EECS 280 Lab 01: Linux

Due Friday, 17 January 2014, 11:55pm

In this lab, you will connect to **CAEN Linux** and use basic Linux commands to navigate files and directories, and to **write**, **compile**, **run**, and **test** a simple program. You will also learn how to use a **debugger** to step through code execution, set breakpoints, and inspect variables.

CAEN manages many of the computers used by the College of Engineering, and we will use the CAEN linux environment for this course. If you don't already have a CAEN account, you will need to work with someone else who does for this lab. Afterward, make your way to the CAEN Hotline in room 1315 of the Duderstadt Center (in the middle area behind the circulation desk) to set up an account ASAP.

This lab covers material from these lectures:

- 01 Intro

Overview

Requirements

You may work on this lab either individually or in groups of 2-3. Include your name(s) in the comments at the top of the file. Submit the files below on CTools. You do not need to turn in any other files. If you work in a group, each person must submit a copy in order to receive credit.

Files to submit:

- `lab01.cpp`

Completion Criteria/Checklist:

To pass this lab, you must finish tasks 0-4.
This checklist will give you an idea of what we look for when grading for completion:

- ✓ (Task 1) Add the "Goodbye World" line to the code.
- ✓ (Task 4) Fix the bug discussed in tasks 3 and 4.

Task 0 - Preliminaries

Connect to a CAEN Linux Environment

Refer to QR - CAEN Linux for instructions on several different ways to connect or work locally on your own computer.

Open a Terminal and Navigate to Your Home Directory

If you're using SSH, you already have access to a terminal and should already be in the right place. If you're using VNC, you will need to launch a terminal from the Applications > System Tools menu or right click on the home directory icon on your desktop and select "Open in Terminal".

To check that you are in your home directory, use the `pwd` ("print working directory") command. It should look like "`/afs/umich.edu/user/u/n/<your uniqname>`", where `'u'` and `'n'` are the first and second letters of your uniqname, respectively. Don't type the `$` sign below when you enter commands - it's just there to indicate something that should be typed at the terminal.

```
$ pwd
/afs/umich.edu/user/u/n/<your uniqname>
```

If you're not in your home directory, you can get there by typing this:

```
$ cd ~
```

Create `eecs280` and `lab01` Directories

There are many Linux commands for navigating and working with directories on the file system. Here are a few basics:

- `pwd` - prints the current working directory
- `ls` - prints a list of the files and subdirectories in the current working directory
- `cd` - changes to a different directory
- `mkdir` - creates a new directory
- `cp` - copies a file from one place to another

The `man` command displays documentation for a command. To scroll down, just press enter or the down arrow. To exit the documentation, press q. For example, if you want to know more about `cd`, type:

```
$ man cd
```

You can even do:

```
$ man man
```

Complete the following steps.

1. Create a new directory called `eecs280` with the command

   ```
   $ mkdir eecs280
   ```

2. Change the working directory to the one you just created. *Pro tip:* try using the tab key to auto-complete the name of your directory after you type the first few letters.
   ```
   $ cd eecs280
   ```

3. Create another directory called `lab01`. Use the same command as step 1, but with a different directory name. *Pro tip:* you can press the "up" key twice to get the command from your history.
   ```
   $ mkdir lab01
   ```

4. Change the working directory to `lab01`.
   ```
   $ cd lab01
   ```

Task 1 - Hello World

For this task, you'll work with a simple program we've already started for you. It should print out "Hello World!", prompt for world's population, and print a different message depending on whether the population was 0, between 0 and 100, or greater than 100. The program itself is trivial, but the main goal is to make sure you're comfortable editing, compiling, running, and testing C++ files.

1. Copy the `lab01.cpp` starter file from the course directory to this directory, using the cp command:
   ```
   $ cp /afs/umich.edu/class/eecs280/lab/lab01/lab01.cpp .
   ```

   Don't forget the period at the end! The single period means "my current directory." Thus, the entire command means, "copy the file found at `/afs/umich.edu/class/eecs280/lab/lab01/lab01.cpp` to my current directory."

   In case you are working locally, the file is also attached to the CTools assignment.

2. Verify that the copy succeeded by listing all the files in your current directory. You should see only the name of the file "`lab01.cpp`".
   ```
   $ ls
   lab01.cpp
   ```

3. Open `lab01.cpp` for editing with the text editor of your choice. (See QR - CAEN Linux for a few options if you need.) Add a line of code to the file so that `"Goodbye World!"` (followed by a newline) is printed after everything else.

