The ECF Unix Environment

1 Summary

This document provides information about how to log in to ECF, manipulate files and folders, change permissions, compile programs without using an IDE, submit assignments, and manage archives (using zip).

To avoid a grade of zero for your labs, please read carefully the section on submitting work.

Also, please read and follow the instructions on setting file permissions to avoid academic offenses related to copying of your work.

2 Logging Into ECF

Before you can gain access to the ECF computer system, you must have your *login name* and a *password* to log onto the system. Your login name (or just *login*) is a 6-8 character string that is used to identify you to the computer system. Your password is an 8 character string known only to you and is used to provide security for your computer account. No one knows what your password is except you, not even the ECF system! **You must never share your password with anyone**.

Once you log in, you will likely be presented with some introductory messages and announcements from ECF, and then you will get an *xterm* window. If you do not get the xterm window you can start one by clicking on "Applications" then "System Tools", and then "Terminal". The xterm window has the UNIX *prompt*: "p*xxx*.ecf%" (e.g., p120.ecf% or p150.ecf %) indicating that the system is ready to accept your commands. In the remainder of this document, we will use just "%" as the prompt and it will appear in front of all commands you type, but of course you will not type it.

3 The UNIX File System

3.1 The File Structure

The UNIX file system is organized into a tree-like structure of directories. At the top of the tree is the *root* directory of the system, designated by a "`/`". Various system directories, named "`bin`", "`local`", "`lib`", "`u`", etc are contained in the root directory. In turn, files and other directories exist in each of these directories, as shown in Figure 1.

Your files and directories exist in the system under your *home directory*, which has the same name as your login name (assumed to be `ast` for the remainder of this document). This is also shown in Figure 1: your files are in the directory called `ast`, which exists in the directory called `1T1`, which in turn is in the directory called `u` that exists in `/`.

