, those slashes “/” seem to be going the wrong way. Microsoft operating systems also use slashes to specify the location of directories and files, but the slashes go the other way: “\”. As it turns out, it’s Microsoft slashes that are going the wrong way... Microsoft copied many of the basic shell command and file management mechanisms from Unix when they created DOS and changed them slightly. This is an example of one of those changes.

There are lots of environment variables that have been set by default for your shell. If you want to see the list type:

```
env <ENTER>
```

These variables have different meanings and uses. For example there’s one variable “LOGNAME” which will tell you the name of the user who created the shell. You can alter these variables or add your own. You can also write programs that check these variables.

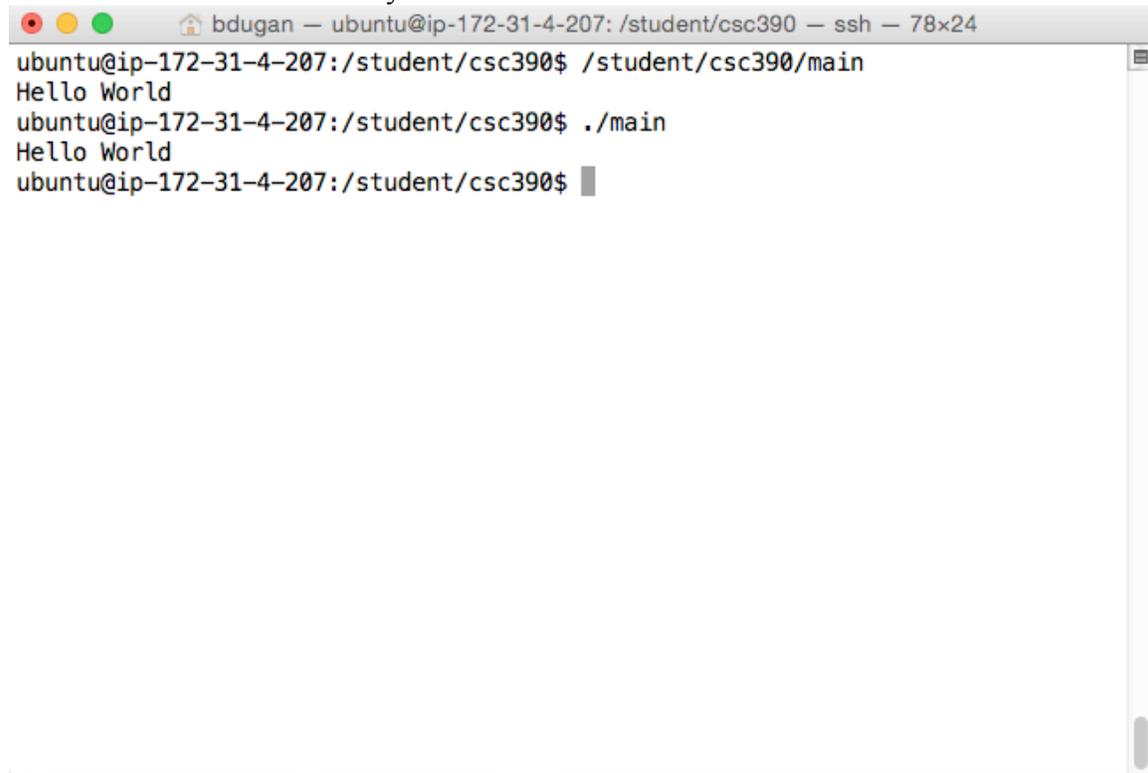
OK. Back to your program. We need to tell the shell EXACTLY where to find your main program. Type in the following command:

```
/student/csc390/main
```

Where <username> is the name of the username you used to log into the workstation. It worked! Unix users hate typing a lot so you can shorten the command to just:

```
./main
```

The “./” means start at the current directory. So “./main” means execute the command “main” in the current directory.

A terminal window with a title bar that reads "bdugan — ubuntu@ip-172-31-4-207: /student/csc390 — ssh — 78x24". The terminal content shows the following sequence of commands and output:

```
ubuntu@ip-172-31-4-207:/student/csc390$ /student/csc390/main
Hello World
ubuntu@ip-172-31-4-207:/student/csc390$ ./main
Hello World
ubuntu@ip-172-31-4-207:/student/csc390$ █
```

## **Makefiles**

Typing in the compiler and linker commands over and over each time you want to build an executable is tedious. It can also become time consuming as you build large programs that require lots of source and header files. It would be nice to have a mechanism which only compiles the files you’ve changed since the last time your executable was created.

Most of you have been using sophisticated development environments like Code Warrior, Turbo-C++, and Visual C++ which do this kind of thing automatically for you. You may have not even been aware of it!

