Lab 1:Get familiar with Linux

By Dr. Yingwu Zhu, Seattle University

Tasks (You should perform the following tasks step by step):

1. Check out your CS1 account
2. Set up your CPSC250 class directory (using basic Linux commands)
3. Emacs editor for coding
4. GNU C++ compiler: compile and run a simple program
5. Create and use Makefiles

Step 1: Check out your CS1 account

All the programming assignments and labs will be done on the department Linux server **`cs1.seattleu.edu`**. If your account has been established, you can login to **`cs1`** using your regular SU account name and password with a telnet program – on our lab machines that program is **Putty** or **SSH Client**.

Start **Putty** (or **SSH client**), and login to **`cs1.seattleu.edu`** using your SU account name and password.

Note that in order to do the programming/lab assignments from your home/dorm, you need to install **Putty** (or **SSH client**) on your laptop/desktop. Then you can access **`cs1.seattleu.edu`** as described above. Go to
http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html
to download **`putty.exe`** according to the OS in your machine.

==========

Step 2: Set up a directory for CPSC 250 class

If you have passed Step 1, you are now in your home directory.

Don't believe it? Try to type the command at the prompt:

```
pwd
```

then press enter. See what you get? **`pwd`** shows your current directory!

It is recommended that you create a directory for your CPSC 250 class. While you can use your home directory for this, it is easier to keep track of different course work if you create a directory for each class.

To create a directory for CPSC 250 named **cpsc250**, type the following command at the prompt:

```
mkdir cpsc250
```

then press enter. If the directory is correctly made, there will be no response from the system except the return of your prompt.

Now see if the directory **cpsc250** is successfully created, type the following command at the prompt:

```
ls
```

then press enter. If the directory is correctly created, you will see it on the list.

Now, your working directory is still your home directory, you can change to the newly created directory typing the following command at the prompt:

```
cd cpsc250
```

then press enter. Your prompt will return showing the new directory. You are now ready to work on your CPSC 250 projects, labs, and programming assignments. Create a directory for Lab 1.

```
mkdir lab1
cd lab1
```

As mentioned earlier, to view your working directory type the following command at the prompt:

```
pwd
```

then press enter. It will show you a path indicating where you are.

Anytime when you want to go to a upper-level directory, type the following command at the prompt:

```
cd ..
```

then press enter. You move one level up on the directory tree.

==========

Step 3: Emacs editor for coding

Make sure your working directory is **lab1**. Create and edit a C++ program called **welcome.cpp** (shown below). To create and edit the C++ program, use the following command. You can also find an introduction to emacs in the Linux Pocket Guide.

```
emacs  welcome.cpp
```

Here is the program.

```
/*
   Author: xxxxx (your name)
   Program: welcome.cpp
   Last Modification: xxx (date)
*/

/*
   Description: this program print a welcome message to
   the name you input.
*/

//Assumption: none

#include <iostream>
#include <string>
using namespace std;

int main()
{
      string name;

      cout << "What is your name? ";
      cin >> name;
      cout  <<  "Welcome " << name << "!" << endl;
      return 0;
}
```

Type the program shown above in emacs.

When you finish the typing, you need to save the file. Press "**ctrl+x**" (hold Ctrl key and press x key on the keyboard) first, wait for a second, then press "**ctrl+c**" (hold Ctrl key and press c key). A message will show up at the bottom to ask you if you want to save the file, type "**y**" to save the file. Note that when you finish editing a file, you always need to save it before quitting the emacs editor.

==========

Step 4: Compile and run the program

To compile the program **welcome.cpp**, enter the following command at the prompt:

```
g++ -Wall -o welcome welcome.cpp
```

Note that, there are five parts in this command, with a space to separate them with each other. **g++** is the command used to compile a C++ program; **-Wall** and **-o** are two of those options used for **g++** compiler, which display all warning messages and allow users to name our own executable, respectively; **welcome** is the executable (generally speaking, you can name the executable any name you want); **welcome.cpp** is the C++ program you want to compile.

Therefore, if we have another C++ program, **`example.cpp`**, to compile this program, we should follow the same format:

```
g++ -Wall -o example example.cpp
```

If errors (and warning) occur during the compiling process (they will be listed on the screen), you will need to *edit* your program and make corrections. Follow the directions on the reference card to open the file you created and make corrections, then compile again. "No news is good news" – in other words, if you simply see the LINUX prompt again after compiling, then there were no errors.

To run the program, enter at your LINUX prompt,

```
./welcome
```

You can verify that you have a compiled unit by typing the list command

```
ls
```

All files in your directory will then be listed and you should see **`welcome.cpp`** (the source file) and **`welcome`** (the executable file).

Playing around is encouraged! Try editing the file to change the output message. Then repeat the compilation and execution steps. More complicated programs may take several rounds of editing and compilation.

==========

Step 5: Create and use makefiles

Makefiles make compiling easier for multiple source files. For detailed information about makefiles, please refer to
http://www.cs.umd.edu/class/spring2002/cmsc214/Tutorial/makefile.html

A makefile typically consists of many entries. Each entry has:

- a target (usually a file)
- the dependencies (files which the target depends on)
- and commands to run, based on the target and dependencies.

Let's look at a simple example:

```
Movie.o: Movie.cpp Movie.h Vector.h
        g++ -Wall -c Movie.cpp
```

The basic syntax of an entry looks like (**put a TAB at the beginning of commands**!):

```
<target>: [ <dependency > ]*
   [ <TAB> <command> <endl> ]+
```

.o files and executables can be as targets!

Let's see an example that prints the n-th Fibonacci number. The program consists of 3 source files:

```
//fib.h
#ifndef _FIB_H
#define _FIB_H

int fib(int n);

#endif
```

```
//fib.cpp
#include <cassert>

int fib(int n) {
        assert(n >= 0);
        if (n < 2)
                return n;
        int a = 0;
        int b = 1;
        for (int i = 2; i <= n; i += 2) {
                a += b;
                b += a;
        }
        return n % 2 ? b : a;
}
```

```
//testfib.cpp
#include <iostream>
#include "fib.h"

using namespace std;

int main(int argc, char* argv[]) {
        if (argc < 2) {
                cout << "Usage: " << argv[0] << " n\n";
                return -1;
        }
        int n = atoi(argv[1]);
        cout << "The " << n << "-th fibonacci number is: " <<
fib(n) << endl;
        return 0;
}
```

```
#Makefile
OBJS = fib.o testfib.o
CC = g++
DEBUG = -g
CFLAGS = -Wall -c $(DEBUG)
LFLAGS = -Wall $(DEBUG)

testfib: $(OBJS)
        $(CC) $(LFLAGS) $(OBJS) -o testfib

testfib.o: testfib.cpp fib.h
        $(CC) $(CFLAGS) testfib.cpp

fib.o: fib.cpp fib.h
        $(CC) $(CFLAGS) fib.cpp

clean:
        \rm *.o *~ testfib
```

Create all these files under your **lab1** directory. Execute the following commands:

```
make

./testfib 12

make clean
```

If you have accomplished all the steps above, congratulation!