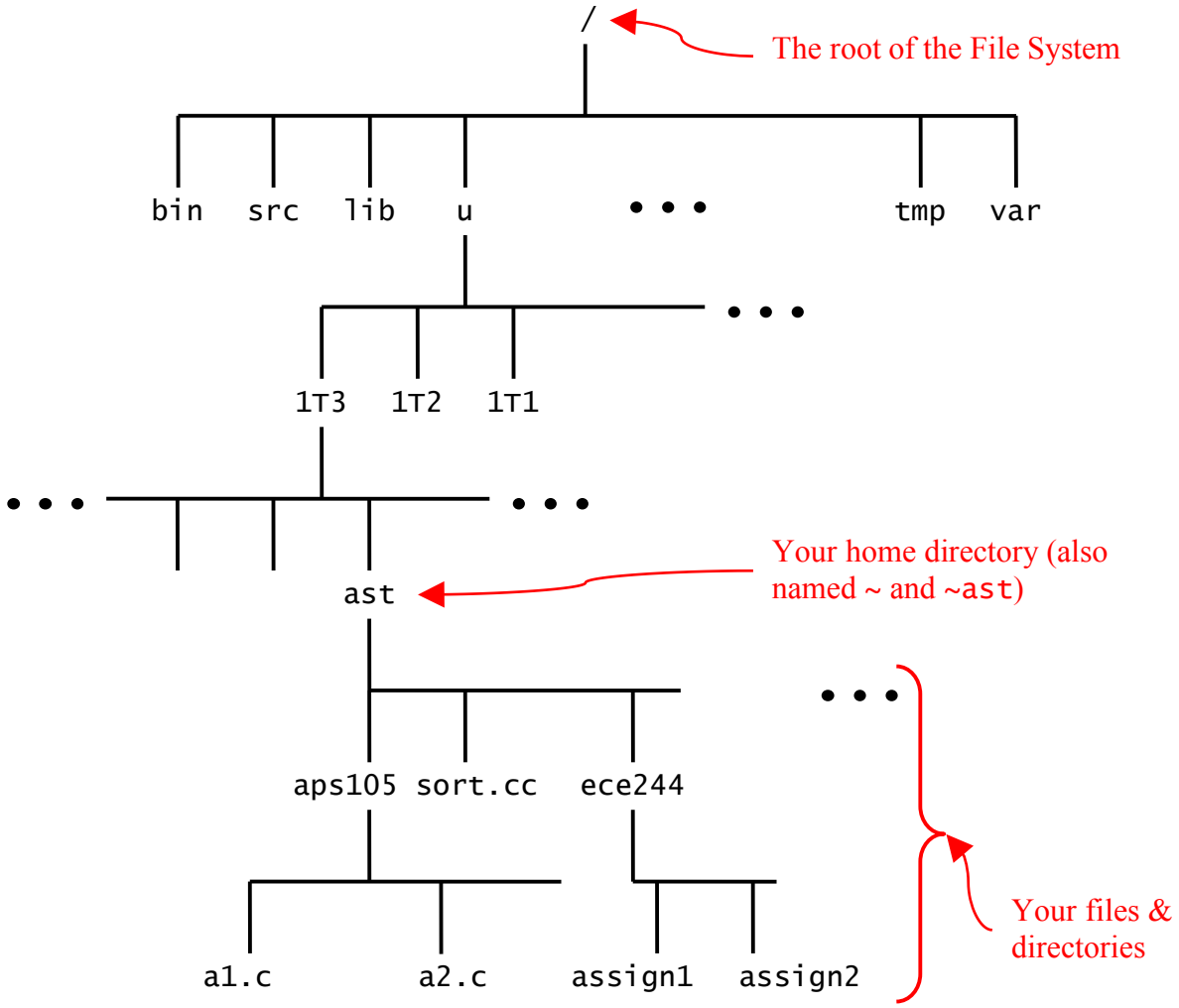

Figure 1: An example UNIX directory structure.

3.2 The Working Directory

When you are logged into the ECF system, you are always associated with one directory or another. The directory you are currently associated with is called your *working directory*. When you first login, your home directory is your working directory. You can then change your working directory using the `cd` command (explained later). At any time, you can find out what your working directory is using the print-working-directory command, `pwd`, as follows:

```
% pwd
```

The command will print the name of your current working directory to the screen.

3.3 File and directory locations

The location of a file or a directory is specified using a *path* through the tree. This path can be *full* (or *absolute*) starting from the root of the tree "/", or *relative* to the current directory, or to some other directory. Thus, in Figure 1, the file `a1.c` can be specified in a number of ways, assuming that your home directory is your working directory:

`/u/1T1/ast/aps105/a1.c`	full or absolute
`aps105/a1.c`	relative to the working directory
`~ast/aps105/a1.c`	relative to the home directory of `ast`
`~/aps105/a1.c`	relative to your home directory

Note how `~ast` specifies the home directory of user "`ast`" and how "`~`" by itself specifies your home directory.

UNIX also provides two short-hands: "`.`" refers to the working directory, and "`..`" refers to the parent directory of the working directory. These short-hands are useful in specifying relative paths. For example, in Figure 1, if the working directory is `aps105`, then the file `a1.c` can also be specified as `./a1.c`, and the file `sort.cc` can be specified as `../sort.cc`.

3.4 Listing Directory Contents

To obtain a listing of the names of all of the files in a directory, you can use the `ls` command:

```
% ls
```

Thus, using `ls` when you first login will print a list of all the files in your home directory.

3.5 Creating Directories

In order to keep your files better-organized, you should create directories to hold files relating to various subjects, courses, etc. For example, to create a new directory, called `newdir`, in your home directory, use the `mkdir` command at the prompt:

```
% mkdir newdir
```

Since `newdir` is contained in your home directory, `newdir` is referred to as a *subdirectory* of your home directory, and your home directory is referred to as the *parent directory* of `newdir`. Similarly in Figure 1, `aps105` is the parent directory of the file `a2.c` while `ece244` is a subdirectory of the home directory `ast`.

3.6 Changing the Working Directory

To work within a directory, you must use the change-directory command, `cd`, to make it your working directory. The `cd` command has the following syntax:

```
cd dirlocation
```

where `dirlocation` is the location of the directory you wish to make your working directory, specified using a path in the directory tree as described above. For example, in Figure 1, if your home directory is the working directory, and you can make `assign1` your working directory, by typing:

```
% cd ece244/assign1
```

You may also accomplish the same thing by specifying the full path to the directory as follows:

```
% cd /u/1T1/ast/ece244/assign1
```

If your working directory is `assign1`, and you can make `ece244` your working directory by typing:

```
% cd ..
```

which takes you to the parent directory of your current directory; i.e., `ece244`.

Finally, you can always return to your home directory by typing:

```
% cd
```

with no directory location specified.

It is important to always know what your current working directory is (using the `pwd` command, described earlier), and how to move to subdirectories of it, or to move to its parent directory.

