Lab 02 - Becoming a C ninja □

Not really… this is basic but useful stuff :-)

Instructor: Lorenzo De Carli, University of Calgary (*lorenzo.decarli@ucalgary.ca*)
Slides by Lorenzo De Carli

In this lab

- Learn to use Make (build tool)
- Learn to use gdb (debugger)
- Let’s build something in C

Compiling programs can get complicated

- This is true particularly when:
  - A program consists of **many different source files**
  - A program consists of **many different deliverables** (executables)
  - You are required to deliver a **clean source folder** (e.g., without executables, output files, etc. for a class homework)
  - Compiling the program requires **specific compiler options** that are easy to forget 😄

Enter make

A build tool

- In a nutshell, a **build tool receives**:
  - A **build target** (typically the executable you want to generate)
  - A list of **prerequisites** to generate it (typically source code files)
  - **Command(s)** to generate it (typically compiler invocation)
- It then checks if the build target exists and is more recent than prerequisites:
  - If yes, **do nothing**
  - If not, run the commands to **build the build target**

How does it work in practice?

- You create a Makefile (literally called “**Makefile**”) in your program directory
- You edit the **Makefile** to specify targets, prerequisites and commands
- You run **make <name of build target>**
- The End!

Basic Makefile example

- Suppose you have a program implemented in three files: **main.c**, **util.h**, and **utils.c**
- Your Makefile will look something like that:

```
main.out: main.c util.h util.c
	gcc -o main.out main.c util.c
```

Prerequisites

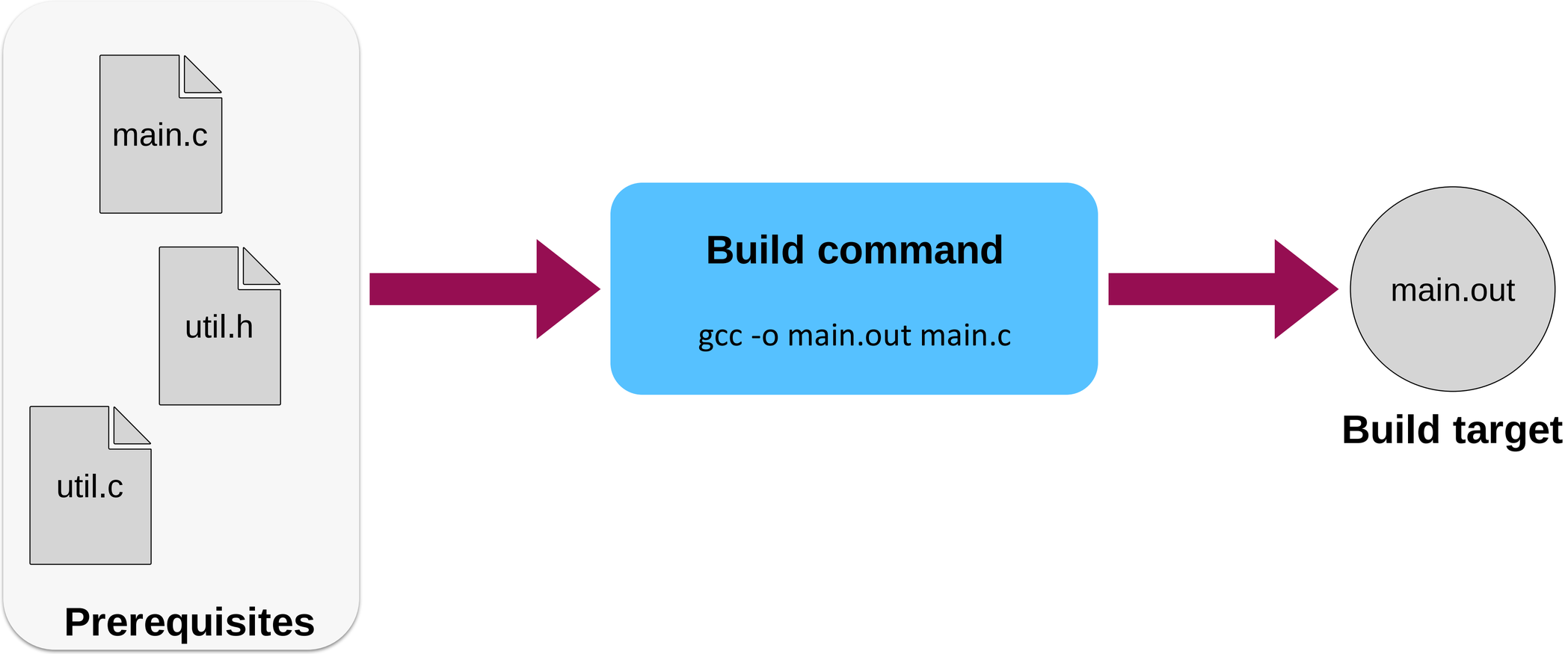
Build command

Note, double TAB

Build target name

Running make: a visualization

Consider the Makefile from the previous slide



Let’s look at a simple example

All examples at *https://github.com/ldklab/ensf461F23/tree/main/lab02/*

Writing better Makefiles

PHONY targets

- What is a **PHONY target**?
  - A target which is **not the name of a file** (no target is actually generated)
- Common **PHONIES**: 🐴 🐴
  - **all:** default target, used to define what should be built by default
  - **clean:** used to remove build targets from folder (cleanup)