4. We will use the g++ compiler in this class. Compile your code with the following command:
   ```
   $ g++ lab01.cpp -o lab01
   ```

Some explanation:

`lab01.cpp`
: The source file we want to compile.

`-o`
: An option that lets you specify the name of the executable produced. (You can think of it as being short for "output program name.") Whatever immediately follows will be the name of your executable. In this case, it's "`lab01`".

BEWARE: if you accidentally type something like "`g++ lab01.cpp -o lab01.cpp`", then the compiler will overwrite your source file with a compiled binary, and you can't get your source code back! If you do this, the source code you wrote is gone forever.

Always be careful when there's a possibility of overwriting files in Linux. There is no "undo" on the Linux command line, and in many cases you won't even get a warning before a file is overwritten. We encourage you to make frequent backups of your code when working on projects.

5. If you get an error message, go back and check over the changes you made to the code. Once you are able to compile successfully, run `ls` again. You should see a file called `lab01`. Depending on the editor, you might also see a temporary file like `lab01.cpp~`.

```
$ ls
lab01 lab01.cpp
```

6. Run your program using the command shown below. (Note the "./" before the name of your executable.) Type 100 for the World's population and press enter when prompted. Your output should look like below.

```
$ ./lab01
Hello World!
How many people are there?
100
It's a small world after all!
Goodbye World!
```

Task 2 - Testing

1. Sometimes we would like to store the results of our program in a file. The > symbol does stream redirection - it redirects terminal output to a file. You'll see no output on the screen, but the file will now contain the output of the command that comes before the > symbol. (Use a population of 250000 from now on.)

```
$ ./lab01 > lab01.out
250000
```

2. One thing that's kind of awkward is that our program expects input (the population), but because we redirected all output to the file, we don't see the prompt "How many people are there?". So let's look at an alternative to stream redirection. We can fix the problem by piping the output of our program to the `tee` command, which writes the output to both the console and the file we specify.

```
$ ./lab01 | tee lab01.out
Hello World!
How many people are there?
250000
It's a small world after all!
Goodbye World!
```

3. You can check the contents of your newly created file by opening it in a text editor or by printing it to the terminal using the `cat` command. Note the 250000 isn't there (why?).

```
$ cat lab01.out
Hello World!
How many people are there?
It's a small world after all!
Goodbye World!
```

4. Now, let's automatically test the output stored in `lab01.out`. We've provided a file called `lab01.out.correct` containing the correct output when a population of `250000` is used. Copy it to your folder using:

```
$ cp /afs/umich.edu/class/eecs280/lab/lab01/lab01.out.correct .
```

   Use the `diff` command to find the differences between two files, and print the differing lines to the screen. If there is no output, it means that the files are exactly the same. If there is any output at all, it means the files are different in some way, even if it is something small like an extra space, which could be hard to spot otherwise.

```
$ diff lab01.out.correct lab01.out
3c3
< It's a big world out there!
---
> It's a small world after all!
```

   The correct output says it's a big world, but our code produced the small message. Unfortunately it looks like we've got a bug somewhere...

Task 3 - Debugging

Although the sharp-eyed student may have already spotted the bug in our code, we want to give you a brief introduction to using a debugger. *Being proficient with a debugger will make your life much easier throughout this class.*

Before we begin, you'll need to recompile your code with the `-g` option, which tells the compiler to include extra information that can be used for debugging purposes. It may also be useful to leave out the -O1 option that can sometimes make the compiled program less comprehensible to debuggers. (http://sourceware.org/gdb/onlinedocs/gdb/Optimized-Code.html)

```
$ g++ -g lab01.cpp -o lab01
```

gdb is the "officially supported" debugging tool that we'll use in this course, but you are welcome to use other debuggers such as Xcode or Visual Studio. QR - Debugging has more details.

