

CMPT 125 D100 LAB 2

TA

TOPICS FOR TODAY

Makefile

Header files

Compile C projects with .h and .c files

Makefile

- What is a makefile?
 - The makefile is a text file that contains the recipe for building your program. It usually resides in the same directory as the sources, and it is usually called Makefile.
- Why it is needed?
 - Provide a way for separate compilation
 - Describe the dependencies among project file
 - Manage Several source files and compile them quickly with a single Command Line.
 - Make compilation time shorter
 - Make checks timestamps to see what has changed and re-builds just what you need, without wasting time re-building other files.

Running “make file”

- Naming:
 - makefile or Makefile are standard, other name can be also used
- Running:
 - make : This will find the make file in the directory and build first target it finds there.
 - make -f mymakefile: will run make files called “mymakefile”. in the other word if you have several makefiles you can run the one you want with make -f *name-of-that-file*
 - make name-of-a-target: will run the rule for a specific target

Running “make file” (for students using Windows)

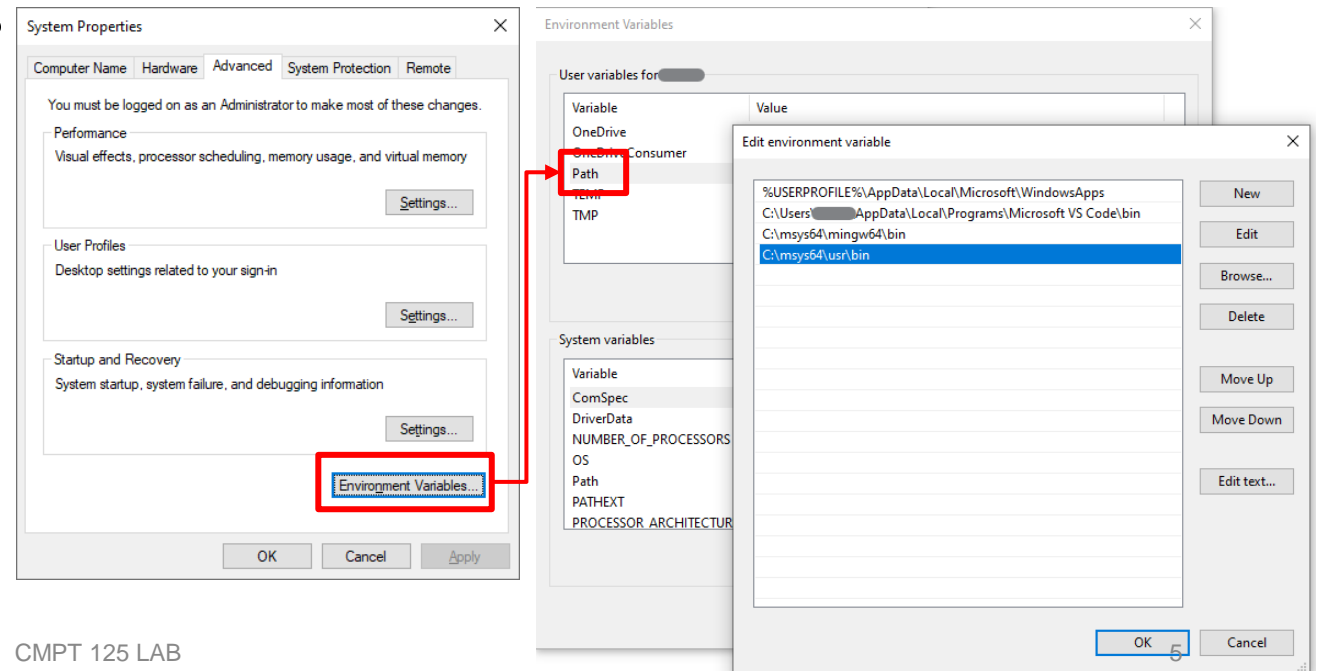
- make is not pre-installed in Windows, but if you have MinGW-w64 installed, you have access to both gcc and make in your computer
- press the **Win** key

look for **Environmental Variables**

choose **Path** and click **Edit...**

add a new path as

`C:\msys64\usr\bin`



What is in makefile - Rule



Makefile — Edited

```
test_assignment: test_assignment.c assignment.c
    gcc -Wall -std=c99 -o test_assignment test_assignment.c assignment.c
```

- A rule tells make two things: when the targets are out of date, and how to update them when necessary.
- A target is out of date if it does not exist or if it is older than any of the dependencies (by comparison of last-modification times). The idea is that the contents of the target file are computed based on information in the dependencies, so if any of the dependencies changes, the contents of the existing target file are no longer necessarily valid.
- In general, a rule looks like this:

```
targets : dependencies
    command
    ...
```

What is in makefile - Explicit Rules

Explicit rules specify the instructions that make must follow when it builds specific targets.