Extending the previous Makefile

```
.PHONY: all
all: main.out

main.out: main.c util.h util.c
	gcc -o main.out main.c util.c

.PHONY: clean
clean:
	rm -f main.out
```

There is nothing special about “all”, Make by default executes the **first command** in the Makefile

One more thing…

Some useful special variables

- make enables to define **variables** using the **`$(VAR_NAME)`** syntax
- There are a few **special pre-defined variables** that can be useful, like:
  - **`$@`**: name of the **target being generated**
  - **`$<`**: name of **first prerequisite**
  - **`$^`**: names of all prerequisites
  - **`%`:** not really a variable, wildcard symbol (useful for rules that apply to all the files of a given type)

Let’s look at one example

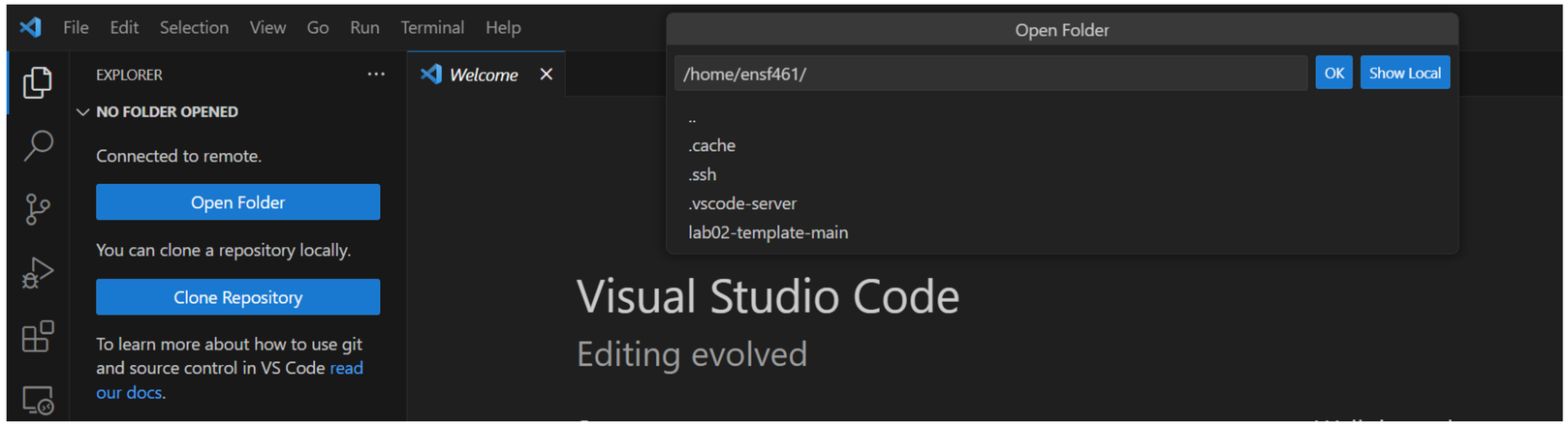
This is actually the Makefile for Lecture #5

```
.PHONY: all
all: example01.out example02.out example03.out

%.out: %.c
	gcc -o $@ $<

.PHONY: clean
clean:
	rm -f *.out
```

Getting code for exercises

- The code for Lab 02 has been uploaded to D2L in the lab02-template.zip
- Downloading the zip to your computer is no problem, but how do we copy the file over to the VM and unzip it?

Accessing a Folder in VS Code

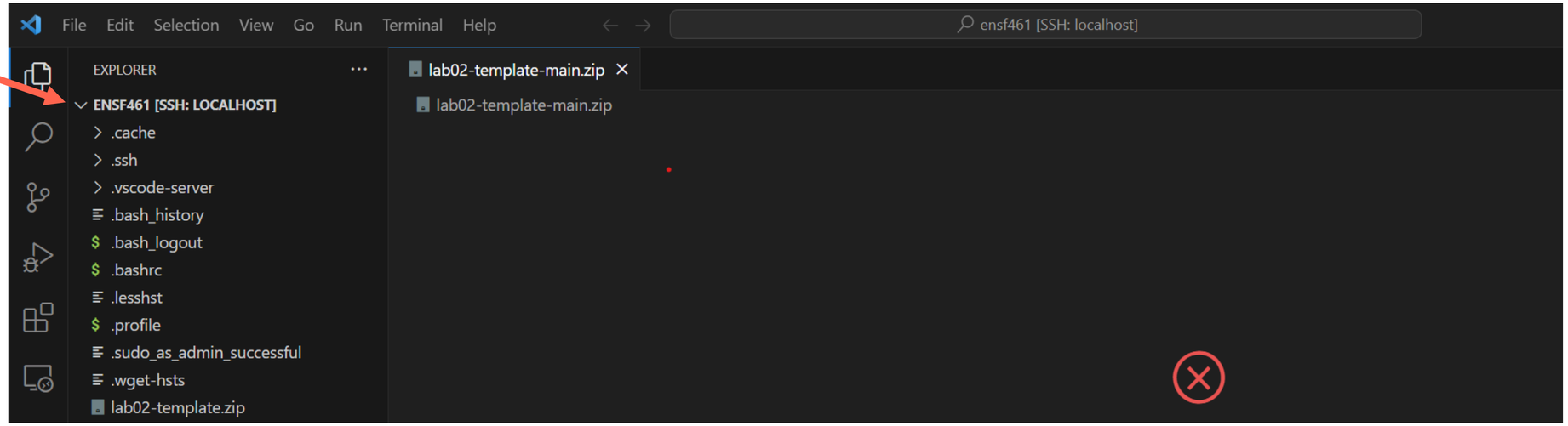
- When you are connected to the VM (using SSH-Remote) simply click on the open folder button
- 

