Systems Programming Technical Document

Updated for Winter 2023

1 Introduction

Welcome to the systems programming labs! In this document, we will cover a variety of topics that should help you become proficient with Unix, the C programming language, and beyond.

The information detailed within the margins in this hallowed document should serve as a foundation for understanding what is expected of you as a student for the remainder of this semester. Reading this document is mandatory, and doing so should provide you with a glimpse into some history and operation of Unix, maintaining repositories with Git, and design principles of contract programming. You are not expected to know everything just from this document immediately, but having a basic understanding of the key principles of this document is expected before you begin the various programming challenges introduced throughout the semester.

In order for any student to receive practicum marks in their course, they are required to complete the *Submitting Your Lab Agreement* section at the end of this document. By following those directions and submitting your signature file, you are agreeing to follow the terms laid out in the *Lab Agreement* section below.

---

2 Requirements

In order to participate in your class, you will need to know the following information:

1. **Your CSID**: Your Computer Science ID is different from your regular Dalhousie login credentials. If you want to log in to any Computer Science-provided services, you will need your CSID and password. If you don't know what those are, you can visit the CSID website here and retrieve that information. If you have any trouble, don't e-mail the instructor or lab staff. Instead, send an e-mail to cshelp@cs.dal.ca and they will get back to you as soon as possible.

2. **An SSH client**: SSH is a protocol for connecting to a remote computer in order to use its system as if you were physically present. If you use Windows 10 or later, the **ssh** command is built into both Command Prompt and PowerShell. If you use Mac or any flavour of Linux, you're nearly guaranteed to have access to the **ssh** command via the Terminal program. If you are using older versions of Windows (XP, 7, 8), you will not have a built-in option for SSH. The quickest solution is to download PuTTY and use our directions in the *Connecting to Timberlea* section with the PuTTY GUI instead of command line. It will open a terminal for you when your credentials are correct. Anyone can use PuTTY, as it is a viable and portable option. An alternative solution is to simply upgrade to Windows 10 or later, you hooligans. The Computer Science faculty gives every student free copies of the Windows operating systems. If you want to know more about upgrading, contact the CS Help Desk at the e-mail above.

---

3 Lab Agreement

By submitting your signature file via the process outlined in the *Submitting Your Lab Agreement* section, you agree to the following:

1. You agree to adhere to the *Lab Submission Requirements* and the individual lab instructions when submitting code for this course. Practicums follow these same requirements, so understanding them is important.

2. You have read and understood the schedule of labs and practicums as outlined in the Important Dates document, packaged with this file.

3. You agree to abide by the contract requirements of the *Design by Contract* section, and the individual requirements of each set of lab instructions, and understand that failure to do so could result in loss of marks during practicums.

4. You agree to read and abide by all Academic Integrity Policies (link) as laid out by Dalhousie University.

5. You agree that you otherwise have read the entire contents of this document.

Submitting your lab agreement by the end of the semester is mandatory. Failure to do so will automatically set all of your practicum grades to 0 at the end of the semester.

---

4 Lab Submission Requirements and Solutions

In this course, labs must be submitted according to a very specific set of rules on content and directory structure. Each set of lab instructions will require you to submit a specific set of files, arranged in specific directory structure, where your files all have specific names. Failure to follow these requirements will result in your submission failing in the testing scripts.

To help you ensure your submission is properly structured (and has the right contents, depending on the lab; see individual lab submission requirements in each set of lab instructions), we have set up a continuing integration (CI) pipeline script for every lab, including this agreement. The **.gitlab-ci.yml** file in the root of each repository contains the CI/CD scripts used for compiling and testing your solutions over the course of the semester, and the **CI** directory contains any extra tools or scripts required by the CI/CD pipeline. Every time you update a lab repository using **git push**, GitLab will automatically run the pipeline script to check your code. By default, the initial push made by the instructors to your repository will alert you with an e-mail saying the pipeline is broken. If you push new code/files and the pipeline job passes, you will receive an e-mail saying your pipeline has been fixed.

If the pipeline is fixed and your pipeline job was successful, your current push is a candidate for a complete program. Congratulations! Now is the time to review your pipeline's job outputs to ensure they're producing valid outputs. Just by human nature, the testing scripts are guaranteed to not be perfect (although we strive to be as close as possible), so checking your outputs will let you confirm that your code is working, or perhaps you have discovered a bug in the testing process. **Always check your outputs before considering your lab completed**.