To begin, use the `gdb` command with the name of the program to be debugged as an argument.

```
$ gdb lab01
```

If you compiled correctly with the `-g` option, the last line printed should say "`Reading symbols from <filepath>…done.`"

1. To run the program within gdb, use the `run` command. (Note that you don't type `(gdb)` since it's the prompt, just like `$`):

```
(gdb) run
Starting program: /afs/umich.edu/user/j/j/jjuett/
eecs280fall2013/eecs280/lab01/lab01
Hello World!
How many people are there?
250000
It's a small world after all!
Goodbye World!

Program exited normally.
Missing separate debuginfos, use: debuginfo-install
glibc-2.12-1.107.el6_4.4.x86_64 libgcc-4.4.7-3.el6.x86_64
libstdc++-4.4.7-3.el6.x86_64
```

2. If you want to step through the execution of your code one line at a time, starting at the very beginning of the program, use the **`start`** command.

```
(gdb) start
Temporary breakpoint 1 at 0x4008e0: file lab01.cpp, line 19.
Starting program: /afs/umich.edu/user/j/j/jjuett/
eecs280fall2013/eecs280/lab01/lab01

Temporary breakpoint 1, main (argc=1, argv=0x7fffffffe528) at
lab01.cpp:19
```

```
19    cout << "Hello World!" << endl;
```

This starts the program, but breaks (pauses) right away. At the end of the output is the line about to be executed, in this case the first line of code in your main function.

3. To move through the program, use the **`next`** and **`step`** commands. Both move forward to the next piece of code, but the `next` command will *step over* function calls while the `step` command will *step into* function calls. Figures 1 and 2 illustrate this - try one or both of them in your own terminal!.

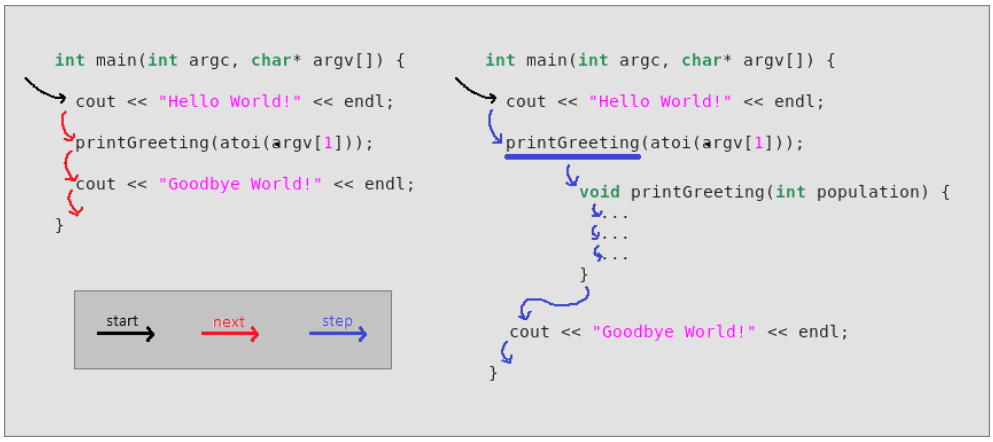

Figure 1. `next` will execute subfunctions all at once while `step` will enter subfunctions and execute them one line at a time.

```
(gdb) next
Hello World!
20        cout << "How many people are
there?" << endl;
(gdb) next
How many people are there?
22        cin >> pop;
(gdb) next
250000
23        printGreeting(pop);
(gdb) next









It's a small world after all!




24        cout << "Goodbye World!" << endl;
(gdb) next
Goodbye World!
25      }
```

```
(gdb) step
Hello World!
20        cout << "How many people are
there?" << endl;
(gdb) step
How many people are there?
22        cin >> pop;
(gdb) step
250000
23        printGreeting(pop);
(gdb) step
printGreeting (population=250000) at
lab01.cpp:7
7         if (population = 0) {
(gdb) step
10        else if (population < 100) {
(gdb) step
11          cout << "It's a small world
after all!" << endl;
(gdb) step
It's a small world after all!
16      }
(gdb) step
main (argc=1, argv=0x7fffffffe528) at
lab01.cpp:24
24        cout << "Goodbye World!" << endl;
(gdb) step
Goodbye World!
25      }
```

```
(gdb) next
0x0000003c69e1ecdd in __libc_start_main ()
from /lib64/libc.so.6
(gdb) next
Single stepping until exit from function
__libc_start_main,
which has no line number information.

Program exited normally.
```

```
(gdb) step
0x0000003c69e1ecdd in __libc_start_main ()
from /lib64/libc.so.6
(gdb) step
Single stepping until exit from function
__libc_start_main,
which has no line number information.

Program exited normally.
```

Figure 2. Using `next` vs. `step` in gdb (`start` was previously called).