- In the prompt open your home folder (should be something like */home/lorezno*) if you are not using the *ensf461* username
- You may be asked for your password before it shows you the folder location prompt

Accessing a Folder in VS Code

- Enter your password into the prompt, and you'll have access to the folder in Visual Studio Code

You may have to expand the view to see your files



- Now you can drag and drop the Zip file (or any other files) into the folder to upload it to the VM!

How to Unzip a file

- Once you have the zip file in your VM, you first need to install the “zip” utility

```
ensf461@ensf461:~$ sudo apt install zip
```

- And then you can unzip the file with the unzip command, for more information on how to zip/unzip files look at the man pages for *zip* and *unzip*

```
ensf461@ensf461:~$ unzip lab02-template.zip
Archive:  lab02-template.zip
ef37340dfeab7ad1db357c035a678464855aabc9
   creating: lab02-template-main/
   creating: lab02-template-main/exercise01/
  inflating: lab02-template-main/exercise01/README.MD
  inflating: lab02-template-main/exercise01/gen_in_range.c
  inflating: lab02-template-main/exercise01/gen_numbers.c
  inflating: lab02-template-main/exercise01/select_arg.c
  inflating: lab02-template-main/exercise01/utils.c
  inflating: lab02-template-main/exercise01/utils.h
   creating: lab02-template-main/exercise02/
```

Submitting Exercises

- After unzipping rename the *lab02-template-main* folder to ***lab02-GroupXX***, replacing XX with your 2 digit group number
- All of the code for the three exercises is now in this folder!
- We'll be using a D2L Dropbox to submit your code and images for Lab 2, at the end of the lab please create a single zip file from *lab02-GroupXX* folder (hint: look at the zip man pages) with whatever you've managed to finish and upload it to your **D2L Dropbox before 5 PM today** (**3 Pts**)
- Don't worry if you're not done everything by the end of the lab, this first submission is your "in-lab" progress, you will have until **5 PM on the day before the next lab** to submit your complete solution to Dropbox (only the last submission to Dropbox will be graded for the following three exercises)

Code for exercise 1

- You can find the code for which you'll need to implement a Makefile in the "Lab 02" folder you unzipped, under **exercise01/**

This brings us to Exercise 1

- Note: in this (and other) exercises, you may be asked to **implement functionality not discussed in class**. It is your task to figure out how to do it
- You may ask classmates, instructors, TAs, online resources, etc.
- In some cases, you may need to look at the **book/slides** for answers

Exercise #1

- In your assignment repository, you will find a suite of C programs. Write a Makefile that compile them using gcc (refer to README.MD for more information). **Further requirements:**
- Define “all” as default Makefile target that builds all other targes (e.g., running “make” should run the “all” target, and in turn build everything else. Also, define “clean” as target to remove all generated executable files. Finally, use a variable to store the compiler command (by default “gcc”), in such a way that changing the value of that variable will change the compiler used throughout the Makefile **(0.5 pts)**
- Use, where possible, wildcards and/or target placeholders ($@, $<, $^) rather than manually writing down prerequisites and targets **(0.5 pts)**
- Test if make is being run on Linux before executing the “all” target. If the OS is not Linux, print “Sorry, I prefer Linux” and do nothing. If the OS is Linux, execute the targets as normal **(0.5 pts)**
- Define a “test” target which compiles each executable, runs it with valid command line parameters, and checks that the return value of each executable is 0 (anything is fine as long as **some** kind of error is returned if this condition is not met) **(0.5 pts)**

Let’s move on to the next topic

Debugging

- **What is a “debugger”?**
- A tool to **instrument program execution** so the following becomes possible:
  - Execute the program **step-by-step** (individual lines of code)
  - **Pause** the program execution at arbitrary points
  - **Print the value** of program variables
  - … and more!

Before we get started…

…you'll need to install a debugger in your VM

- Run the following:

  sudo apt update
  sudo apt install gdb

- Note: you can find the code for the program I will show you in the assignment GitHub repository, under **example02/**

Let’s start with a simple program

test.c U × | Makefile U

test.c > main(int, char **)

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {

    if ( argc < 2 ) {
        fprintf(stderr, "Usage: test.out <loop iterations>\n\n");
        return -1;
    }

    int i = atoi(argv[1]);
    while ( i < 10 ) {
        printf("%d\n", i);
        i++;
    }