If you would like to see the outcome of any pipeline, you can click the pipeline number in the e-mails you receive. If you want to access the pipeline interface directly, you can do so as follows:

1. Visit the GitLab group for either CSCI 1120 or CSCI 2122 and sign in with your CSID. If you don't know your CSID, you can find it here. If the site is not immediately available, we will announce when the Computer Science IT team creates your course group for the new term.
2. Once you have signed in, you should see a list of all the labs currently assigned to the group. Those labs are sub-groups and will contain repositories with names that match your CSID. Click on your desired repository link to enter that lab's repository page.
3. Once you're on the repository page, clicking **CI/CD** on the left hand side of the screen will take you to the pipeline menu.
4. To show the outcome of a job, find the job you'd like to the pipeline log for and click it's status box, which normally says passed (green), failed (red), or running (blue).
5. On the pipeline screen, you can get a variety of information, with the ability to click individual scripts below in the Pipeline tab to see their output.
6. The output page shows each command that was executed by the pipeline script in order. These lines will be green. The resulting output for each of those commands will be shown in white. Check the log to see at which point in the pipeline script your code failed. You can use that information to search for problems in your code. If the last line of the job says "Job Successful", then you likely do not have any problems, but it's good to still read the output to confirm success.

4.1 Solutions

Solutions for each lab will be provided in the course's GitLab group in a repository named **Solutions**. These will be updated the day after the due date (for students in CSCI 1120) or the day after the recommended completion date (for students in CSCI 2122). Any solutions which create libraries or data structures may be provided during practicums as C object files (.o) for use by students, as necessary. You can find a list of which object files will be provided in each practicum's overview, which will be posted to Brightspace before the practicum dates.

5 Design by Contract

Throughout this course, you will be expected to adhere to the principles of Design by Contract (DbC). Design by Contract is a paradigm of software development which creates conditions and structural requirements on code such that the use of the code is divided into two perspectives: the **client** and the **provider**.

The **client** is the point of view of the person *using* your code (not running it). They are given a set of requirements (called a client's **pre-condition**) for using your code, and if they follow those requirements they should receive a guaranteed result (called a client's **post-condition**). On the other side of the system is the **provider**, who receives some explicit or implicit benefit to the client following the rules of the pre-condition, and these benefits allow the provider to guarantee the client's post-condition in the simplest means possible.

For example, on Halloween in Canada there is a cultural agreement between children (the client) and homeowners (the providers) that if the children dress up in costumes when they visit homes (the pre-condition), then the homeowners will give them candy to eat (the post-condition). The benefit to the client is that as long as they adhere to the costume rule, they will receive free candy. The benefit to the provider is that they don't have to provide costumes to children, don't have to leave their house to provide the candy to children, and checking if children on their doorstep are wearing costumes is very simple.

For visualization purposes, these types of contracts can be broken down into simple tables:

	Client	**Provider**
Requirement	Client Pre-Condition	Provider Post-Condition
Benefit	Client Post-Condition	Provider Pre-Condition

where the Halloween example above can be laid out as:

	Client (Children)	**Provider** (Homeowners)
Requirement	Be a child. Dress up in a costume on Halloween. Go to people's houses.	Provide candy to children dressed in costumes.
Benefit	Receive free candy.	Don't have to provide costumes for children. Able to supply children with candy without having to leave their house.

When writing code, especially in the later parts of the course, you will often be given a contract by which your C functions must be designed. As a student, you are considered to be the provider and the instructions provide you with the client's necessary conditions for proper execution. You can think of the testing and marking files as the client, which means you must adhere to their requirements as per the contact in order to receive full marks. The contract descriptions will include:

1. **Pre-Condition**: The client's pre-conditions will include function parameter types, function parameter order (function signature), function names, and the formatting of any data the client will provide those functions. The pre-condition can also dictate specific files in which code must be contained, which could include the names and numbers of files required.
2. **Post-Condition**: The client's post-condition is the expectation of a specifically formatted result of the correct type. This means that any functions you write will need to output exactly as shown. Data types, data format, and number of outputs are all to be specified in the post-condition requirements for a function.

Note that the function contracts specifically dictate the structure of your functions from the client's point of view, but how you fulfill the requirements with your function code is up to you. Ensure you structure your code exactly, and provide appropriate header files, as it will be checked during the marking process. Also ensure your function outputs are generated **exactly as shown** in any instructions, as the first-pass marking programs we use to generate all of your code's necessary outputs will produce output errors if your format isn't exact. Be sure to make good use of the code testing programs via the CI pipeline when you push your work to the course repositories.

6 Basic Unix Expectations

In this section we will discuss how Unix commands are structured, how you enter them to the terminal, and how some introductory Unix commands are used.

Unix commands are made up of three components: the **command** itself, **options** (sometimes called flags), and **arguments** (sometimes called parameters). Options allow you to modify the way a command executes. Options are denoted with a **-**, followed by a letter, or sometimes a series of letters if you are entering multiple options. More advanced options (such as those provided by your C compiler) may be whole words. Some commands require multiple arguments (such as multiple file names), while some commands don't require any arguments at all.

Generic Unix command: **command -options argument1 argument2**

The Faculty of Computer Science has several servers available for both undergraduate and graduate students. In this class, you will be using the Timberlea undegraduate server. This server runs on the Debian Linux distribution and uses the XFS file system by default.

