Consolidation session One:
Version control, the build system and linking

February 2019

Log into your Linux account and download exercise.tar.gz from the assignments tab on Vula.

Part One: Setting up your local repository

Last year you should have had some exposure to version control and specifically how to use Git. In this course we expect you to version control every assignment. **If your submission does not contain a git repository a significant penalty will apply.** Let's recap what to do in order to create a local repository.

If this is the first time you're using git on a machine in the CS department you will need to setup some identifying information (this is a once off step). These two commands are the two critical ones to complete. These details will appear in your commit history and will identify your work when working in a team.

```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

Extract the exercise tarball and navigate to the newly created exercise directory using

```
tar -xvf exercise.tar.gz
cd exercise
```

In this folder you should find the following files

```
driver.cpp
factorial_library/factorial.cpp
factorial_library/factorial.h
```

Now that everything is extracted lets initialize a repository. Run the following:

```
git init
>Initialized empty Git repository in \\
   /scratch/CS3022H_repos/csc3022h/minituts/twentyfour/.git/
```

This will create a hidden `.git` folder inside the exercise folder. Make sure that it exists by running `ls -a`. You should see the folder listed. This folder contains the commit history of your project. If you delete it your revision history will be lost and your project no longer contains a git repository. **Take care not to remove this folder**.

Let's now add the given program code to the newly created repository and check the status of what have been modified and staged for commit since we created our repo. After we've ensured that everything that should be in this unit of work has been added we can commit the unit of work.

```
git add driver.cpp factorial_library/
git status
>On branch master

>Initial commit

>Changes to be committed:
> (use "git rm --cached <file>..." to unstage)

>       new file:   driver.cpp
>       new file:   factorial_library/factorial.cpp
>       new file:   factorial_library/factorial.h

> Untracked files:
> (use "git add <file>..." to include in what will be committed)

>       ../.minitut_three.tex.kate-swp
>       ../minitut_three.tex
>       ../minitut_three.tex.backup
>       ../solution/
```

Git is telling us that we've added three files to the staging area. Now when we run the commit command the files (as they stand when we ran the add command) will be committed as one unit of work. So lets do this now.

```
git commit -a -m "Initial commit. Added the files the TA gave us to my repository. \\
  I'm now ready to do some actual work and I can roll back should I screw up!"
> [master (root-commit) ed50317] Initial commit. Added the files the TA gave us to my \\
 repository. I'm now ready to do some actual work and \\
 I can roll back should I screw up!
> 3 files changed, 50 insertions(+)
> create mode 100644 exercise/driver.cpp
> create mode 100644 exercise/factorial_library/factorial.cpp
> create mode 100644 exercise/factorial_library/factorial.h
```

The commit -a flag tells git to add any new changes that may have been made to previously added files to the list of files staged for commit. This flag can be very useful as a shortcut to individually re-adding changes to the staging area using `git add`. As with all shortcuts just make sure you don't accidentally stage files that should not have been staged for commit. The -m flag tells git to use the message specified as the commit message instead of firing up a full blown text editor for you to type in the message. Your commit message should always be very detailed, describing exactly what changes are being committed. This will help you to find a previous commit that may have introduced regression in your code and to find the segment of code that is causing problems.

Take a look at your git log. It will show you your commit history along with an identifier for each commit. You will use this unique checksum identifier each time you want to refer back to a specific commit.

```
git log
> commit ed5031779463e3ab74c66f26d2689a3a69be6bcd
> Author: bennahugo <bennahugo@aol.com>
> Date:   Wed Feb 11 11:54:01 2015 +0200

>   Initial commit. Added the files the TA gave us to my \\
    repository. I'm now ready to do some actual work and \\
    I can roll back should I screw up!
```

You can also use git to backup and transport your project code between your personal machine and the lab computers (say goodbye to the antiquated and fallible flash drive) using remote repositories. The CS department has a GitLab server where you can safely store your code / text files (please note: no binaries) and you will be required to use this for team collaboration on your capstone project. After you learned how to link things together you may want to look at the `Using the departmental GitLab server` document under resources on Vula. We also suggest that you work though the tutorial at the Software Carpentry website to learn how to collaborate and resolve conflicts `http://software-carpentry.org/v5/novice/git/index.html`.

Part Two: Compiling and linking

Explanation

As it stand the code I've provided to you is not very useful because it has not been compiled into an executable program. You should recall from last year that Makefiles provide an easy (well relatively easy way ;)) to tell the computer exactly how to go from source code to executable code. It is worth pointing out that Makefiles can be used anywhere where you have a well defined sequence of steps to take input and produce some output from it. Although it is predominantly used in build systems, it can also be used ensure repeatable experiments and work flows in other scientific disciplines.

A Makefile comprise of a number of rules which tell the computer exactly how to go from format A to format B to format C and so on till the final output product is created. In our case this will be our executable code. They take the form of

```
output_file: source_file_1 source_file_2 ... source_file_N
       invoke some process with inputs source_file_1 source_file_2 ... \\
       source_file_N and produce output_file
```

The rule above tells the following to Make: "output_file depends on files source_file_1 . . . source_file_N. If one of those dependencies have been modified since the last time output_file was produced or output_file does not exist yet then we need to reproduce output_file from scratch. We can produce output file by invoking some process (or sequence of processes) with the dependencies in order to produce an output file."

Therefore, if you have to compile some code that computes the shortest path in a graph you may have a set of rules such as the following, in a file called `Makefile` (case sensitive):