For the most part, these environments owe their origins to editors like emacs and program building systems like makefiles.

You use a makefile to identify and act upon:

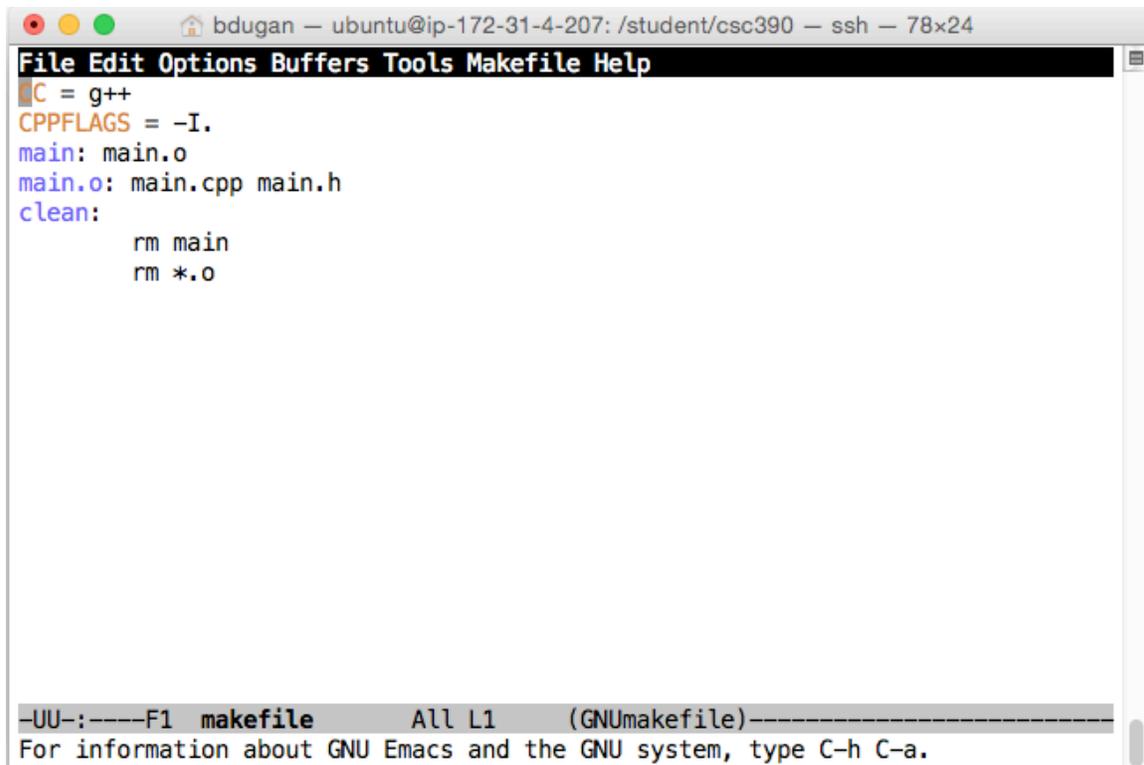
- source files you want to compile
- compiler you want to use

- parameters to compiler
- object files you want to link
- linker you want to use
- parameters to linker

Makefiles have a weird and picky syntax. I still get a little mystified by them from time to time so I'm going to give you the makefile for our simple program verbatim. You should go through the makefile tutorial on the class website to get more details and a better understanding. You'll need this understanding for the rest of the projects you complete in this course.

Go back to the emacs editor and create a new file called "makefile". Put the following in the file and save the file:

```
CC = g++
CPPFLAGS = -I.
main: main.o
main.o: main.cpp main.h
clean:
    rm main
    rm *.o
```



```
File Edit Options Buffers Tools Makefile Help
CC = g++
CPPFLAGS = -I.
main: main.o
main.o: main.cpp main.h
clean:
    rm main
    rm *.o

-UU-:----F1 makefile All L1 (GNUmakefile)-----
For information about GNU Emacs and the GNU system, type C-h C-a.
```

The first line “CC = g++” tells the make program that the “g++” compiler and linker will be used to make programs with this makefile. The second line “CPPFLAGS = -I.” tells the compiler to look in the current directory for include files. The next line “main: main.o” states that an executable “main” will be created from the object file “main.o”. Because nothing appears beneath this line, the default link command will be used when creating this executable. The next line “main.o: main.cpp main.h” states that the object file “main.o” should be built from two files: “main.cpp” and “main.h”. Again because nothing appears beneath this line, it also states that the default compile command should be used when creating this object file.