Like many Unix file systems, Timberlea's system is based on having user account (home) directories situated in a root file system, where users are able to execute commands in a sequential fashion to manage their tasks. Douglas McIlroy, who proposed the system for handling Unix pipelines (which you will learn about in Lab 1), wrote a 6-page technical paper about the Unix Time-Sharing System in 1978 (link) which described the main design principles of Unix as follows (paraphrased and interpreted):

1. Make each program do one thing well.
2. Expect the output of every program to become the input of another.
3. Test early, test often. Rebuild things which are written clumsily.
4. Building a program to do a job is better than getting help from someone who can't.
5. If you build something, don't be afraid to discard it in favor of something better.
6. Small is beautiful.

Philosophy of code design aside, the modern Unix operating system is still based on those principles, as you'll find below and in labs to come. Most of the base Unix commands you find in the operating system are small, independent programs with simple functionality, but having them interact in sequence means that your small, independent programs can become larger, more complex, and robust programs with very little effort. Read below to get started with understanding some Unix commands and expect to become familiar with command sequencing (pipelining) in future labs.

6.1 Directory Management

When working within the Unix file system, your tasks will inevitably lead to the creation and management of **directories** (also known as folders in other operating systems). Directories let you organize your data into file sets so that you have a better logical flow in your file system, similar to a filing cabinet. They are a special type of file (hotly debated!) which contain references to other files, while also being allowed to reference other directories. This means you can organize your files and directories into other directories, which eventually leads to a full-fledged file system. Below you will find some commands useful for navigating and creating your directory structures.

6.1.1 Making Directories

mkdir is used to create a directory, (short for make directory). To use it, type **mkdir** followed by the name of the directory you would like to create. You can also use it to make multiple directories.

```
mkdir example
mkdir folder1 folder2 folder3
mkdir a a/longer a/longer/path
```

6.1.2 Removing Directories

rmdir is used to remove a directory. To use it, type **rmdir** followed by the name of the directory you would like to delete. You may notice that if the directory is not empty, **rmdir** will fail. You can overcome this limitation with the **rm** command below.

```
rmdir example
```

6.1.3 Changing Directories

cd is short for change directory and it is used to navigate throughout your file system. To use it, type **cd** followed by the name of the directory you would like to enter. You can also enter a longer path. To navigate up one level (to the directory which contains the directory you're currently in), use the **cd ..** command.

To navigate to your home directory, use **cd** with no arguments, or **cd ~/** instead.

```
bayer@timberlea:~$ cd MyDirectory
bayer@timberlea:~/MyDirectory$ cd AnotherDirectory
bayer@timberlea:~/MyDirectory/AnotherDirectory$ cd ..
bayer@timberlea:~/MyDirectory$
```

```
bayer@timberlea:~$ cd MyDirectory/AnotherDirectory
bayer@timberlea:~/MyDirectory/AnotherDirectory$ cd
bayer@timberlea:~$
```

6.1.4 Listing a Directory's Contents

ls is used to list the contents in a directory. **ls** has MANY options, so be sure to check out **man ls** to see all of the options.

```
bayer@timberlea:~/MyDirectory$ ls
file1.txt  file2.txt  file3.txt
```

The **-a** option, short for "all", will list **every** file in the directory, including any file that starts with **.**, as those files are omitted by **ls**.

```
bayer@timberlea:~/MyDirectory$ ls -a
.  ..  file1.txt  file2.txt  file3.txt
```

The **-l** option, short for "long list", will show all of the files as a list, along with details about file permissions, the number of hard links in the directory, (1 if it is a file), the owner of the file, the group the owner belongs to, the date the file was last modified, and the name of the file.

```
bayer@timberlea:~/MyDirectory$ ls -l
total 2
-rw-rw-rw-. 1 bayer csgrad 0 Nov 15  2000 file1.txt
-rw-rw-rw-. 1 bayer csgrad 0 Sep  4 12:05 file2.txt
-rw-rw-rw-. 1 bayer csgrad 0 Sep  4 12:15 file3.txt
```

Combining **-l** and **-a** into **-la** will show a list of **every** file in the directory, with the details listed above.

```
bayer@timberlea:~/MyDirectory$ ls -la
total 11
drwxrwxrwx.  2 bayer csgrad  5 Sep  4 12:05 .
drwx------. 13 bayer csgrad 20 Sep  4 01:45 ..
-rw-rw-rw-.  1 bayer csgrad  0 Nov 15  2000 file1.txt
-rw-rw-rw-.  1 bayer csgrad  0 Sep  4 12:05 file2.txt
-rw-rw-rw-.  1 bayer csgrad  0 Sep  4 12:15 file3.txt
```

6.2 File Management

In this section, we will cover some basic functionality for dealing with files (and, as implied above, these commands can also work on directories). Simple operations include copying, moving, creating, and renaming.

6.2.1 Simple File Creation and Timestamp Modification

touch is used to update the time that a file was last accessed, or last modified. If you enter a file that does not exist, it will create an empty file with that name.