```
driver.run: driver.o graph.o node.o
       g++ -o driver.run driver.o graph.o node.o
driver.o: driver.cpp graph.h
       g++ -c -o driver.o driver.cpp
graph.o: graph.cpp graph.h node.h
       g++ -c -o graph.o: graph.cpp
node.o: node.cpp node.h
       g++ -c -o node.o node.cpp
```

When you run the command `make`, the program `GNU Make` will search for the file `Makefile` in the current path and execute the rules you've written in that file.

Note that the steps to take in the execution of each rule is indented by a tab space and not space characters. The C++ compiler will first compile each `cpp` source file into a standalone `object file`. These object files are then linked together into some executable code. Recall from lectures that `header files (.h files)` act like the catalogue of methods (or the skeleton outline of a class in the case of classes) that are implemented in a cpp file. This allows you to effectively separate declaration from implementation. We require that you always separate your code into headers and source files.

Here g++ refers to the GNU compiler suite's C++ compiler. The lab machines have GCC-4.8 installed. Ensure your machine is running the same version of GCC. We will not tolerate submission issues arising due to compilation on older versions of the compiler. It is your responsibility to check that you're running the correct version of GCC from the get go. On my local machine this will look like this:

```
g++ --version
> g++ (Ubuntu 4.8.2-19ubuntu1) 4.8.2
> Copyright (C) 2013 Free Software Foundation, Inc.
> This is free software; see the source for copying conditions.  There is NO
> warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

The `-c` flag tells the compiler to only compile the code and not to do linking at this stage. Simply compiling and linking everything together in one giant call to g++ is not good practice and we do not recommend it. When you start working on bigger systems next year you will see that it can take minutes to hours to compile an entire system. If you do not specify dependencies the compiler will happily go and redo unnecessary work.

A complete list of options for `gcc` is available here `https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html#Invoking-GCC`.

A more thorough discussion on makefiles can be viewed here `http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/`. The solution will contain a makefile that you can modify and use in your pracs.

Your task

- **Note: You are not to modify the provided code or the structure of the files in the directory in any way.**
- You are required to create a makefile (in the `factorial_library` directory) to compile a shared library (analogous to a Dynamic Link Library for those more familiar to Windows). To compile this first output object files and then link it into a source library. Write make rules to do the following tedious commands automatically.

  ```
  g++ -c -o factorial.o factorial.cpp -fPIC -shared -std=c++11
  g++ -o libbrain_dead_factorial.so factorial.o -fPIC -shared -std=c++11
  ```

- Commit this unit of work. **DO NOT ADD ANY OBJECT OR SHARED OBJECT FILES TO YOUR REPOSITORY... ONLY YOUR NEW MAKEFILE**
- Now that you have a library write a makefile (in the `exercise` directory) to compile the driver and link against this newly created library. Notice that the header file is inside the `factorial_library` directory. Normally gcc will go look in /usr/include and /usr/local/include amongst other places for headers. This directory is certainly not one of them. Use the `-I` flag to tell gcc to go and look for the header in the `factorial_library` directory. Now use -lbrain_dead_factorial to include libbrain_dead_factorial.so. Similarly tell gcc to go and look in our custom `factorial_library` directory through the -L flag. Your makefile should invoke the following commands:

  ```
  g++ -c -o driver.o driver.cpp -std=c++11 -I ./factorial_library/ \\
      -L ./factorial_library/ -lbrain_dead_factorial
  g++ -o driver driver.o -std=c++11 -I ./factorial_library/ \\
      -L ./factorial_library/ -lbrain_dead_factorial
  ```

- Commit this unit of work. **DO NOT ADD ANY OBJECT OR EXECUTABLE FILES TO YOUR REPOSITORY... ONLY YOUR NEW MAKEFILE**

- Now try and run the program. Remember to ensure that the file's execution permission is set to read, write, run for your user `chmod 700 driver` will do the trick. Execute the program by typing `./driver`. You'll get an error like this:

```
./driver: error while loading shared libraries: libbrain_dead_factorial.so: \\
          cannot open shared object file: No such file or directory
```

  Remember that the environment does not yet know about our custom library path. Just as with gcc we need to tell it where we're storing our custom libraries. You should recall from second year that the `LD_LIBRARY_PATH` environment variable contains all the custom directories where we store our libraries. Running `export LD_LIBRARY_PATH=factorial_library/ && ./driver` will do the trick. You will see the following output

```
> The factorial of 0 is: 1
> The factorial of 1 is: 1
> The factorial of 5 is: 120
> The factorial of 13 is: 6227020800
```

- **Challenge:** write a custom make rule to run your program when the user types in `make run`. Commit this unit of work.
- Check your commit log. You should see the commits you have done.
- Create a tarball of your exercise and submit this to the assignment tab. Make double sure that your tarball contains your .git repository along with your source files. Remember: if it does not contain a .git repository then you will be penalized. **NEVER SUBMIT BINARY CODE TO VULA**.