

CS351, Fall 2022

First Lab: Preliminaries*

Assigned: Aug. 30, Due: Sep. 12, 11:59PM

Acknowledgement: TA Alexander Wolosewicz (awolosewicz@hawk) is the lead person for this lab. If you have questions about this lab, contact your TA. They'll filter them on to the lead as needed—don't contact the lead person directly unless they're your TA.

1 Objectives

You'll write plenty of code this semester, but before you get to it you first need to learn how to edit, test, and submit your work. We'll cover those steps in this lab.

- Log in to `fourier` (the course server) with SSH
- Learn to use a terminal multiplexer (`tmux`)
- Obtaining lab assignments on `fourier`
- Edit files on `fourier`
- Completing lab assignments on `fourier`
- Compile C code with `gcc`
- Use `make` to run tasks
- Makefiles: Understand the basics of how the “make” build tool works

*Acknowledgement: This lab is based on earlier IIT CS351 material by Michael Lee.

2 Log in to `fourier` (the course server) with SSH

At this point you should've received an e-mail containing login credentials for `fourier.cs.iit.edu`, a CS department Linux server on which you work on machine problems for this class. Using `fourier` will ensure a consistent working environment, and we will be using that same system to evaluate your submissions. If you're interested in working on your own Linux machine (or virtual machine), this is possible for many of the labs—but it is essential that your lab assignments compile and work on `fourier`.

To log in to `fourier`, you'll need an SSH client. At a terminal on most computers running Linux, macOS, or Windows, you can do this with the command:

```
> ssh username@hostname
```

where “>” is the command prompt, and “hostname” is `fourier.cs.iit.edu`. In lab assignments the prompt can also be shown as “`unix>`”, and on your computer it can show up as “\$”, but this is a customisable detail.

On Windows you may need to download a separate SSH client such as PuTTY if you don't have a command line client.

After logging in, resist the overwhelming urge to immediately start being productive! Repeat after me: TERMINAL MULTIPLEXERS ARE AWESOME. You'll learn about them next.

3 Learn to use a terminal multiplexer (tmux)

Lab assignments require multiple coding, testing, and debugging sessions to complete, which require multiple corresponding login sessions on `fourier`. Each time you login, you'll have to navigate into the appropriate work directory, re-open source file(s) in editors, start up debugging sessions, run trace files to obtain output, etc. All before you can even think about getting work done.

All the above constitute annoying cognitive overhead, and we can all use less of that. Wouldn't it be great if you could set things up — your editing windows, debuggers, trace output position — just the way you like it, one time, then have it remain that way each time you log back in? That's where terminal multiplexers fit in!

A terminal multiplexer is a program that:

- Allows you to create and manage multiple terminal sessions from a single screen. You can run a different program in each session (e.g., editor, debugger, shell).

- Continues to run after you log out, so when you reconnect you can simply pick up where you left off.

Using a terminal multiplexer will save you time and effort! `tmux` is a widely-used terminal multiplexer, and you can use it on `fourier`.

i To start `tmux`, just use the command `tmux`. At labs you can be walked through a few demo sessions. The important keys are as follows:

C-b : (control-b) is the default prefix key—i.e., it prefixes all the following command keys.

c : create a new window

n/p : change to the next/previous window

" : split the current window

<space> : arrange the panes in the current window according to some preset

o : change to the next pane in the current window

! : break the current pane out of the current window

? : list all key bindings

d : detach from `tmux`

After detaching from a session, the following command will re-attach you to it (given that you only have one `tmux` session running):

```
> tmux at
```

You should **NOT** create a new `tmux` session (with the command “`tmux`”) each time you log in. Instead, you need only do this once — on subsequent logins you will simply reattach to your existing session with “`tmux at`”.

So... Logged in: check . `tmux` session attached: check . Moving on.

4 Obtaining lab assignments on `fourier`

Lab assignments will be deployed to a directory called `labs/` that'll be created in your home directory. Each lab assignment will have its own directory inside `labs/`. The one for preliminaries will be called “preliminaries-handout”—so its full path will be `~/labs/preliminaries-handout`.



All the labs' directories will end in "-handout".

Try out the following interaction through your shell on `fourier`, and confirm that you get the similar output:

```
# We're currently in your home directory.
# Initially it only contains the labs/ directory.
> ls -lh
total 4.0K
... . ... cs351TA 4.0K Aug 18 14:13 labs

# Unimportant details are abstracted as ellipses.
> ls -lh labs/
total 8.0K
... . ... cs351TA .... Aug .. ..:.. preliminaries-handout
... . ... cs351TA .... Aug .. ..:.. README

# Above we see preliminaries-handout, the directory of your
first lab assignment.
# Let's look inside preliminaries-handout next.
> ls -lh labs/preliminaries-handout/
total 16K
... . ... cs351TA ... Aug .. ..:.. hello.c
... . ... cs351TA ... Aug .. ..:.. hello.h
... . ... cs351TA ... Aug .. ..:.. main.c
... . ... cs351TA ... Aug .. ..:.. Makefile
```


5 Editing files

A good programmer is disciplined in their choice of programming languages, coding conventions, and text editors. Emacs and Vim are two widely-used UNIX text editors. They're extremely featureful and flexible. Both are installed on `fourier`. Pick one (or both) and learn a new feature every day.