To create an empty file, or multiple empty files, use **touch**, followed by the name of the file(s) you want to create.

```
bayer@timberlea:~/MyDirectory$ ls
bayer@timberlea:~/MyDirectory$ touch file1.txt
bayer@timberlea:~/MyDirectory$ ls
file1.txt
bayer@timberlea:~/MyDirectory$ touch file2.txt file3.txt
bayer@timberlea:~/MyDirectory$ ls
file1.txt  file2.txt  file3.txt
```

To update the modification date and the last accessed date, use **touch** with no options. To update the last accessed date only, use the **-a** option. To update just the modification date, use the **-m** option.

```
touch file1.txt
touch -a file2.txt
touch -m file3.txt
```

To use a time that isn't the current time, use the **-t** option, followed by the desired date. The date is in the format of YYYYMMDDHHMM.ss (year, month, day, hour, minute, second). In the below example, both the access and modification dates are changed to November 15, 2000 at 12:45:20.

```
touch -t 200011151245.20 file1.txt
```

6.2.2 Removing Files or Directories

rm is used to remove (delete) a file. To use it, type **rm** followed by the name of the file you want to remove. You can also use **rm -r** to remove a directory, and everything inside it. **Use the recursive (-r, -R) options with extreme caution, as it will delete everything you tell it to, with no ability to undo!**

```
rm hi.txt
rm -r folder1
```

6.2.3 Moving and Renaming Files or Directories

mv is used to move a file or directory. **mv** requires two arguments: the file or directory you would like to move (known as the source), and the location you would like to move the source to, (known as the destination). To use it, type **mv** followed by the source and destination.

mv can also be used to rename a file. From the directory the file is in, enter **mv** followed by the current name of the file and the new file name.

```
mv file1.txt AnotherDirectory
mv file2.txt file1.txt
```

6.2.4 Copying Files or Directories

cp is used to to create a copy of a file or directory. Similar to **mv**, it requires a source file or directory to copy, and a destination to place the copied file in.

```
bayer@timberlea:~/MyDirectory$ cp file1.txt AnotherDirectory
bayer@timberlea:~/MyDirectory$ ls -R
.:
AnotherDirectory  file1.txt  file3.txt

./AnotherDirectory:
file1.txt
```

6.3 Code Editors

The Timberlea server has several text editors installed, including **nano** (a very simple text editor), and two coding-specific editors named **vim** (intermediate) and **emacs** (advanced). We recommend students use Vim, as it has enough features to be robust, while also being fairly simple when it comes to basic usage. We do not offer support for text editors outside of basic functionality, but there are many sources online for explaining useful vim/emacs commands. In the section below we will explain basic Vim usage.

6.3.1 Vim

To use Vim, type **vim** followed by the name of the file you want to edit. If the file name does not exist, a new file will be created, but will not be stored on the hard drive until you specifically tell Vim to do so.

Vim has two modes: **Command**, and **Insert**. Command mode allows you to enter commands to save your file, quit the program, etc. Insert mode allows you to edit the text in your currently opened file.

To switch from insert mode to Command mode, press the Escape key. To switch from command mode to Insert mode, press **i**. You will know you are in Insert mode if "– INSERT –" appears at the bottom left of the Vim window. If you are in Command mode, you will either see the name of the file or a blank space.

To enter a command to Vim, you must be in Command mode. Commands can come in the form of a hotkey (single button press) or a string command. Typically, string commands require you to start with a colon (:) to let Vim know that you're not trying to press a hotkey. It's possible to enter more than one command in your text string, as you will see with **:q!** and **:wq** below.

Common commands include:

i changes from Command mode to Insert mode.

:w writes (saves) your file to the hard disk.

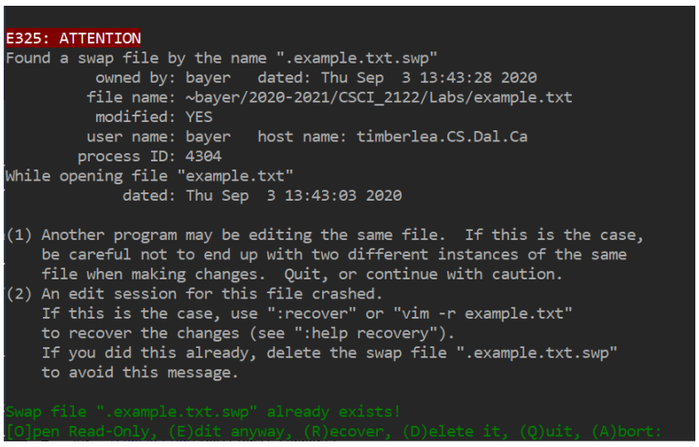
:q will quit Vim if you have saved your changes. If you have not saved your changes, it will throw an error.

:q! will quit Vim **without saving** your changes.

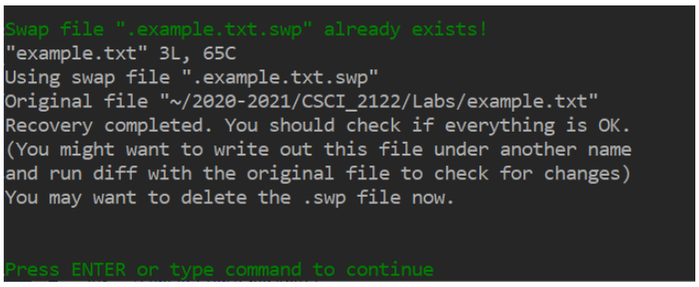
:wq will first write, (save), your file, then exit Vim.