    return 0;
}
```

test.c U | Makefile U ×

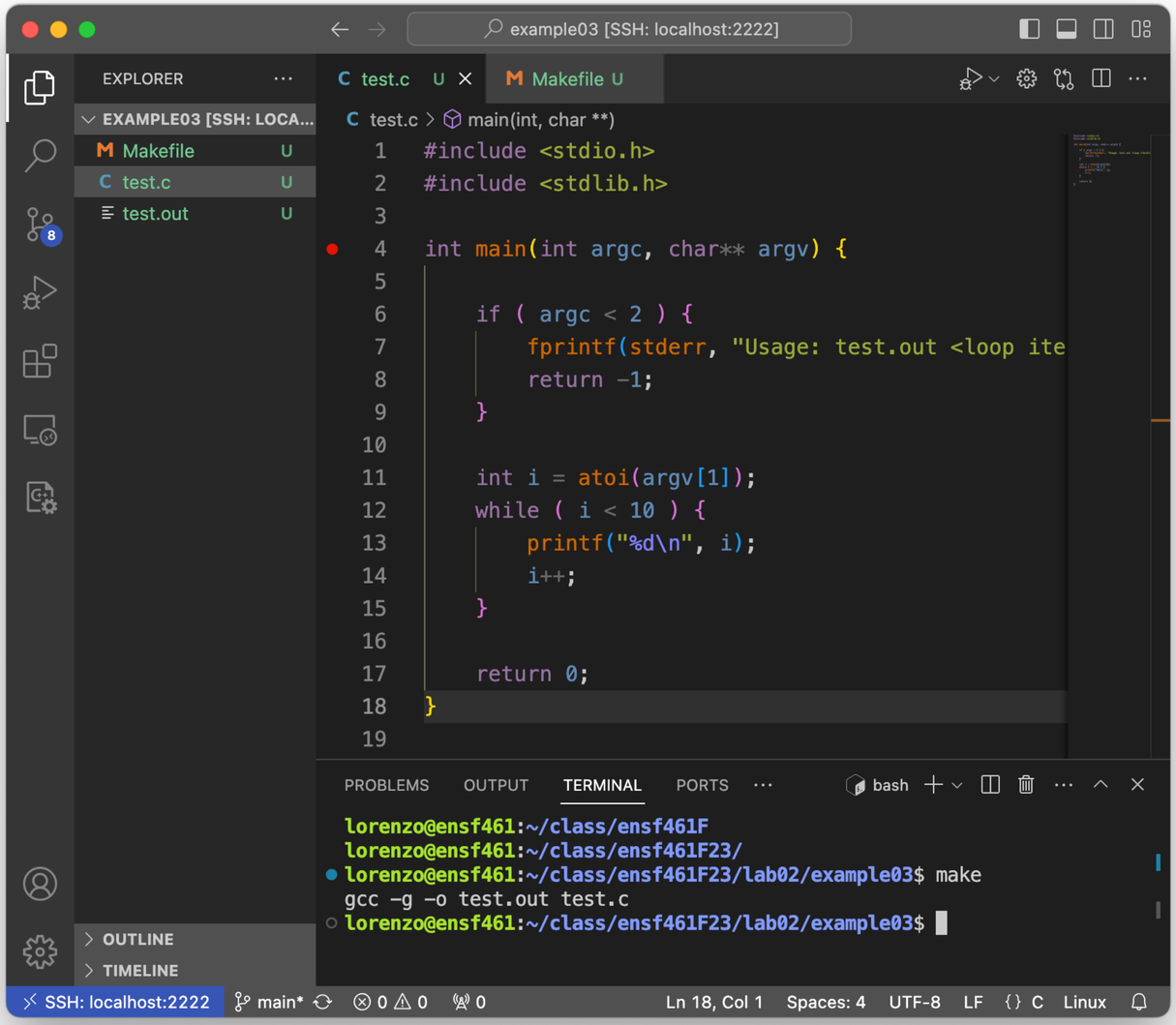
Makefile

