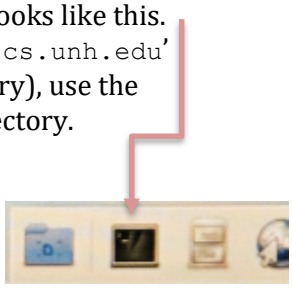
CS515 – Lab 1

The purpose of this lab is to help familiarize you with our lab environment and to get started programming in C++. You will be working with our CS server **`agate.cs.unh.edu`** using the GNU g++ compiler and makefiles. This lab will be graded, but will not count as heavily as other labs.

We allow and encourage *pair programming* for CS 515 labs. Find a lab partner to work together with on your labs (can change over time). Pure pair programming has the pair share **one** terminal and **one** keyboard, making sure both of you take turns to be the driver on the keyboard within the lab period. This can be done remotely using screen-sharing on Discord or Zoom, if you like. It may be a good strategy to switch for each lab task, although many just work "side-by-side". You may share your lab code but each student has to submit his or her own submission.

Download all starter files from `~cs515/public/1L` to your home directory on agate. To do this, log in to your account on agate (`agate.cs.unh.edu`) using a terminal window (or Command Prompt if you're on a Windows machine). If you're in Kingsbury N218, the terminal window icon looks like this. Open the terminal window and then within that type '`ssh <yourUNHloginID>@agate.cs.unh.edu`' (without the quotes) to log into agate. Next (you should already be in your home directory), use the following commands to create some directories and copy the starter files to your 1L directory.



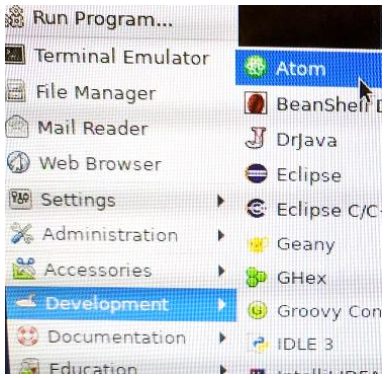
```
mkdir cs515              ← this creates/makes a directory called 'cs515'
cd cs515                 ← makes cs515 the current directory
mkdir 1L                 ← creates a directory called '1L' inside cs515
cd 1L                    ← makes cs515/1L the current directory
cp ~cs515/public/1L/* .  ← copies everything from the public directory into this one (note
                           that the dot is important, and separate from the rest—it
                           indicates "this directory")
```

If you don't yet have an account on agate, please email CS support staff immediately at support@cs.unh.edu and CC me so that I am aware of the situation.

Task 1. Edit C++ programs.

C++ programs usually have the extension ".cpp". You may choose any name for the file (they don't need to match the class names like they do in Java), although it is good practice to use the name of a major class within the file. You may use any text editor you choose to edit your programs. For working at home, many choose to download and use CLion or Visual Studio Code—they can take a little extra effort to learn how to use than a simple text editor like Scite or Atom, but they are more useful/powerful in the long run (beyond this course).

If you're in Kingsbury N218: We recommend you use Atom or **`scite`**. To start Atom, choose Developer->Atom from the start menu in the corner of the screen.
To start the Scite editor, first open a local terminal window and then at the command line shell prompt type in




```
$ scite &   ← '$' is the current shell prompt
            ← add & at the end of the command so it runs at
              background
```

Everyone:

TO DO: Edit the starter code `hello.cpp` and add some funny text to be displayed. [1]

---

[1] When working remotely, you'll use an editor on your own machine and then transfer files back and forth (unless you learn to use a text-based editor like `vi`). TextWrangler is an option on Mac's, since it can edit files directly through SFTP. Some prefer full IDE's like Eclipse or Code::Blocks. Agate is not designed to support serving many graphical applications like Atom or `scite` directly.

Task 2. Compile C++ programs at the command line

If you're working from home: Once you save your changes to your code file, you'll need to transfer the file up to the `agate` server. You can do this with a graphical program like WinSCP (for Windows), but you may want to learn how to use the `scp` command from ***another*** terminal window (works on all platforms). This is how to use the `scp` command:

```
scp hello.cpp <yourUNHloginID>@agate.cs.unh.edu:cs515/1L/
```

Everyone:

Go back to the command line window—the first terminal window where you used `ssh` to connect to `agate`. (If you need to make a new one, remember that after connecting using `ssh`, you need to move to the working directory containing the C++ source files to be compiled. Use `cd` and `ls` command to do this).

The base command for the Gnu C++ compiler is **`g++`**

To compile a program that is in a single file, the easiest compilation uses the command format:

```
g++ <filename>
```

Example:

```
$ g++ hello.cpp
```
← *again, '$' represents the shell prompt (don't type it)*

This command will create an executable program called `a.out`, which is the default name for the executable file.

You may also specify the name of the output file during compilation by using the `-o` option.

```
$ g++ hello.cpp -o hello
```
← *here we use the –o option to rename the default a.out to hello*

The default executable `a.out` is now renamed to `hello`.