In the event that you accidentally disconnect from the server, Vim automatically saves a recovery file known as a *swap file*. This file (hopefully) contains your unsaved edits to a file. When you open Vim, you will see a screen that

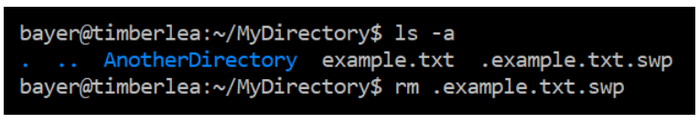
looks like this:



If you see this message, press **R** to load the recovery file. Next, you will see this screen:



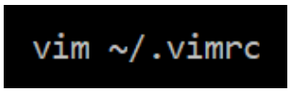
Press enter to continue. Then, press the escape key to ensure you are in command mode, and type **:w** to save your file. **Once you exit Vim, you need to delete the recovery file.** The recovery file ends in **.swp**. This file is only visible by using **ls -a**. Use **rm** to remove the swap file.



6.3.2 Configuring Vim

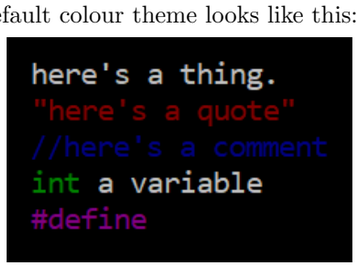
Vim has commands that change how Vim looks. Common changes include setting visible line numbers, and changing the colour theme. To see line numbers, open Vim, press the escape button to ensure you are in command mode, and type **:set number**. The command to hide line numbers is **:set nonumber**.

However, Vim will not remember changes that you make to the environment, and it will load in the default style the next time it is opened. We can create a file containing commands for Vim to run every time Vim starts, called a *vimrc* file. To create or edit the **vimrc** file, you can enter the following command:



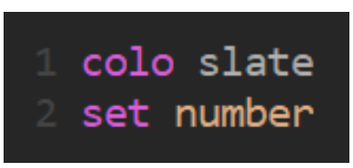
then enter the commands you would like Vim to run at startup without a colon, and save the file. Notice that the file path starts with ~/. This is similar to the **./** directory, except instead of referring to itself, ~/ refers to your home directory, which is the same directory that you start in when you first log in to Timberlea.

Vim has 17 built in color themes. The default colour theme looks like this:



A list of these themes, samples of how they look, and further customization options can be found here.

To change themes, open Vim, and from command mode, enter **:colo** followed by the name of the theme you want to use. For example, to change to the theme to Slate, you would use **:colo slate**. To add this command to your **.vimrc** file, add **colo slate**..



6.4 Support

In this course, you will receive support for the required (basic) functionalities of **gcc** (for compiling C code), **gdb** (for debugging C code), and **vim**. Beyond the basics of Unix and C, presented here and in other labs as required, we will offer very limited support and **will** adhere to the following rules:

1. We expect all code to be written, compiled, and executed on the Timberlea server. We do not offer any support for compilers, IDEs, text editors, or any other software that we have not introduced in this document or during the remainder of the course. We don't offer any guarantee that using other software will work and the responsibility of using any such software lies with you.

2. All code submitted to the GitLab repositories for this course will be compiled and executed on Timberlea. It is your responsibility to test any and all code on Timberlea before submission. We do not provide help with compiling on a system other than Timberlea using **gcc**.

3. When asking questions about Unix and C programming, we expect that you have attempted a variety of methods to solve your issue before reaching out to the instructors in this course. Simple tasks such as manually entering commands (without copy-and-paste), reading **man** pages for Unix commands, and checking online resources are considered mandatory baseline solutions to any issues you might have before asking for help. That said, while the lab material is challenging, **we do not expect you to be hung up on a single problem for many hours**. Set reasonable time limits on trying to solve a problem yourself before seeking extra help from an instructor.

---

7 Connecting to Timberlea

Note that "yourCSID" denotes **your** CSID; don't type in "yourCSID" as shown. If you don't know your CSID, refer to the *Requirements* section at the top of this document to retrieve your CSID and password. **You should also take a moment to change your CSID password if you are still using the default.**

1. First, we are going to open either Command Line (Windows) or Terminal (MacOS, Linux). Alternatively, Windows users can use PuTTY.

   Windows

   (a) Press the Windows key, or click the Windows icon in the bottom left corner of the screen (the default location), to open the Start Menu.
   (b) Type 'cmd', and open Command Prompt from the menu by clicking it or pressing Enter.

   MacOS
   Open the Terminal application. This can be done by entering Launchpad, and typing "terminal", or clicking the magnifying glass in the top right corner, then typing "Terminal".