```makefile
.PHONY: all
all: test.out

%.out: %.c
    gcc -g -o $@ $<

.PHONY: clean
clean:
    rm -f *.out
```

Note the “-g”: it is important!!

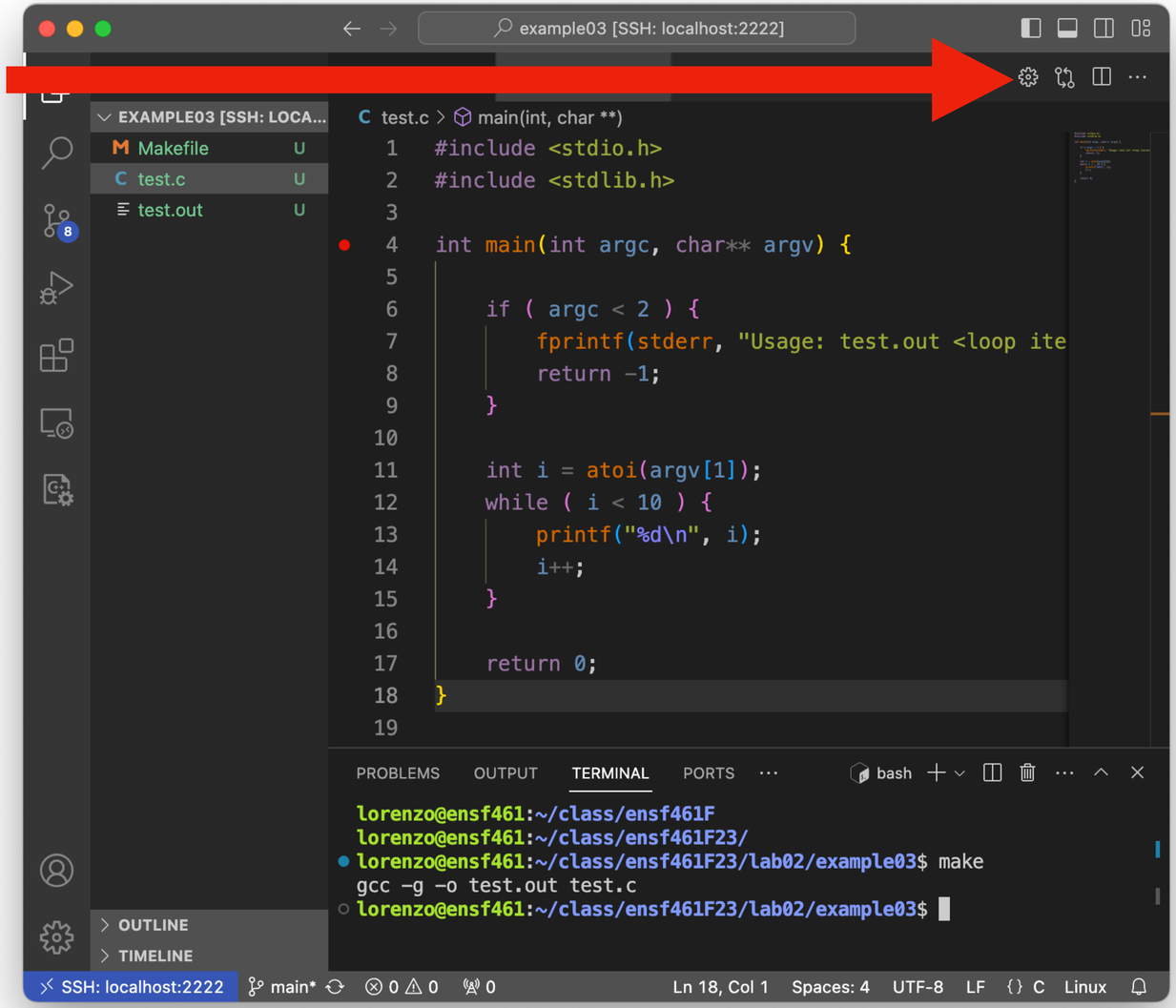
First: build the program

- No need to do anything fancy… pop up a terminal and run “**make**” (you can use **vscode’s terminal**)

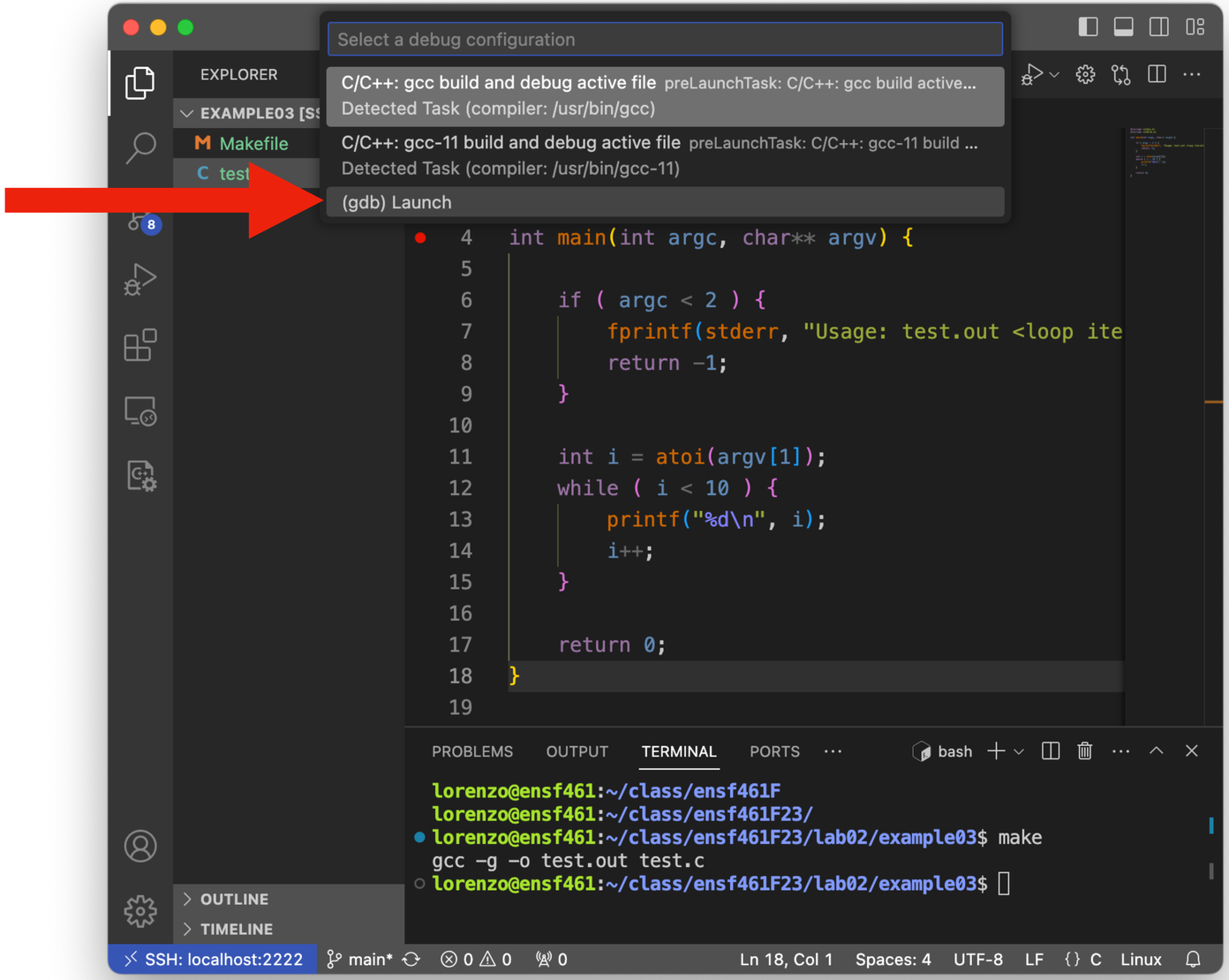