3.7 Moving, Copying, Deleting Files

Using UNIX commands, you can type commands to list your files, move, copy, delete, etc. For example, the `cp` command is used to create a copy (duplicate) of a file. It has the following syntax:

```
cp file1 file2
```

This takes a file named `file1` and creates a new file called `file2` that is an exact duplicate of `file1`.

The `mv` command is used to move (or rename) a file, and has the following syntax:

```
mv file1 file2
```

This takes a file named `file1` and creates a new file called `file2` that is an exact duplicate of `file1`. The original `file1` is removed.

Finally, the `rm` command is used to delete (or remove) a file:

```
rm file
```

Be careful when using the `rm` command; files that you remove cannot be retrieved. Thus, it pays to use the `rm` command with the "`-i`" option as follows:

```
rm –i file
```

The command will prompt for confirmation before the file is deleted.

3.8 Protecting Your Files

Every UNIX file has an owner (and group) associated with it. You are the owner of all files and directories in your home directory. As the owner you may choose to allow or deny others to

view your files. In order to protect your assignments from being copied by others, you should use the `chmod` command to protect you files. When you create a new directory `newdir`, use the `chmod` command as follows (assuming your working directory is the parent of `newdir`):

```
% chmod –R go-rwx newdir
```

This prevents others from reading or searching through `newdir`. You can use the command on any existing directory as well to protect it and all the files contained in it.

Three sets of permissions are set: user, group, and other (abbreviated above as u/g/o). The permission types that can be granted are read, write, and execute. With this understanding, the line above can be interpreted as followed: from group and other, subtract read/write/execute privileges. To check that we have set the privileges correctly, use the ls command in its long (-l) form to get more details:

```
% ls -l
```

This will give output similar to below:

```
total 0
drwxr-xr-x  27 tsa       eceprof  918 Aug 16 16:18 documents
drwxr-xr-x   7 tsa       eceprof  238 Aug 18 20:01 lab1
drwxr-xr-x  13 tsa       eceprof  442 Aug 16 18:32 lab2
drwxr-xr-x   4 tsa       eceprof  136 Aug 16 14:24 tutorial
```

The important part is the sequence of letters in the left column. It starts with ‘d’ to indicate that the entry is a directory. Next, the rwx indicates that the user (tsa) can read, write, and execute. Next, r-x indicates that members of the group “eceprof” can read and execute. The final r-x means that others (not user “tsa nor group eceprof”) can read and execute. Normally execute (‘x’) means that you are allowed to run the program if it’s a binary file or a script – in the case of directory, execute actually means you can enter the directory to view its contents.

Note that your directories should read drwx------, and your files should read -rw------- such that only the owner (you) can read them.

3.10 Getting Help on UNIX Commands

You can get more information on any UNIX command by using the `man` command (for manual page). For example, type

```
% man ls
```

to get information on the `ls` command. As the information is displayed, `--More--` appears at the bottom of the screen. At this point, you can press the spacebar to display the next screen, or press `q` to quit.

Indeed, if you wish to find out more about the man command, type:

```
% man man
```

3.11 Using zip/unzip to compress/decompress multiple files

Sometimes, it is more convenient to save multiple files into a single file, eg. for transmission over e-mail or downloading via the web. The common ZIP program and file format for Windows often accomplishes this, and has been implemented for UNIX as well.

Zip, which uses the extension .zip, stores and compresses multiple files and their folder hierarchy within a single file, which can later be expanded. The command is `zip` to compress and `unzip` to decompress

Some of the instructional material for the course is distributed this way. To get the files, simply download the relevant zip file (here called foo.zip), place it in your ece244 folder, and run:

```
% unzip foo.zip
```

You can get more information by using the man command, or using the `--help` option.

4. Useful Tools

Please note that the functions of many tools listed below (editor, compiler, terminal, debugger) can all be accessed from within the NetBeans Integrated Development Environment (IDE) for greater simplicity and convenience. Lab materials assume that you will be using NetBeans, though you may want to learn to use the individual components as well.

4.1 The Terminal

As noted previously, the terminal and command line remain very important for Unix users. While graphical user interfaces (GUIs) can be very handy, some things can only be done or remain easiest for a skilled user to do from the command line. It is also useful when logging in to computers remotely since text is much more compact to transmit than the instructions to draw the GUI.

Another important use for the terminal is creating and testing *shell scripts*, which issue a series of commands to automate certain repetitive tasks. By reducing the effort to accomplish common tasks, effective terminal use and scripting can make you a more efficient programmer. As you advance in your programming career, shell scripting is a very important skill to learn. The `submitece244f` and `exercise` commands are so implemented – thus saving you from typing a long and error-prone series of commands to submit your assignment.