   Linux
   Most dedicated Linux users likely don't need instruction on opening Terminal. Depending on your distribution, it could be as simple as pressing CTRL+ALT+T, or you may be able to hit the Windows key on your keyboard to open the program search menu. Rather than us explaining every option, a simple Google search for how to open your distribution's default Terminal will provide you with detailed steps.

   Putty (Alternative SSH for Windows)
   Instructions for signing in to Timberlea with Putty can be found here.

2. Next, we are going to use **ssh** command to connect to the Timberlea server. SSH stands for Secure Shell and more information can be found at the official protocol website. In the case of both the **ssh** and PuTTY, in order to connect to a remote server, you will need two pieces of information: a username (your CSID, in this case), and a server address. When signing into any remote server, a simple trick to combine those fields is to separate them with an @ symbol, such that you tell SSH both the username and server address in a single string, formatted as **username@server-address**. SSH always runs on port 22 (as per the protocol standard), so there's no need to set or change that unless we tell you to.

   To log in to Timberlea with a terminal or command prompt, you can type the following command and then press the Enter key:

   ```
   ssh yourCSID@timberlea.cs.dal.ca
   ```

   If your computer asks a yes or no question (about authentication verification), type **yes** and press the Enter key.

   Next, you will be asked for a password, which you can enter now. **When you enter your password, you will not see anything being entered in the password field. This is normal**. Press the Enter key once you've entered your password and you will be logged into your home directory on Timberlea.

   If you receive an error saying your SSH connection is refused, e-mail the CS Help Desk with your CSID and IP address, as it is likely your IP has been blocked from too many failed login attempts.

8 Using Git

Git is version control software that allows programmers to track changes to their code over time. Git is usually hosted on a server, in our case, **git.cs.dal.ca**. A project in Git is known as a repository, or repo for short. When you submit your lab code over the course of this semester, you will be required to do so via git repositories, so it's important that you understand the process to follow in order to have your code pushed to the GitLab servers properly.

If you want to verify which files are currently present in a given Computer Science hosted repository (and get access to other tools, such as the CI/CD pipeline), you can visit the Dalhousie Computer Science GitLab here. You will need to sign in with your CSID and password before you can access any groups or repositories you belong to.

The following sections should provide you with the very basic functionality for using Git, but there are many other commands and features available. You can find information on those online by searching for Git guides and examples.

8.1 Clone

git clone followed by a URL will copy a repository from a Git server into your local directory. The local copy is known as a "local repository". This copy is "tracked", meaning that git keeps track of the differences between your local repository and the remote Git repository. You are able to enter Git commands which will affect the cloned repository if you perform them inside one of the cloned directories.

Cloning a project hosted on the CS Git server will require you to enter your CSID and password to complete the clone.

8.2 Add

git add followed by a directory or filename *stages* modifications made to the given files. When a file is staged, it tells Git which files to update in the next **commit**. **git add .** stages every file in the local repository, but we will provide the usual warning that **mass command execution (such as adding everything) should be done with caution**.

8.3 Commit

Committing is essentially "saving" changes made in your local repository, which lets Git know that you want to put these files in their current state on the Git server. Committing **does not** update the file on the Git server, as updating the Git server requires a **git push** command (see below). Any changes made to a file after committing will not be reflected in Git. If you change a file, you will need to commit again.

A required part of a Git commit is a commit message. This should be a short message that briefly describes the changes made in this commit, so that you (or other users) can understand why the changes needed to be made. Usually this is a short message which reflects small changes. Larger changes should be identified in the message, but the documentation should be updated to reflect those changes instead.

To commit your staged files, type **git commit -m "a commit message."**

8.4 Push

Once files have been added and committed, you have to send the changes you made to the Git server. Type **git push** to push your changes to the Git server.

Pushing to a project hosted on the CS Git server will require you to enter your CSID and password to complete the push.

8.5 Pull

Using **git pull** will update your local repository with the code that is on the Git server. If another developer has pushed their changes to the Git server, pulling will put their changes in your local repository. **If you have made changes to the respository locally and have not pushed them to the server, pulling the contents of the Git server's repository could overwrite your changes. Be careful!**

8.6 Status

git status shows the differences between your local repository and the remote repository. It also tells you if new files have been added in your local repository, if files have been removed from your local repository, and what files have not been staged. It's recommended that you check your Git status before performing any commits or pushes to ensure the upcoming changes are what you're expecting.

8.7 Branches

Branching makes a copy of a repository at the point of a particular commit. The default branch in Git is traditionally called "master", but that naming standard is slowly shifting toward "main" instead. Code in the master branch should be fully functional. When a developer wants to make changes to code in master, they first create a branch, then they can work on that branch without potentially breaking any code in master. Branching is very useful when multiple developers are working on one project, because each developer can work on their own branch.

8.7.1 Checkout

To switch branches, use **git checkout** followed by the name of the branch you want to switch to. To see a list of available branches, use **git branch -a** and to create a new branch from your local repository, use **git branch -b** followed by the name of the new branch.