Next: create a debug configuration

Click on the **gear icon** on vscode's bar



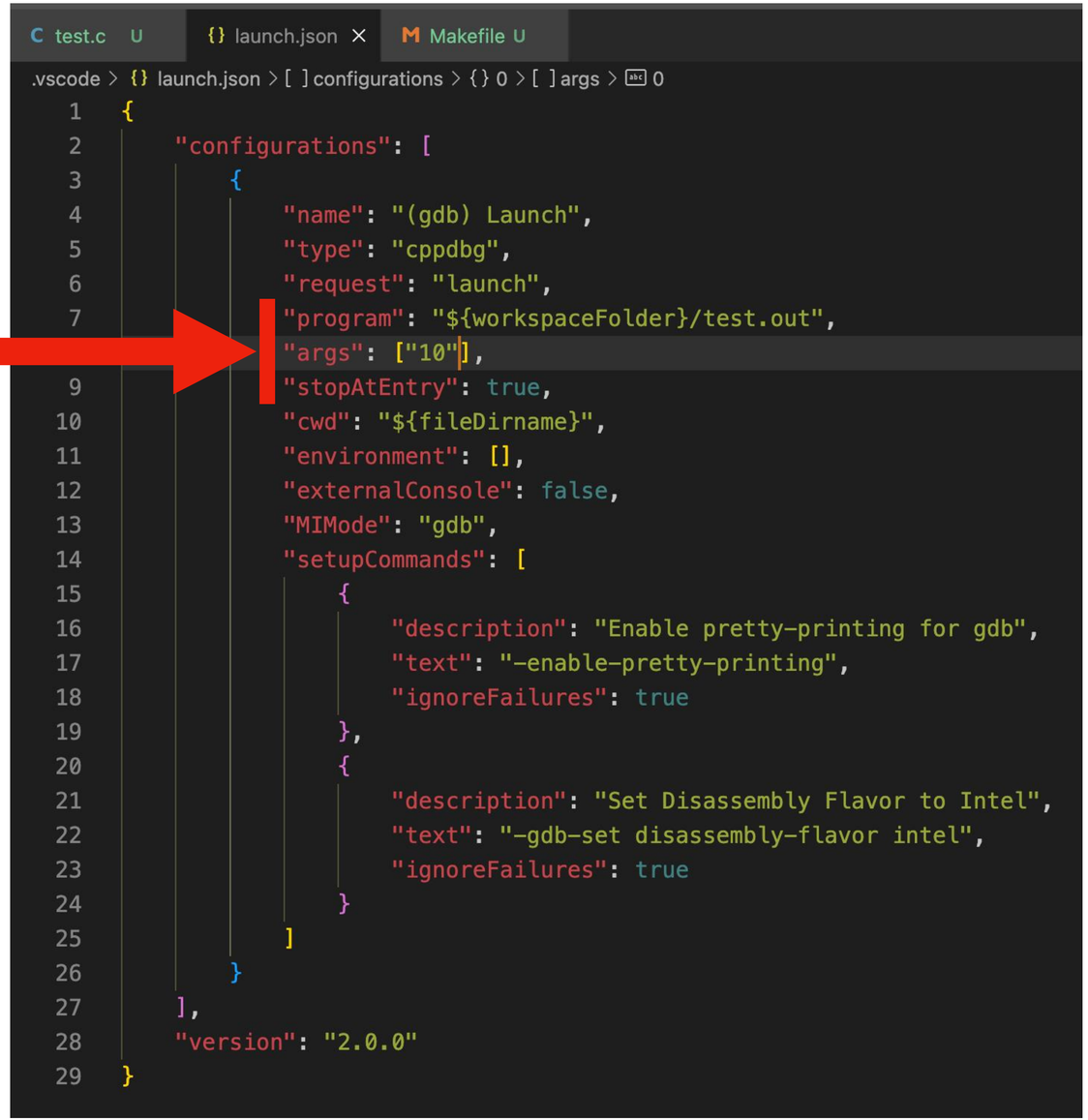
Click on “gdb - launch”



vscode will create a template for you…

…but you need to edit it!

You need to set the program name (“program”),
the command line arguments (“args”),
and you may want to set “stopAtEntry” to **true**



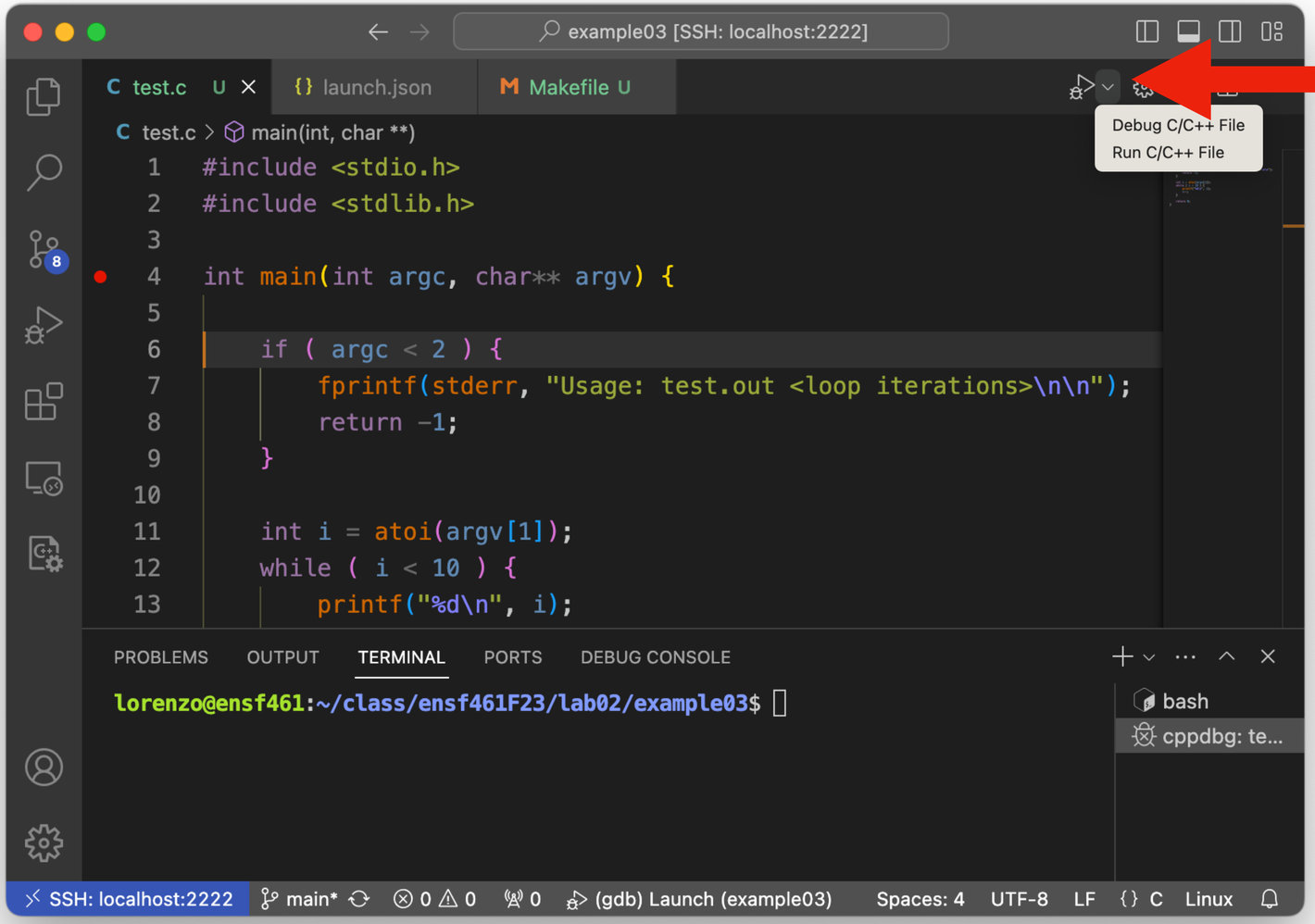