If we use `next`, the lines of code within the `printGreeting()` are executed all at once and are not individually shown - we *stepped over* `printGreeting()`. While stepping through code, you can also type continue to resume execution of the code by typing **`continue`**.

Pro tip: Instead of typing `next` or `step` repeatedly, you can just press enter again to repeat the previous command. gdb also offers command completion if you press tab and will even recognize some abbreviated commands (e.g. `n` for `next`, `s` for `step`, `b` for `break`). If you'd like to know more, just use the **`help`** command.

4. Since the wrong message is printed in our program output, it's reasonable to assume the bug is in the `printGreeting()` function. We could use the approach above by using `step` (why not `next`?) until we get to the desired place in our code, but this would get really annoying for larger, more complex programs.

   A better approach is to set a *breakpoint* at the desired line in our code that tells the debugger to pause execution whenever that line is reached. The first line of code in `printGreeting()` is line 7 (your modifications shouldn't have affected this), so let's set a breakpoint there and run the program.

```
(gdb) break 7
Breakpoint 2 at 0x40089f: file lab01.cpp, line 7.

(gdb) run
Starting program:
/afs/umich.edu/user/j/j/jjuett/eecs280fall2013/eecs280/lab01/
lab01
Hello World!
How many people are there?
250000

Breakpoint 2, printGreeting (population=250000) at
lab01.cpp:7
7    if (population = 0) {
```

5. Notice that gdb's output shows the value of each variable in scope when the breakpoint occurs (in this case, right before the execution of line 7). The current value of `population` is `250000`, which is fine, so let's go ahead and step forward.

```
(gdb) step
10   else if (population < 100) {
```

To see the value of `population` again, use the **`print`** command.

```
(gdb) print population
$1 = 0
```

That's weird, population got set to 0. How did that happen? This should be the clue you need to find the bug. (Many of you may have already figured it out - it's a classic!) It seems some careless GSI typed = instead of == in the condition of the `if`, so an assignment rather than a comparison is happening.

Exit the gdb debugger by typing `quit`.

```
(gdb) quit
A debugging session is active.

   Inferior 1 [process 26793] will be killed.

Quit anyway? (y or n) y
```

The g++ compiler is quite sophisticated, and it is actually able to detect many kinds of possible bugs and issue warnings about them. To enable this, we just have to add a few more options to our compilation command.

```
$ g++ -Wall -Werror -O1 -pedantic -g lab01.cpp -o lab01
cc1plus: warnings being treated as errors
lab01.cpp: In function 'void printGreeting(int)':
lab01.cpp:7: error: suggest parentheses around assignment used as
truth value
```

In this case the compiler is able to warn us about the bug. Don't get used to it - you'll still need to use the debugger in many cases.

Here are the details for these new options:

`-Wall`
An option that turns on warnings about many things -- "Warn All".

`-Werror`
An option that treats all warnings as errors (and so your code won't compile if you have warnings). This is good because if you have warnings, your code is likely either wrong or has

unnecessary things in it.

`-O1`

Turns on the first level of optimizations. This generates some additional warnings that `-Wall` doesn't. (If it isn't clear, it's a capital letter O, not a zero.)

`-pedantic`

An option that instructs the compiler to issue all warnings *required* by the ISO C++ standard and reject any forbidden extensions. Without `-pedantic`, g++ allows some additional "features" or coding practices *explicitly* forbidden by ISO C++, which might reduce the portability of your code.

Task 4 - Wrap-Up

Make the appropriate change in your `lab01.cpp` file to fix the bug and recompile using:

```
$ g++ -Wall -Werror -O1 -pedantic -g lab01.cpp -o lab01
```

There shouldn't be any errors or warnings. Run your program again and write the output to lab01.out using:

```
$ ./lab01 | tee lab01.out
Hello World!
How many people are there?
250000
It's a big world out there!
Goodbye World!
```

One last time, check your program's output against the solution. Nothing should be printed when you test with the `diff` command, which means the files match exactly. If anything is printed, something's not right.

```
$ diff lab01.out.correct lab01.out
```

Turn in your modified `lab01.cpp` file on CTools. See QR - Labs for more details on doing this.