---

9 Submitting Your Lab Agreement

In this section, we will use the information in this document to create a baseline directory structure for the class, pull your "LabAgreement" Git repository to an appropriate location, add your signature to the repository, then push that repository to the Git server.

Note that in this example, we are using a Windows 10 Command Prompt and accessing the Winter 2022 files for CSCI 2122. These are purely for example purposes and you should fill in the appropriate fields which match your course when you need to clone your repository. For example, if you're a student in CSCI 1120 this semester, then any mention of **2022-winter** should be changed to **2023-winter** and any mention of **csci-2122** should be changed to **csci-1120**. You can also access your repository via your GitLab group and use the URL for your repository when cloning.

9.1 Sign in to Timberlea and Create Directories

To start, we will sign in to Timberlea and create some basic directory structures for managing your lab code throughout the semester. The simplest directory structure is to create a **CSCI1120** or **CSCI2122** directory (based on your course), and inside that create a directory for every lab submission you require. In the future you can use **mkdir** to make any directories you want, but we will start here by creating a **LabAgreement** directory for holding your Lab Agreement repository. Note that the repository also contains a **LabAgreement** directory: don't get confused! You will see them both in the images below.

Follow these steps to gain access to Timberlea and create your directories. Any time you see the instructions entering a CSID (**rsmith** in these instructions), you should be entering your CSID instead.

1. Sign into Timberlea using your CSID.

```
C:\Users\RD>ssh rsmith@timberlea.cs.dal.ca
rsmith@timberlea.cs.dal.ca's password:
********************************************************************************

        Welcome to Dalhousie University!  (timberlea.cs.dal.ca)

This is a SHARED machine! Please be respectful of the other users. If you are
starting large or long running jobs, please consider adjusting the priority
with the 'nice' command.  Grad related jobs should be running on hector!

********************************************************************************

ATTENTION PYTHON USERS: Python 3 is now the default version.
A minimal Python 2 is available at /usr/bin/python2.7 but Python 2 is
officially end of life. Please adjust your code for Python 3.

********************************************************************************

JupyterHub is now available for running Jupyter notebooks. Log in with your
CSID at https://timberlea.cs.dal.ca:8000

********************************************************************************

If the symbolic link to your public_html directory is deleted, you can
recreate it by running or copying and pasting the following command:

ln -s /users/webhome/$USER ~/public_html

********************************************************************************
Last login: Mon Sep  7 12:50:51 2020 from 134.41.44.95
********************************************************************************
Last login: Mon Sep  7 12:50:51 2020 from 134.41.44.95
rsmith@timberlea:~$
```

2. Use the **mkdir** command to create a directory named **CSCI2122**.

```
rsmith@timberlea:~$ mkdir CSCI2122
rsmith@timberlea:~$
```

3. Use the **cd** command to navigate into the **CSCI2122** directory.

```
rsmith@timberlea:~$ cd CSCI2122
rsmith@timberlea:~/CSCI2122$
```

4. Use the **mkdir** command to create a directory named **LabAgreement**, then use **cd** to navigate into it.

```
rsmith@timberlea:~/CSCI2122$ mkdir LabAgreement
rsmith@timberlea:~/CSCI2122$ cd LabAgreement
rsmith@timberlea:~/CSCI2122/LabAgreement$
```

9.2 Use Git to Sign the Lab Agreement

Now that you have a directory for holding your repository, we can use Git commands to retrieve your lab agreement repository, make changes to those files, then push those changes to the Git server so the markers can see them.

1. Use **git clone** to clone your lab agreement repository from the Git server. You will be prompted to provide your CSID and password. Enter them as normal. **Remember to use your own CSID instead of "rsmith"!**

```
rsmith@timberlea:~/CSCI2122/LabAgreement$ git clone https://git.cs.dal.ca/courses/2022-winter/csci-2122/labagreement/rsmith.git
Cloning into 'rsmith'...
Username for 'https://git.cs.dal.ca': rsmith
Password for 'https://rsmith@git.cs.dal.ca':
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 11 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (11/11), done.
rsmith@timberlea:~/CSCI2122/LabAgreement$
```

2. Once the clone process is complete, you can use **ls -al** to view the contents of the directory. You should now have a directory in this directory matching your CSID. Use the **cd** command to navigate into your CSID directory.

```
rsmith@timberlea:~/CSCI2122/LabAgreement$ ls -al
total 10
drwxr-xr-x. 3 rsmith csgrad 3 Sep  7 12:59 .
drwxr-xr-x. 3 rsmith csgrad 3 Sep  7 12:55 ..
drwxr-xr-x. 5 rsmith csgrad 6 Sep  7 12:59 rsmith
rsmith@timberlea:~/CSCI2122/LabAgreement$ cd rsmith
rsmith@timberlea:~/CSCI2122/LabAgreement/rsmith$
```

3. Use the **vim** command to open the **LabAgreement/signature.sig** file.

```
rsmith@timberlea:~/CSCI2122/LabAgreement/rsmith$ vim LabAgreement/signature.sig
```