What is in makefile - Implicit Rules

Implicit rules tell make how to use customary techniques so that you do not have to specify them in detail when you want to use them.

```
%o : %.c
$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<
```

For example, there is an implicit rule for C compilation. File names determine which implicit rules are run. For example, C compilation typically takes a '.c' file and makes a '.o' file. So make applies the implicit rule for C compilation when it sees this combination of file name endings.

The above defines a rule that can make any file 'x.o' from 'x.c'. The '%' can match any nonempty substring. The command uses the automatic variables '\$@' and '\$<' to substitute the names of the target file and the source file in each case where the rule applies.

What is in makefile - Variable

The commands in built-in implicit rules make liberal use of certain predefined variables. You can alter these variables in the makefile

CC: Program for compiling C programs; default `cc'.

in a makefile, it would be written as:

```
CC=cc
```

Variables whose values are additional arguments for the programs. The default values for all of these is the empty string, unless otherwise noted.

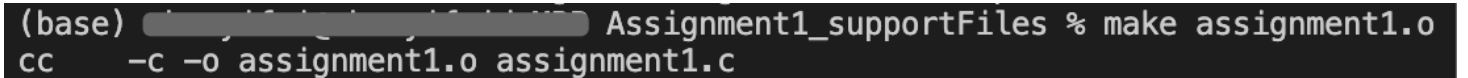
CFLAGS: Extra flags to give to the C compiler.

CPPFLAGS: Extra flags to give to the C preprocessor and programs that use it (the C and Fortran compilers).

Makefile example

Try to run the following command:

```
make assignment1.o
```



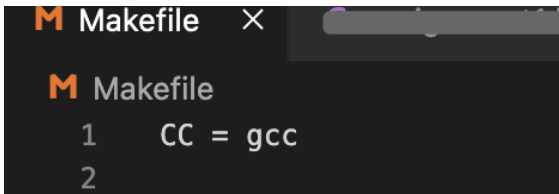
```
(base) Assignment1_supportFiles % make assignment1.o
cc -c -o assignment1.o assignment1.c
```

It would run the command based on the pre-defined implicit rule:

```
cc -c -o assignment1.o assignment1.c
```

If you change CC to gcc in the makefile:

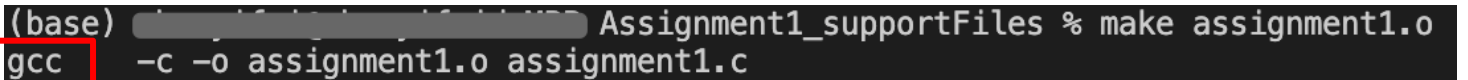
```
CC=gcc
```



```
Makefile
1 CC = gcc
2
```

Try to run the following command:

```
make assignment1.o
```



```
(base) Assignment1_supportFiles % make assignment1.o
gcc -c -o assignment1.o assignment1.c
```

It would run the command based on the pre-defined implicit rule, with a replace of cc to gcc:

```
gcc -c -o assignment1.o assignment1.c
```