Note that you may have several terminal windows open at a time. For instance, you may have one for manipulating files, one for debugging, one for compiling, several running other jobs in the background, etc.

4.2 Text Editor

In addition to source code files, programmers often have to read and write a number of text files – for test cases, Makefiles, data, documentation, etc. Text editors come in many different styles and are a matter of personal preference. What is important is that you find one that works well for you, and with which you are highly productive. Popular instances include:

gvim	A graphical and customizable editor. It has mouse support, but you can also accomplish an incredible number of things from the keyboard. Very fast and productive once you learn, but has a steep learning curve.
emacs	An extremely stable, long-standing open-source project, Emacs is very customizable and has numerous plugins but can be very hard to learn initially. It would be considered a "classic" but still sees wide use today.
gedit	Has many of the same basic features as gvim and emacs but may be less daunting to learn
netbeans	The NetBeans IDE includes a powerful text editor.

You can try out the various editors by typing their name at the command prompt. Features commonly considered "must-have" include:

- syntax highlighting: the ability to recognize and highlight language key words
- search & replace (preferably with regular expressions)
- auto-indent
- line numbering
- jump to line number
- bracket matching (ability to highlight or jump to a matching bracket or parenthesis)

Other popular productivity items sometimes found include:

- folding (the ability to 'collapse' a function definition into a single line, or 'expand' it back)
- syntax-error highlighting (find syntax errors before compiling)
- source browsing (navigation by variable/function/class definition)
- auto-completion
- refactoring (the ability to rename variables or methods in a syntax- and scope-aware way across files)

4.3 Compiler

For our C++ programs to run, the source code file(s) must be *compiled* and *linked* with the *standard libraries*. Compilation is the process of converting human-readable source code into binary instructions that the computer can execute directly (*machine code*). The standard libraries are a set of functions that all C++ implementations agree to provide. They include, for instance, the provisions for allocating and de-allocating memory, accessing files, printing to the terminal, standard math functions, and many more. Linking means taking function calls that cannot be found within the source files, finding them in a library, and inserting a pointer to the library file at the appropriate point.

To compile C++ code, we use the C++ frontend to the GCC compiler suite by typing `g++`.

While we will rarely use the compiler directly (see section on IDE below), it is worth being familiar with what is happening "behind the scenes". These commands will most commonly be

executed either by a Makefile (more detail later), or by the IDE. To compile a file in the simplest case, simply call `g++` and tell it what file to compile:

```
% g++ -std=c++11 hello.cpp
```

This will (if successful) produce an executable called `a.out` from the source file `hello.cpp`. Generally when compiling for this course, we will add several other options as listed below:

```
% g++ -std=c++11 -g –Wall hello.cpp –o hello
```

-g	Generate debug symbol information (see separate handout about the debugger)
-Wall	Show all (W)arnings – more than the default, may result in lots of compiler output, but will often highlight potential errors
-o hello	Save output as hello instead of a.out

We can of course specify more source files (just add after `hello.cpp`), if more than one is required for the program.

4.4 Debugger

A *debugger* is a tool that helps you find errors (i.e., "bugs") in your programs. It does so by allowing you to control the execution of a program. You can make the program stop when it reaches a particular point in the code: this is called a *breakpoint*. While at a breakpoint, you can examine the values of variables and inspect the state of the program. You can *resume* execution from the breakpoint, possibly to another breakpoint, or step through one statement at a time in your program. You can also set *watches* to view the values of variables you are interested in.

The use of a debugger can save you hours of frustration. While a debugger can never tell you what a bug is, it can help you quickly determine where the bug is located in your code. Combined with your knowledge of what the program should be doing, you can quickly locate and fix your bug.

Learning to use a debugger does take some time. However, this is time well-spent. It certainly beats debugging your code by staring at it for hours at a time, by randomly changing lines in the code, or by inserting print statements all over the code.

For this course, we use the very powerful and extremely common GNU Debugger (GDB). It is free, open-source, and available for many different computer platforms. The debugger is an important tool that is covered in greater detail in a separate document. It is also used by the NetBeans IDE.

4.5 Integrated Development Environment (IDE)

An *Integrated Development Environment* (IDE) is a very powerful suite of tools meant to enhance the developer's productivity. As a minimum, an IDE will typically integrate a text editor with syntax highlighting, tools for browsing source code, compiler/build system, and a debugger. It will often also include collaboration tools such as version control. As the name implies, these features are integrated so that the programmer does not need to switch programs – it can all be done from a single "dashboard" with a consistent look and feel. By combining

many functions into one, it makes switching between components (eg. moving from a compiler error to the relevant source line in the editor, or using the editor to set a debugger breakpoint) very easy.