```
{
    "configurations": [
        {
            "name": "(gdb) Launch",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}/test.out",
            "args": ["10"],
            "stopAtEntry": true,
            "cwd": "${fileDirname}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                },
                {
                    "description": "Set Disassembly Flavor to Intel",
                    "text": "-gdb-set disassembly-flavor intel",
                    "ignoreFailures": true
                }
            ]
        }
    ],
    "version": "2.0.0"
}
```

Sometimes vscode will refuse to create a template

If so, do the following

- Create a folder named "**.vscode**" in your project directory
- Create a file named "**launch.json**" and copy the content of the file at: https://github.com/ldklab/ensf461F23/blob/main/lab02/example_launch.json
- Edit the file as discussed in the previous slide

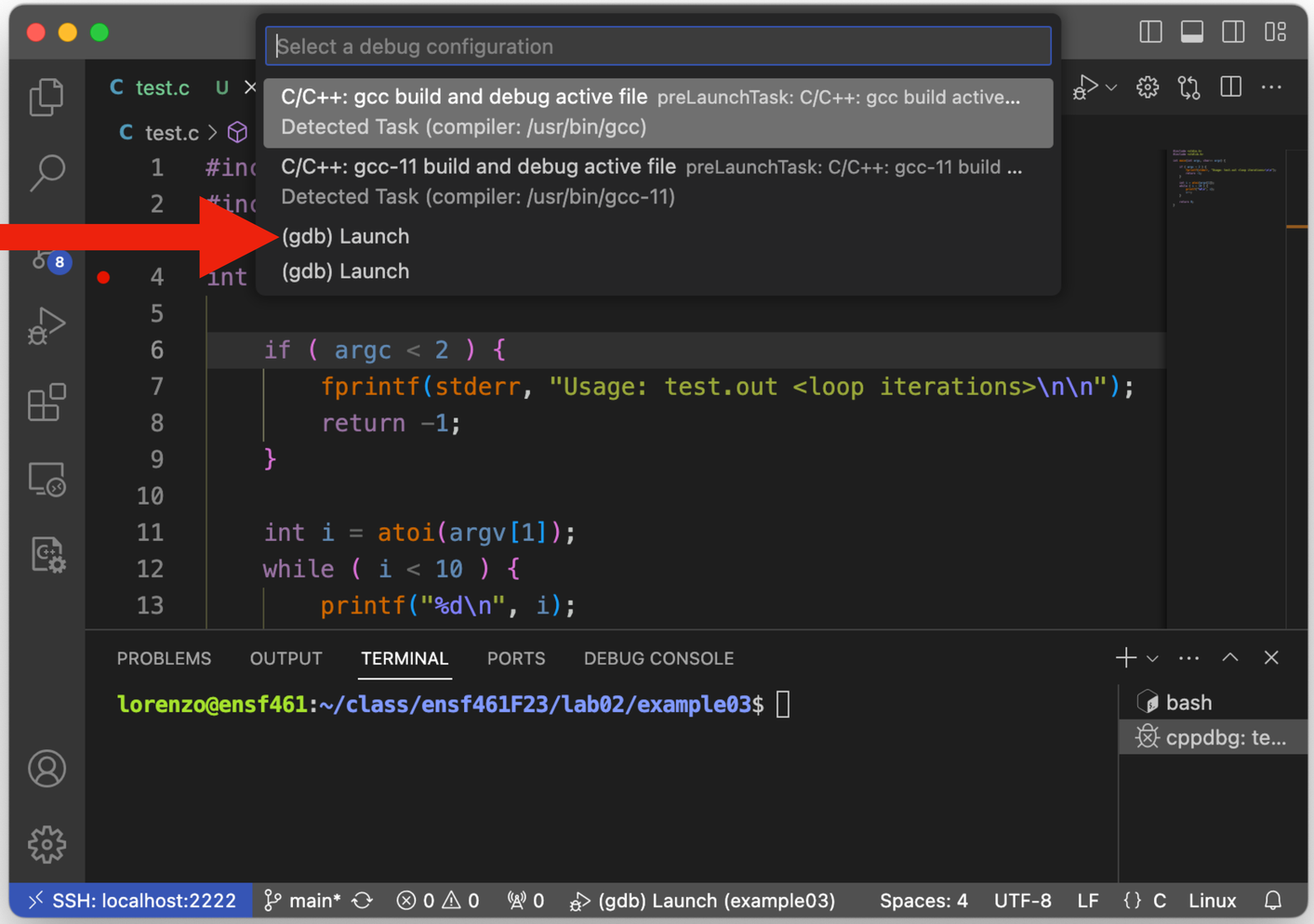
Back to test.c



Click on the run/debug icon, select “Debug C/++ file”

One more step

Choose “(gdb) Launch”



Et voila!

Program variables

Execution controls

Current location

example03 [SSH: localhost:2222]

(gdb) Launc

VARIABLES

Locals

i: 0

argc: 2

argv: 0x7fffffffe0f8

Registers

WATCH

CALL STACK Paused o...

main(int argc, char **

BREAKPOINTS

All C++ Excepti...

test.c 6

test.c U | launch.json | Makefile U

test.c > main(int, char **)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char** argv) {
5
6      if ( argc < 2 ) {
7          fprintf(stderr, "Usage: test.out <loop iterations>\n\n");
8          return -1;
9      }
10
11     int i = atoi(argv[1]);
12     while ( i < 10 ) {
13         printf("%d\n", i);
14         i++;
15     }
16
17     return 0;
18 }
19
```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