The final line “clean:” states that if the user types “make clean” two commands are supposed to be executed “rm main” and “rm \*.o”. “rm” is a shell command to remove a file. So these two commands will remove the main program executable and any object files in the current directory providing you with a clean slate for building your program.

OK. Let’s try using this makefile. First, make sure you save it. Go back to the shell and use “ls” again to make sure that it exists:

Now execute the makefile by typing the command:

```
make <ENTER>
```

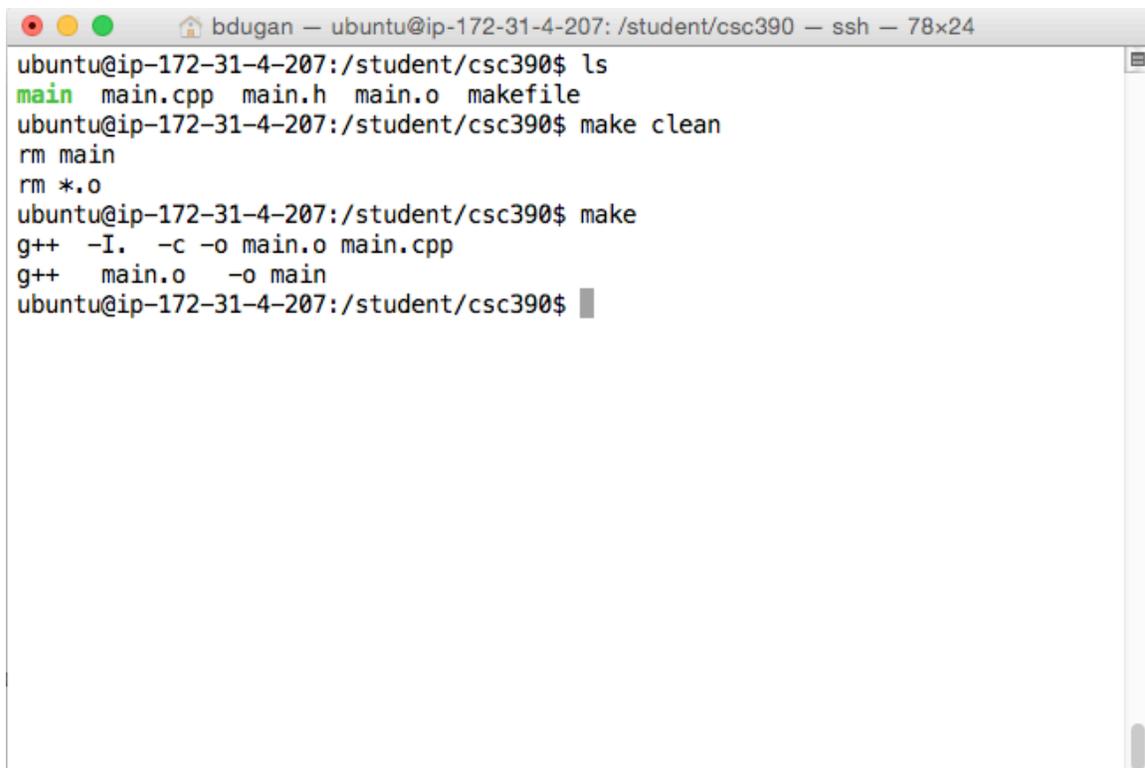
What happened? The make program determined that nothing needed to be done based on the date/times that the files used to build main were created. The timestamps for the main.cpp and main.h files were older than then main.o file. The main.o timestamp was compared against the main executable timestamp and it was found to be older. Nothing needed to be done!

Ok... we want to force the make program to do some work. Type:

```
make clean <ENTER>
```

This should erase the main executable and any object files. Now type:

```
make <ENTER>
```

A terminal window titled 'bdugan - ubuntu@ip-172-31-4-207: /student/csc390 - ssh - 78x24'. The terminal shows the following commands and output:

```
ubuntu@ip-172-31-4-207:/student/csc390$ ls
main main.cpp main.h main.o makefile
ubuntu@ip-172-31-4-207:/student/csc390$ make clean
rm main
rm *.o
ubuntu@ip-172-31-4-207:/student/csc390$ make
g++ -I. -c -o main.o main.cpp
g++ main.o -o main
ubuntu@ip-172-31-4-207:/student/csc390$
```

Run your “main” program one more time to verify that it was build correctly.