4. On the first line of the file, press **i** on your keyboard to enter Insert Mode and type your CSID. The file should not contain any other information. Do not enter your password! (Note that the Vim image may look different from yours; it has line numbers turned on, so don't include "1" at the start of the line with your submission. You only need to enter your CSID.)

```
  1 rsmith
~
~
~
~
~
~
~
-- INSERT --           1,7           All
```

5. After you've entered your CSID, press Esc on your keyboard to enter Command Mode, then type **:wq** and press Enter on your keyboard to save the file.

```
  1 rsmith
~
~
~
~
~
~
~
~
:wq
```

6. Use **git add** to tell Git to stage the updated **LabAgreement/signature.sig** file, then use **git status** to ensure the **signature.sig** file is ready to be committed.

```
rsmith@timberlea:~/CSCI2122/LabAgreement/rsmith$ git add LabAgreement/signature.sig
rsmith@timberlea:~/CSCI2122/LabAgreement/rsmith$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   LabAgreement/signature.sig

rsmith@timberlea:~/CSCI2122/LabAgreement/rsmith$
```

7. Use **git commit** with the message **"Agreement has been signed."** to tell Git to commit the changes on the next push.

```
rsmith@timberlea:~/CSCI2122/LabAgreement/rsmith$ git commit -m "Agreement has been signed."
[master a31a729] Agreement has been signed.
 Committer: Robert Smith <rsmith@timberlea.CS.Dal.Ca>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 1 file changed, 1 insertion(+)
rsmith@timberlea:~/CSCI2122/LabAgreement/rsmith$
```

8. Use **git push** to move your signature file to your repository on the Git server. You will need to enter your CSID and password to proceed.

```
rsmith@timberlea:~/CSCI2122/LabAgreement/rsmith$ git push
Username for 'https://git.cs.dal.ca': rsmith
Password for 'https://rsmith@git.cs.dal.ca':
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 32 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 313 bytes | 62.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
To https://git.cs.dal.ca/courses/2022-winter/csci-2122/labagreement/rsmith.git
   05e1a7c..074d69a  master -> master
rsmith@timberlea:~/CSCI2122/LabAgreement/rsmith$
```

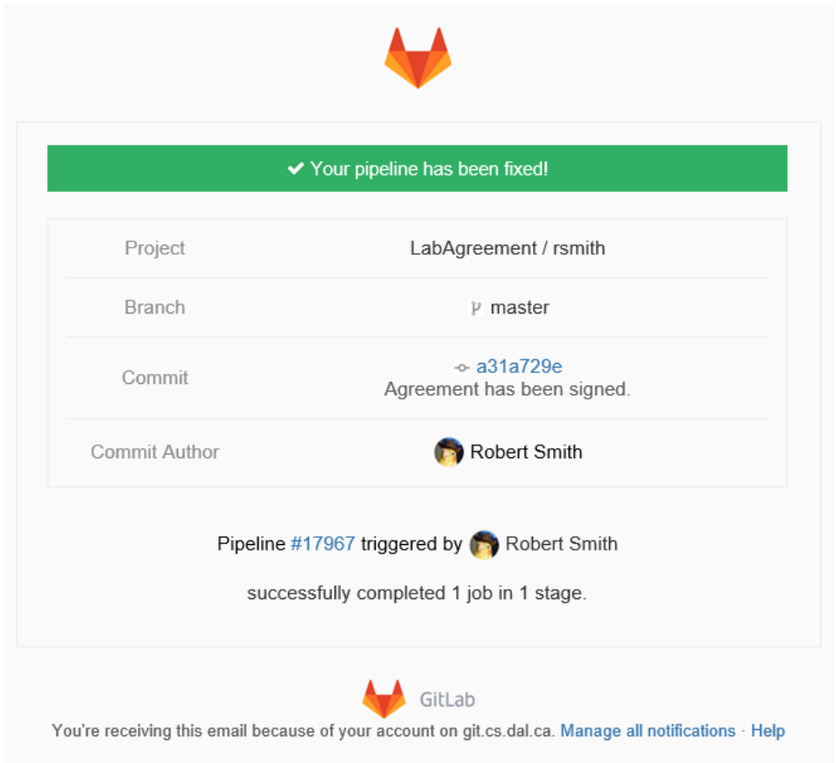
9. Patiently await an email that lets you know if you did it correctly. If you do not receive an e-mail stating that the pipeline is fixed after a few minutes, don't panic! Check the pipeline status on either the CSCI 1120 or CSCI 2122 GitLab group to see why the pipeline failed. It could be one of the following reasons:

   (a) You accidentally deleted the **signature.sig** file.

   (b) You accidentally moved or renamed the **signature.sig** file.

   (c) You did not properly include your CSID in the **signature.sig** file. Make sure the first line of the signature file is your CSID, with no other information in your file.

   Once you read the pipeline output, where you went wrong will likely become obvious. Go back and make any necessary changes and repeat the add/commit/push process to try again.



If you receive an e-mail stating your pipeline has been fixed, congratulations! You have successfully agreed to the lab agreement and we hope you have a great semester with the systems programming labs!