SSH: localhost:2222 main* 0 0 0 (gdb) Launch (example03)

Now, let’s switch to vscode and see a debugger in action

Exercise #2

- Following the previous instructions, create a suitable launch configuration under **exercise02/.vscode/launch.json** and include it within your zip file submission **(1 pt)**
- Capture a screenshot of Visual Studio code running a debug session on the program we just saw, and save it as **exercise01/vscode.jpg** (PNG or BMP is also fine) and include it within your zip file submission **(1 pt)**

Exercise #3

C/C++ practice

- Under “exercise03” in the Lab 02 assignment you will find a C program to complete
- The C program receives a CSV file as input, with an arbitrary number of lines. Each line contains a list of comma-separated integers
- The program should be executed as such:
  parsecsv.out numbers.csv output.csv
- Output should contains, for each line, the mean and standard deviation of the corresponding line in the input file

More on exercise #3

- The repository already contains the Makefile, and all necessary source code files
- The source code files have various TODOs (unimplemented functions). To successfully complete the exercise, fill in every TODO.
- You will see that the code uses a **linked list** with **dynamic memory allocation** for processing the file
- There are 6 TODOs, each worth **0.5 points**
- Let’s take a look at the code

Submission Reminder

- You need to submit a single zip file to Dropbox twice:
  - One “in-lab” submission is due **today before 5 PM**, including all of the work you were able to complete in the lab, in a single zip called *Lab02-GroupXX.zip*, this submission is worth **3 Pts**
  - You will have until **5 PM on the day before the next lab** to submit your completed lab solution, in a single zip file with the same *Lab02-GroupXX.zip* name, this submission will be graded for the remaining **7 Pts.**
- You may submit your solution to your Dropbox as many times as you’d like, but only the LAST submitted zip file will be graded for the three exercises (please double-check that the zip file opens properly!)