TO DO: Compile the `hello.cpp` file to an executable named hello.

Task 3. Execute C++ programs at the command line

If the compilation is successful, the executable can be executed directly at the shell prompt.

```
$ ./a.out
```
← *run a.out*

```
$ ./hello
```
← *run hello*

The "`./`" indicates the executable program to be run is from the current directory.

Note: C++ compilation produces native machine executables so they can run directly at the shell prompt. (Remember that Java compiles into byte code and needs to be executed on a Java Virtual Machine. E.g. `$java HelloWorld`)

TO DO: run `hello` executable at the command line.

More on the next page…

Task 4. Use a makefile to simplify compilation (20 points)

Manual compilation is only feasible for simple programs. When you work on programs that involve multiple files, using a `makefile` can greatly simplify the compilation process. As a first example, the provided makefile is used to compile the program named `hello.cpp` into an executable named `hello`. You only need to type at the shell prompt:

```
$ make
```

The `make` program will look for a file `makefile` in the current directory and compile the program indicated by the first *target* (basically a rule) in that file. The first target is usually "all", which invokes additional targets listed after the colon. Each target consists of 3 main parts

- The target name, followed by a colon. For the starting makefile, `all`, `hello`, and `clean` are all names of targets.
- Space-delimited list of dependencies. These are the names of files that must exist in order for the target to be able to trigger. If the file doesn't exist, it looks for a target by the same name and triggers it, if present.
- Commands to execute when the target is triggered (each line must start with a *tab* character, not just spaces). Note that some commands include variables (macros) that were defined earlier in the makefile.

`make` is an important tool that is used not only to compile C/C++ programs, but pretty much anything that needs to be compiled, E.g. postscript, Java, Fortran. Here are some links to help you get familiarized with makefiles.

http://www.cprogramming.com/tutorial/makefiles.html

https://www.cs.bu.edu/teaching/cpp/writing-makefiles/

http://www.tutorialspoint.com/makefile/why_makefile.htm

Now modify the `makefile` so when you type `make`, it will also compile the given starter code file `addUp.cpp` to an executable named `addUp`.

TO DO: create a new target named `addUp` in the `makefile`. This new target should compile `addUp.cpp` and will also name the executable "addUp". Add the `addUp` target to the list of dependencies on the `all` target (toward the top of the `makefile`).

More on the next page…

Task 5. Execute a console-based program using shell redirection. (20 points)

The `addUp.cpp` program will prompt the user to input 100 numbers and produce the sum of them. Instead of inputting 100 numbers from the keyboard (which is time-consuming and error-prone), you may store all the numbers in a text file and then use the shell redirection feature to look for keyboard input from the file. When using redirection, `stdin` (`cin`) is actually reading from the file instead of the keyboard. File redirection is particularly convenient when we want to test programs that take keyboard input since you don't need to type in data at every program run.

```
$ ./addUp < numbers0
```

Here, the '<' is used to redirect the file `numbers0` to the stdin of the program. It's like an "arrow" pointing from the input file back into the program. The program will now read the numbers from the file as if they were typed at the keyboard.

Note: when we work with file redirections, we are using features from the operating system's shell program (such as bash)—we are **not** modifying the actual program functionality. That is, the `addUp` program still has the same functionality, where it reads from `stdin` (`cin`) and prints output to `stdout` (`cout`).

When running the program with input redirected, you won't be prompted to type in any inputs. This is because the keyboard input is now "connected" to the file. Note that the input does not appear on the screen when you run the program.

When the calculation is finished, the output of the program is still printed to standard output (screen). Can you try to figure out how to redirect the output of the `addUp` program to be saved into a file named `results`, without modifying the `addUp.cpp` program? (*Hint*: you'll still start with what you had, but now you'll add a new "arrow" that points to the output filename, using '>'. Feel free to ask your favorite search engine about "file redirection", too!)

TO DO: Run the program addUp with input and output file redirection using a new input file of your own called `myNumbers`. Add the command that works to your makefile. This should involve adding a `results.txt` target that depends on "addUp", as well as adding the new `results.txt` target to the list of dependencies for `all`. Remember to try deleting your results file and run `make`, to be sure it is actually creating the `results.txt` file.

Submission:

For this lab, you need to submit the `makefile` from tasks 4 & 5, your new input file (`myNumbers`), and the completed `README` file. Submit the files to Lab 1L in the CS 515 course on Mimir at the end of the lab section (this is the only lab where you don't have almost a week to complete it). Here is a list of files you are expected to submit:

`makefile  myNumbers  README`

To get to the CS 515 Mimir course for the first time, go to the Modules page in Canvas (http://mycourses.unh.edu) and click on the link in Week 1 entitled "Mimir One-Click Login App" (it's near the link for this lab). It should guide you right to where you can choose which "project" you want to work with. Note that when you submit, tests will be run and you can get immediate feedback as to whether or not everything is working as it should. If you need help interpreting the feedback it gives, please ask for help from a TA or from your instructor.

You can resubmit as needed, and your last submission will be graded.