Significant examples include Eclipse (free, open-source, cross-platform), XCode (freely available for Apple computers, tablets, and phones), and Microsoft Visual C++ (licensing options vary).

For ECE244, we recommend and support a free IDE called NetBeans 8.2. It provides syntax highlighting, source browsing, extensive help features, integration with the GCC compiler, and integration with the GDB debugger. The provided interface is very visual and intuitive, making it easy and worthwhile to learn.

There is a separate tutorial on getting, installing, and starting with NetBeans. We strongly recommend that you use NetBeans unless you have a strong preference for another environment.

5. Preparing, Compiling, and Submitting an Assignment

5.1 Directory preparation

Use your login and password to log onto the ECF system.

In your home directory, create a new directory for ECE244 (this course!) by typing the following at the prompt:

```
% mkdir ece244
```

Make sure that the directory is protected by typing:

```
% chmod go-rwx ece244
```

It is your responsibility to ensure that others cannot copy your work. If another student is able to copy your work, and we determine that two submitted solutions stem from the same source, then we have no option but to take action against both parties, since it is impossible to tell who copied from whom.

Now use the `ls` command to see the new "`ece244`" directory in your list of files.

Change your working directory to `ece244` using the following command:

```
cd ece244
```

If you type the `ls` commend now, you will get no output, since the directory is currently empty; i.e., has no files in it.

It is a good idea to have a separate folder for each course, and for each course folder to have a separate sub-folder for each lab or assignment.

For instance, your first lab files might be stored in `~/ece244/lab1`

5.2 Compiling

Note: You may also choose to use NetBeans (or another IDE) to create and compile your code. In that case, you may safely skip the compiling section here.

You will use a simple example program whose only purpose is to print the message "`Hello, world!`" to the screen. An initial version of this program is provided; simply download it into your working directory (`ece244/examples/helloworld`) from the "Examples" folder in the "Contents" section of the course's web site.

Use the `ls` command to ensure that the file exists

Compile the program by executing the command:

```
% g++ -std=c++11 -g –Wall hello.cpp –o hello
```

This will create an executable called `hello`. You can run it with by typing:

```
% ./hello
```

which runs the executable called `hello` in the working directory.

5.3 Testing the "`Hello, World!`" Program

The automarker program will test all code that you submit. It does so by compiling your programs and then running them with various inputs, or *test cases*. Each time it runs your program it compares the output to that of the reference solution. If your output matches the reference output then your program passes the test case, otherwise it fails. To assist you in testing your program, we make ***some*** of the test cases the automarker will use available to you. To check if you program works correctly on these test cases, you use the **exercise** command.

If your output does not match the expected output `exercise` will tell you. The `exercise` command has the following syntax:

```
~ece244i/public/exercise assignment# program_name
```

Let's assume that our example program should be named `hello`, but, we mistyped our compilation line as:

```
% g++ -g –Wall hello.cpp –o Hello
```

This will create a output file called `Hello` (note the uppercase "`H`") when you compile it. If you ran exercise with this file name, exercise will complain:

```
% g++ -g –Wall hello.cpp -o Hello
% ~ece244i/public/exercise 1 ./Hello

Error - exercise only supports the following executables: hello
Quitting!
```

If `exercise` is run with an executable that does not match the name expected by the automarker, it will print a message like the one above (note that the error messages may be slightly different from the ones printed in this document). To fix the problem, simply specify the correct name when you compile the program, or you can ***copy*** `hello.cpp` into another file `Hello.cpp` using the `cp` command, and re-compile:

```
% g++ -g –Wall Hello.cpp -o Hello
```

Now run exercise again:

```
% ~ece244i/public/exercise 1 ./hello

################################################################
################################################################
Running Testcase 1
################################################################
################################################################

Running the following input on your program:
================================================================
----------------------------------------------------------------

Your program produced the following output:
================================================================
Hello
world!

----------------------------------------------------------------
<<<<<<<<< Comparing output to solution version 1 >>>>>>>>>>>>
Output does not match solution version 1

Output did not match any of the solution versions
The following 1 output(s) would have been accepted:

Solution version 1 output:
================================================================
Hello, world!

----------------------------------------------------------------
Running diff on program output and solution version 1 output
( < actual output vs > expected output )
================================================================
1,2c1
< Hello
< world!
---
> Hello, world!
----------------------------------------------------------------

############### Testcase 1 Results ############################
Functionality: FAIL - Output did not match any solution version

################################################################
################## Summary for 1 test(s) #######################
################################################################

Functionality: FAIL: 1 test(s) failed!

################################################################
################################################################
################################################################
```

Note that `exercise` no longer prints an error message regarding the file name. It does, however, print a different error message. This is an example of the second major pitfall of the automarker: typos. Any difference from the expected output, no matter how minor, will cause your program to fail a test case. Here the program had a line break instead of a comma in its output.