Submitting Your Work to elearn:

You need to aggregate the files and directories you use into a single file to upload to elearn. Goto the directory above your programming project, type ls to verify you are above the directory, then type this command:

```
tar -cvf csc390.tar ./csc390
```

You should see output indicating which files are going to be placed in the “.tar” file (tar stands for terse archive file). Type “ls” to verify that the csc390.tar file now exists:

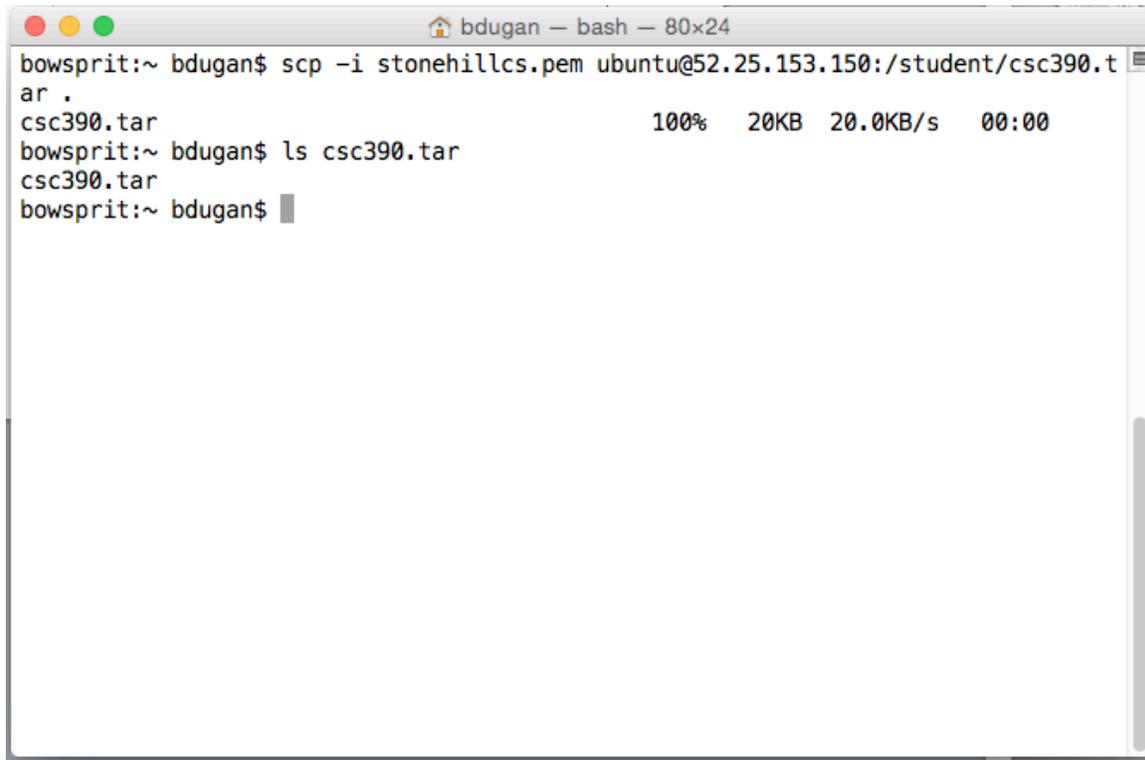


```
bdugan — ubuntu@ip-172-31-4-207: /student — ssh — 84x24
ubuntu@ip-172-31-4-207:/student/csc390$ ls
main main.cpp main.h main.o makefile
ubuntu@ip-172-31-4-207:/student/csc390$ cd ..
ubuntu@ip-172-31-4-207:/student$ ls
csc314 csc324 csc390 json.zip lost+found
ubuntu@ip-172-31-4-207:/student$ tar -cvf csc390.tar ./csc390
./csc390/
./csc390/main.cpp
./csc390/makefile
./csc390/main.h
./csc390/main.o
./csc390/main
ubuntu@ip-172-31-4-207:/student$ ls
csc314 csc324 csc390 csc390.tar json.zip lost+found
ubuntu@ip-172-31-4-207:/student$
```

ON YOUR LOCAL MACHINE NOT YOUR VIRTUAL MACHINE (for Linux & OSX) type the following command:

```
scp -i stonehillcs.pem ubuntu@52.25.153.150:/student/csc390.tar .
```

The command will copy the “.tar” file from your virtual machine to your local machine. Use “ls” or a graphical file browser on your local machine to verify that the file exists:



```
bdugan — bash — 80x24
bowsprit:~ bdugan$ scp -i stonehillcs.pem ubuntu@52.25.153.150:/student/csc390.t
ar .
csc390.tar                               100%  20KB  20.0KB/s   00:00
bowsprit:~ bdugan$ ls csc390.tar
csc390.tar
bowsprit:~ bdugan$
```

ON YOUR LOCAL MACHINE NOT YOUR VIRTUAL MACHINE (for Windows) you have a number options for tools that use the scp protocol:

<http://stackoverflow.com/questions/8975798/scp-w-ssh-copying-a-local-file-from-windows-to-a-remote-server-using-scp>