# Lab 1
## Hello, world!

*1 September 2015*

The drill for this lab is the **Chapter 2 drill** (p53). Come to lab on Tuesday either with it completed or with a specific written question in your notebook identifying which drill step you got to and what about it you're stuck on.

Notes and adjustments:

- As noted in class on Friday, the steps on this one are geared to Windows users. For step 1, "set up a project" can be interpreted as "create a directory for this lab" (using `mkdir` as described last week). For step 2, type in the program but omit the `keep_window_open` line, and ignore the rest of the stuff about what to do if `std_lib_facilities.h` is missing.
- Also as mentioned on Friday, remember that the command to compile is `compile filename.cpp` (replacing the filename as appropriate), and when the compiling succeeds, you'll run your program by typing `./a.out` on the command line.
- For step 4, take notes on the errors and error messages.

  Specifically, I want you to come out of this with a page or section of your notebook that has a table of the general form

  | Error message the compiler gives me | What it meant/how I fixed it |
  |---|---|
  | ... | ... |

  It is primarily for your own benefit, but I'll ask to see it sometimes and we'll be talking about it occasionally in class. Leave yourself enough room that you can keep adding to it: as you edit more and more programs, you will continue to encounter more and different error messages, and these notes will be a useful reference for you.

I'll be circulating around the lab to answer questions. If you're stuck on some part of the drill, ask me about that (and while you're waiting for me to get to you, look at the next section about man pages). If you're not stuck but haven't finished the drill, work on that now. If you're done with the drill (including making notes about the errors), go on to the next section.

## Command line feature of the day: Man pages

Since the early days of Unix back in the '70s, a standard feature has been the inclusion of electronic copies of the user and programmer manuals. They were implemented by means of the "`man`" program, and "man pages" exist for most command-line commands that you might use. Type

```
man ls
```

, for instance, and you'll see a description of the program that lists files and directories. (Use the arrow keys, PageUp, etc to scroll the page, and hit 'q' to quit.) Under "SYNOPSIS" you can see how it's called—by typing "`ls`" followed by options (if any) and then filenames (if any). Under "DESCRIPTION" is a listing of all the possible options. One is `-a`, which makes `ls` list all entries, even if they start with a period. In another window, type

```
ls -a
```

Scrolling down the manpage, check out what the often-useful `-l`, `-t`, and `-R` do. Try them out.

Technically, it is not the `man` program itself that is displaying this helpful information; it just pulls the text out of a database, and a program called `less` does the displaying. Type

```
man less
```

to see some of the different commands that you can use inside that program.

For that matter, you can

```
man man
```

to find out more about how to run the `man` program, like how to display all the pages in the manual on a particular topic, rather than just the first.

## After the drill

After you've completed the steps listed in this chapter's drill, I have a few more things I want you to do for this lab.

## Documentation

One program development habit I want you to start building right away is to always include at least minimal documentation with anything you submit. Because your lab submissions will generally be directories full of stuff, there should always be one easy-to-find file that says what the stuff is and what to do with it. By convention, this file is usually called `README.TXT` (and if that's the file we look for first, it is by definition easy to find).

In any programs you hand in for this course—and in any future courses you take from me, for that matter—I *always* expect a readme of some sort, even if it's very minimal. Always always. At a bare minimum it needs to say who you are, what it's supposed to do, and how to compile/run/test it or otherwise make it do what it's supposed to.

Our program doesn't do much, so for now the file won't be very long. Use vim to edit a file named `README.TXT`, and into it type the following text (filling in blanks as appropriate):

```
Lab1:  Hello
Date:  _____
            TODAY'S DATE
Author: _____________________________________
                    YOUR NAME

The first program prints "Hello, world!" on a line by itself.
To compile:
     compile hello_world.cpp
To run:
     ./a.out
```

(Note that if your source code is not named `hello_world.cpp` you should make your readme reflect what it's actually called!) That's all you have to do with the `README.TXT` file (for now). Save it and quit.

## Playing with the program

First of all, if your `hello_world.cpp` still has an error in it from the end of the drill, fix the error (and make sure that it compiles and runs correctly). Then, in the same directory:

- Make another program, that prints some other one-line message (such as "Hello, CMSC160!", but feel free to be creative). Hint: most of this program will be exactly the same as `hello_world.cpp`.
- Make a third program, that prints something that is at least five lines long, maybe a very short story or perhaps a picture—you may have noticed that the terminal window uses a monospace font, and you can take advantage of that to make artwork. (Note that the backslash is a special character in quoted text, so if you want to use it in your artwork, type two backslashes to generate a single printed backslash.) The length of the output should because you actually end the lines, not simply by making them so long that they wrap to the next line on the screen.
- Update your readme file (`README.TXT`) to also include descriptions of these additional programs, including what to type to compile them and then how to run them. Since this is all in a single directory, there should only be a single readme file, but it will have descriptions and instructions for all the programs you've written for the assignment.

Don't forget that part of developing a program is testing it: after you've typed in what you think will work for the second and third programs, you still have to compile it and run it to be sure it does what you expect.

## Rubric

Each full-length lab this term will include, at the end of its handout, the rubric by which I will grade it. This can be a sort of to-do list to make sure you haven't forgotten anything, and as the labs get longer and crunch time gets crunchier, the rubric can help you triage which features are worth more points and hence are more important to work on.

RUBRIC

**2** Present in lab, with completed drill or question about it
**Drill**
**1** `hello` source file exists
**1** ...and it compiles and works
**1** ...and I've seen your table of errors and explanations (in notebook ok)
**Additional**
**1** Second program prints different line
**1** Third program runs, prints something
**1** ... prints five or more lines, using newlines
**1** Readme file covers req'd details for hello program
**1** ... and other two programs

### Rubric check bonus

I'm trying something new this term to encourage everybody to really carefully check the rubrics each week. In your readme file, include a breakdown of all the rubric points you think you got. (Line-by-line is fine; prose like "all three drill points" is ok too as long as it's clear.) When I grade your lab, if my assessment matches yours—not just the total but on each point—then you'll get a successful rubric check mark for that week's lab.

At the end of the term, you'll get a half point for each successful rubric check.

Note that the rubric check is quite independent of the rest of your score that week—if you have a rough week and don't get much past the drill and think you're getting a 4, and you say so (and say which four points you're getting), that's a successful rubric check. My hope is that this will both boost scores (make sure you don't miss any easy points you might have forgotten about) and reduce surprises (because you know what score you'll be getting), but self-assessment and metacognition—knowing what you don't know and still need to work on—are useful skills in their own right, and the points for this are available to everyone.

## Handing in

If you don't finish all of that during the lab period (especially if you had technical difficulties during the drill, or are being very creative with the

third program), that's fine. This lab will be due **this Friday the 4th at 4pm**. (In general, final work for a lab will be due the following Monday at 4pm; this lab is short, though, and Monday is Labor Day.)

Your work for this lab includes multiple files within a directory (maybe you named the directory `lab1`, or maybe `hello` or something else). As you did last week during Lab 0, first make sure you are not "in" the directory with the files. Then, if your directory is named `lab1`, you would type

```
handin cmsc160-3 lab1 lab1
```

If your directory were named `hello`, you would instead type

```
handin cmsc160-3 lab1 hello
```

and in general, that third command line argument will be whatever directory you want to be handing in.

If all goes well, you should get a message to that effect telling you which files got handed in. If something is amiss, there will be an error message to tell you what to fix. Whatever it says, do read the message—it's generally fairly descriptive about what happened, whether successful or not.