Now fix the problem in the example program. Edit `Hello.cpp` and add the missing comma on line 5. The result should look like this:

```
      cout << "Hello, world!\n";
```

Now recompile and run `exercise` one more time:

```
% g++ -g –Wall hello.cpp -o hello
% ~ece244i/public/exercise 1 ./hello

################################################################
################################################################
Running Testcase 1
################################################################
################################################################

Running the following input on your program:
================================================================
----------------------------------------------------------------

Your program produced the following output:
================================================================
Hello, world!

----------------------------------------------------------------

<<<<<<<<< Comparing output to solution version 1 >>>>>>>>>>>>

############### Testcase 1 Results ############################
Functionality: PASS - Output matches solution version 1

################################################################
################## Summary for 1 test(s) #######################
################################################################

      Functionality: PASS: All tests passed

################################################################
################################################################
################################################################
```

This time `exercise` tells you that the output was correct. You should use `exercise` for every assignment, but you should also remember that passing all of its test cases does not guarantee a perfect mark – the automarker uses some test cases that are not available to `exercise`. You should always perform additional testing.

5.4 Submitting the Work

When you have completed and tested your program, you should submit it for marking using the `submit` command. The `submit` command takes only one argument: the lab number.

```
~ece244i/public/submit assignment#
```

For example, to submit the files needed for your lab 1 to be marked, you would first make sure all the necessary files are in your current directory (which for lab 1 are the `hello.cpp` file and another file called `convert.cpp`). Then type:

```
~ece244i/public/submit 1
```

This command will search for all the files needed for the assignment in the current directory, copy them to the course automarker directory, compile them, and run the `exercise` program on them. If anything goes wrong in this process, `submit` will print an error message. You should carefully examine the error message, take whatever corrective action is appropriate and submit again. For example, if you had the `hello.cpp` program in the current directory, but did not have convert.cpp in the directory, `submit` will complain:

```
% ~ece244i/public/submit 1

Checking for the expected files in the current directory
hello.cpp convert.cpp
can't find file 'convert.cpp'.
Error: Can't find all specified files

No files were submitted
```

It is very important to read the output from `submit` very carefully and ensure there are no errors. If `submit` cannot find the appropriate files or they do not compile correctly, the automarker will not be able to run your program and you will receive 0 on the lab.

If we copy the `convert.cpp` file into the current directory and run `submit` again, `submit` will succeed and produce output like this:

```
% ~ece244i/public/submit 1

Checking for the expected files in the current directory
hello.cpp convert.cpp
The provided list of files is OK.

Attempting to build your code. Make output follows:
================================================================
== g++ hello.cpp -o hello
g++ convert.cpp -o convert
----------------------------------------------------------------
--
The build appears to have been successful...

Running exercise on hello...
############### Testcase 1 Results ##########################
Functionality: PASS - Output matches solution version 1
```

```
################## Summary for 1 test(s) ####################
Functionality: PASS: All tests passed

hello passed exercise
Running exercise on convert...
############### Testcase 2 Results ##########################
Functionality: PASS - Output matches solution version 1

############### Testcase 3 Results ##########################
Functionality: PASS - Output matches solution version 1
################## Summary for 2 test(s) ####################
Functionality: PASS: All tests passed

convert passed exercise

Submitting files

Verifying submission
submitece244f -l 1
total 16
-rw-r----- 1 submit ece244f 2748 Sep 7 10:38 convert.cpp
-rw-r----- 1 submit ece244f 173 Sep 7 10:38 hello.cpp

All filesizes match. All timestamps match submission time.
Submission was successful
```

Reading through the output above you can see that `submit` found all the necessary files, compiled them successfully, and the resulting programs passed the `exercise` test cases. The last line states that submission was successful; you should always check to make sure `submit` prints this success message.

You can submit a lab as often as you like; the automarker will always mark only the last submission.