CS264 Laboratory Session 1

Dr. Edgar Galván

Tuesday Sept. 26^{th} 2017

Deadline: All solutions to be submitted by 5pm Tuesday 3^{rd} October 2017

1 Lab objectives

In this lab you will first begin to learn to use git for revision control and for managing your practical work during the course. You will then work through a number of introductory level C++ programming problems. The aim of these problems is to reinforce your understanding of the concepts discussed in the lectures and in the C++ tutorial.

Learning Outcomes
By the end of this lab sheet you should be able to:

1. Locally organise and manage your work with git.
2. Write simple C++ programs that make use of (i) primitive data types, (ii) the facilities of the iostream library, (iii) branching and looping control structures, and (iv) functions.

2 Introduction to git

With any piece of software, the associated codebase evolves starting with initial development, and then following through to testing, deployment and maintenance. As you become experienced with larger codebases, and in particular, codebases that are collaboratively developed by teams, the need for robust tools for managing the evolution of that codebase becomes critical. Probably the most important tool here is the revision control system (RCS).

To motivate need for an RCS, consider the scenario where you are working on a large codebase and need to add a new feature. The feature is going to make a number changes to the codebase, potentially across multiple files. Making such changes comes with risks in that if the changes go awry it could result in a perfectly good codebase ending up in a broken state. More importantly, reverting the changes can be difficult since it can require keeping track of updates to multiple separate files.

One possibility is to make a backup copy of the codebase prior to making the required changes, however this comes with many drawbacks. For example, unless you are particularly disciplined very quickly you can end up with multiple copies of the codebase without a clear indication of which version is which. Now imagine trying to use this strategy to coordinate the activities of multiple designers and developers all working on a large software development effort. Hopefully it is obvious that such an approach will be completely unmanageable.

An aim of an RCS is to provide a set of tools for managing and tracking changes such as the above in a controlled and reversable fashion. Examples of RCS's include: CVS, subversion, Microsoft Visual SourceSafe, Git, Mercurial, etc. In this course you will use git for managing all code you develop during the lab sessions. During this course you will use git to store and manage all code you write. One of the aims of this lab is to introduce the central concepts involved in

using git. Note that technically git is known as a *distributed version control system*. In today's lab we will only use the local version control features of git. In next week's lab we will see how to start to use the distributed aspects of the tool.

As part of the lab sheets and in the labs themselves you will be given information on how git works. However, you are encouraged to also take some time to read through some of the online `http://git-scm.com/doc` documentation, which is very accessible.

The essense of RCS's is to provide a *repository* to store and manage the codebase. Users make changes to the codebase by first *checking out* a local copy of the codbase (sometimes referred to as a *working copy* or *working directory*). Once the user has completed their changes they then store their changes back in the repository by *checking in* their changes.

The clever part in this is how the repository store updates to the codebase. Each time you make a checkin, git stores a new *snapshot* of the codebase including your new changes. As such the repository keeps track of all changes as a series of snapshots. This has a wide range of benefits the most obvious of which is the ability to roll back any changes that you make.

For example, imagine you have been trying to add a feature to the codebase over a few days, and checked in your changes at different points. Now lets say your manager has decided that the feature is going to take too long to complete and therefore wants you to roll back your changes. In *git* this is simply done by checking out the relevant previous snapshot (which it turns out can be done with a single command). In fact, better still, it is possible to shelve your development on the feature to date (in what's known as a *branch*), which effectively puts it to one side until you are ready to come back to it.

Note that this is only the start of what you can do with git. Again, have a look at the online documentation to find out more.

2.1 Getting setup

The first step in using git to track changes for a given code base is to initial a repository at the top level directory. To do this you should first create a folder somewhere in your home directory in which you will store your lab work for CS264. I would suggest that you create a folder called `CS264` and then create a subfolder within that called `labs`. Each week you should create a subfolder of `labs` called e.g. `lab1`, `lab2`, etc.

Step 0.1: To start we will setup the directory as far as `labs` and then change directory into `labs`. To do this, run the following commands on the command line:

```
cd ~
mkdir CS264
cd CS264
mkdir labs
cd labs
```

Ensuring that you are in the `labs` directory, we will now initialise a git repository to manage changes to all files and subfolders. This is done using the `git init` command.

Step 0.2: Initalise your lab git repository using the following command: `git init`

Note that in the above command `git` is the program for handling all git commands and `git init` tells `git` to run the `init` command to initialise a repository in the current directory. `git` provides extensive help files that can be accessed using `git help` to get a high-level menu, and `git help` *command* to get help on a specific command e.g. `git help init`.