You can also use an editor or IDE on your own computer to edit files remotely on `fourier`. Editors/IDEs such as Atom and Visual Studio Code have good support for such workflows.

6 Completing lab assignments on `fourier`

Once a lab assignment is assigned to you on `fourier` as described above, you can then complete the assignment by following the instructions that are specific for that assignment.

 Go to your `~/labs/preliminaries-handout` directory, then open and read the “`main.c`” file located there. For this lab assignment you’ll find instructions in `main.c`; subsequent lab assignments will have a README file.

For this lab assignment, look for TODO comments in them in `main.c`. The TODO comments provide instructions for completing this assignment: you’ll see that you need to initialize some variables and write some C code. After doing this, save your changes.

To submit the assignment, you do not need to do anything after saving your changes. At the deadline, the lab assignment will be collected from directory and graded.

But before you’re ready with this assignment, you must ensure that it compiles and executes correctly! We’ll turn to that next.

7 Building

In the `~/labs/preliminaries` directory you’ll find these files:

- “`Makefile`”: more on this later.
- “`hello.h`” and “`hello.c`”: files that declare and define the `say_hello_to` API.
- “`main.c`”: contains the definition of the “`main`” function.

7.1 Compiling C programs

Let’s try to compile `main.c` (the “`>`” indicates the shell prompt, and is followed by the command you should enter):

```
> gcc main.c
/tmp/ccbmZeXz.o: In function ‘main’:
main.c:(.text+0x24): undefined reference to ‘say_hello_to’
main.c:(.text+0x30): undefined reference to ‘say_hello_to’
collect2: ld returned 1 exit status
```

Didn’t work. (Why not?)

Try compiling `hello.c`:

```
1 > gcc hello.c
2 /usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../lib64/crt1.o
   : In function ‘_start’:
3 (.text+0x20): undefined reference to ‘main’
4 collect2: ld returned 1 exit status
```

What gives?

Now try:

```
> gcc hello.c main.c
```

Good. No errors. The compiler created an executable for you named “a.out”, by default. Run it:

```
> ./a.out
Hello world!
> ./a.out planet
Hello planet!
```

We can of course rename the executable, but let’s have the compiler put it in the right place for us:

```
> rm a.out
> gcc -o hello hello.c main.c
> ./hello asteroid
Hello asteroid!
```

But what about the multi-stage compilation and linking process discussed in class? This is actually going on behind the scenes already (try invoking gcc with the “-v” flag to see what it’s doing). To build the intermediate object files and link them together in separate steps, do:

```
> gcc -c hello.c
> gcc -c main.c
> gcc -o hello hello.o main.o
```

See how we’re referring to the “.o” files in the third step? Those are the object files that were generated when we invoked gcc with the -c flag, which tells it to stop before the linking step.

If the project being built is complex enough, it may be necessary to separate the final linking step from the creation of a multitude of intermediate object files. Manually invoking the compiler in such situations is a real pain, and definitely not something we’d want to keep doing!

Enter the standard C build tool: **make**

7.2 Use make to run tasks

make is used to automate builds. We’ll try it out next.

```
# We delete the files we previously built.
> rm -f *.o hello

# Run make
> make
gcc -g -Wall -O2 -c -o hello.o hello.c
```

```
gcc -g -Wall -O2 -c -o main.o main.c
gcc -g -Wall -O2 -o hello hello.o main.o
```

There's automation for you! Try it again:

```
> make
make: Nothing to be done for 'all'.
```

make knows that no files have changed since we last build the executable, see. We can update the timestamp of one of the files (using touch) to force a rebuild:

```
> touch hello.c
> make
gcc -g -Wall -O2 -c -o hello.o hello.c
gcc -g -Wall -O2 -o hello hello.o main.o
```

See how it only rebuilds one of the intermediate object files? Pretty nifty. We can also ask it to run a test for us, then clean up generated files:

```
> make test
Running test...
./hello tester
Hello tester!
> make clean
rm -f hello.o main.o hello
```

8 Makefiles: Understand the basics of how the “make” build tool works

make is not magical, of course—it makes use of the provided “Makefile” to determine what action(s) to take, depending on the specified target. For reference, here are the contents of said Makefile. Note the four targets (all, hello, test, and clean), two of which we invoked explicitly before—the first target (all), it turns out, is used by default when we ran the make command with no argument.

```
1 CC      = gcc
2 CFLAGS  = -g -Wall -O2
3 SRCS    = hello.c main.c
4 OBJS    = $(SRCS:.c=.o)
5
6 all:    hello
7
8 hello:  $(OBJS)
9         $(CC) $(CFLAGS) -o hello $(OBJS)
10
11 test:  hello
```

```
12     @echo "Running test..."
13     ./hello tester
14
15 clean:
16     rm -f $(OBJS) hello
```

We'll go over as much of this during our lab demo as we can, but you should check out the GNU make manual for details—at the very least, skim through the Introduction.

9 Finishing up

To earn the points for this lab assignment, you must have saved the changes to `main.c` in the `~/labs/preliminaries-handout` directory. The procedure for doing this is similar in most future labs (except that the filenames might be different, of course) so be sure to read the lab assignment instructions carefully! Procedural takeways:

- Save your files in the correct directory for harvesting by the grading system. The files need to be in that directory before the assignment deadline. And ensure that your code compiles, runs, and produces the expected results.
- Using “make” to build, and often times some variation on “make test” to run a hardcoded test.