2.2 Making changes

Now that we have setup the repository let's make some changes. First you should create a directory for today's lab.

Step 0.3: From the `labs` directory make a subdirectory called `lab1`: `mkdir lab1`

Note that lab1 has now been added to your working directory, but has **not** been added to your repository. Adding changes to your repository is a two step process. To add changes you first *stage* all the changes you want to commit, and then once everything is *staged* you *commit* the staged changes to the repository. Staging is done using the `git add` command, and committing is done using the `git commit` command.

Step 0.4: Stage the addition of the `lab1` folder using the following command: `git add lab1`

Next we will commit the single change that we have staged above. As mentioned above this will record a new snapshot in the repository, storing our latest changes, and bringing the repo up to date with our working directory. A very important part of this process from a management and communication point of view is to record a brief message explaining the nature of the changes being committed. This message can be reviewed later by both you and others through the `git log` command.

Step 0.5: Commit the staged changes using the following command: `git commit -m 'Added folder for lab session 1'`

The `-m` flag is used in the above command to specify the log message for the commit.

2.3 Adding a file

We will now add a source file to lab1 for exercise 1. To do this you should use your favorite text editor to save the following file as `exercise1.cpp` within the `lab1` subfolder.

```
#include <iostream>

using namespace std;

int main(){

}
```

Now add the cpp file to the repository.

Step 0.6: Change directory into the `lab1` subdirectory and then stage and commit the new file using the following commands:

```
git add exercise1.cpp
git commit -m 'Added some boiler plate code for exercise 1'
```

The new cpp file should now be tracked by the repo. To go one step further let's add one line to the `exercise1.cpp` file as follows. Make sure that you have saved the file after you make the change.

```
#include <iostream>

using namespace std;

int main(){
cout << ``Hello world!\n'';
}
```

The working directory now has changes that have not yet been committed. To see what changes you have made you can use `git diff exercise1.cpp`. Try it and see if you can figure out the output.

Finally let's stage and commit the changes that you have made.

Step 0.7: Change directory into the `lab1` subdirectory and then stage and commit the new file using the following commands:

```
git add exercise1.cpp
git commit -m 'Added hello world output to exercise 1'
```

2.4 As you continue....

As you continue through the lab you should stage and commit changes as you update files and complete particular tasks. At the very least you should have a separate commit for each of the exercises.

One very important point is that git should be use to store code and not executables. Executables are derived from your code and so it is redundant and inefficient to track them in the repository. So as you work through the labs you should really only be add .cpp and .h files (and folders of course) to the repo.

3 Beginning C++ Programming

For each of the problems given below write a C++ program that provides a solution that problem. Each box provides a filename to use. Please ensure that you use those filenames. Failure to do so can result in loss of marks. **You should include a comment at the top of each source file with your full name followed by your surname, the latter in capital letters (e.g., Edgar GALVAN) and student number (if you have one).**

Recall that you can compile a cpp program as follows: c++ -o nameOfProgram.o nameOfProgram.cpp

Exercise 1: Write a program that inputs three integers from the keyboard, and prints the sum, average, product, smallest and largest of these numbers. You should save the source in a file called `exercise1.cpp`.

Exercise 2: Write a program that reads in two integers and determines and prints if the first is a multiple of the second. You should save the source in a file called `exercise2.cpp`.

Exercise 3: Write a program that inputs a five-digit number, separates the number into its individual digits and prints the digits separated from one another by three spaces each. You should save the source in a file called `exercise3.cpp`.

Exercise 4: Develop a C++ program that will determine if a department-store customer has exceeded the credit limit on a charge account. For each customer, the following information is available:

1. account number (an integer);
2. balance at the beginning of the month;
3. total of all items charged by the customer this month;
4. total of all credits applied to the customer's account this month;
5. allowed credit limit

The program should input this information, calculate the new balance (= *beginning balance + charges - credits*) and determine if the new balance exceeds the customer's credit limit. For those customers whose credit limit is exceeded, the program should display the customer's account number, credit limit, new balance and the message "Credit limit exceeded."

The program should permit multiple customer details to be inputted with the above information being printed to the screen after each customer. The program should terminate when an account number of -1 is inputted.

You should save the source in a file called `exercise4.cpp`.

Exercise 5: An integer is said to be *prime* if it is divisible only by the two distinct factors 1 and itself. Write a function that determines if a number is prime. Use this function in a program that determines and prints all the prime numbers between 1 and 5000.

You should save the source in a file called `exercise5.cpp`.