Makefile example

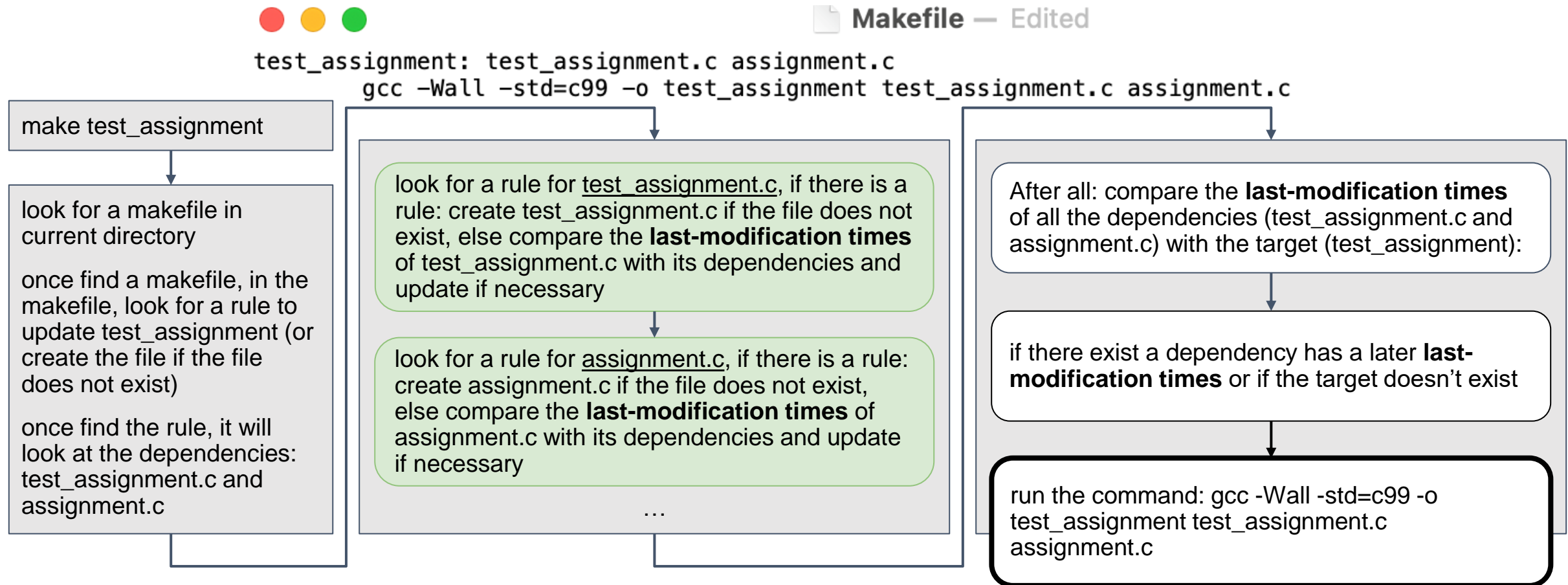


Makefile — Edited

```
test_assignment: test_assignment.c assignment.c
    gcc -Wall -std=c99 -o test_assignment test_assignment.c assignment.c
```

- To use it, run the command in the shell/terminal (make sure your current working directory has all the files):
 - make or make -f Makefile or make test_assignment or make -f Makefile test_assignment
- Then the executable with the name test_assignment will be created. You can run it with the command:
 - ./test_assignment
- You have to run this command everytime you modify your code to get the updated executable

Makefile example - Explicit Rules



Makefile example: make test_assignment



Makefile — Edited

```
test_assignment: test_assignment.c assignment.c
    gcc -Wall -std=c99 -o test_assignment test_assignment.c assignment.c
```

gcc:

C program compiler

-Wall:

enables all compiler's warning messages. This option should always be used, in order to generate better code.

-std=c99:

The C programming language has many versions. This flag lets the compiler know that we're using the standard version C99 (released in 1999)

-o test_assignment:

set target file name

test_assignment.c assignment.c:

the source files that target file depends on

Makefile example: make test_assignment



Makefile — Edited

```
test_assignment: test_assignment.c assignment.c
```

```
    gcc -Wall -std=c99 -o test_assignment test_assignment.c assignment.c
```

```
gcc -Wall -std=c99 -o test_assignment test_assignment.c assignment.c
```

is the same as:

```
gcc -Wall -std=c99 -c test_assignment.c -o test_assignment1.o
```

turns test_assignment.c into assignment.o

```
gcc -Wall -std=c99 -c assignment.c -o assignment1.o
```

turns assignment.c into assignment.o

```
gcc -Wall -std=c99 -o test_assignment assignment.o test_assignment.o
```

Linking: takes assignment.o and test_assignment.o into “test_assignment”

-c: use the -c flag to tell the compiler to skip the link stage, in the case of compiling several files separately then combine

Makefile example: linkTestAssignment

Tried to write a new makefile with name “Makefile1” with rules that will compile the code into an executable called “linkTestAssignment” in the following way:

```
gcc -Wall -std=c99 -c -o test_assignment1.o test_assignment1.c
gcc -Wall -std=c99 -c -o assignment1.o assignment1.c
gcc -Wall -std=c99 -o linkTestAssignment test_assignment1.o assignment1.o
```

-c: use the -c flag to tell the compiler to skip the link stage, in the case of compiling several files separately then combine

Makefile example: linkTestAssignment

Tried to write a new makefile with name “Makefile1” with rules that will compile the code into an executable called “linkTestAssignment” in the following way:

```
gcc -Wall -std=c99 -c -o test_assignment1.o test_assignment1.c
gcc -Wall -std=c99 -c -o assignment1.o assignment1.c
gcc -Wall -std=c99 -o linkTestAssignment test_assignment1.o assignment1.o
```

-c: use the -c flag to tell the compiler to skip the link stage, in the case of compiling several files separately then combine

```
M Makefile1
1  assignment1.o : assignment1.c
2      gcc -Wall -std=c99 -c -o assignment1.o assignment1.c
3
4  test_assignment1.o : test_assignment1.c
5      gcc -Wall -std=c99 -c -o test_assignment1.o test_assignment1.c
6
7  linkTestAssignment: test_assignment1.o assignment1.o
8      gcc -Wall -std=c99 -o linkTestAssignment test_assignment1.o assignment1.o
```

```
● (base) Assignment1_supportFiles % make -f Makefile1 linkTestAssignment
gcc -Wall -std=c99 -c -o test_assignment1.o test_assignment1.c
gcc -Wall -std=c99 -c -o assignment1.o assignment1.c
gcc -Wall -std=c99 -o linkTestAssignment test_assignment1.o assignment1.o
```


Makefile example: linkTestAssignment

Tried to write a new makefile with name “Makefile1” with rules that will compile the code into an executable called “linkTestAssignment” in the following way:

```
gcc -Wall -std=c99 -c -o test_assignment1.o test_assignment1.c
gcc -Wall -std=c99 -c -o assignment1.o assignment1.c
gcc -Wall -std=c99 -o linkTestAssignment test_assignment1.o assignment1.o
```

-c: use the -c flag to tell the compiler to skip the link stage, in the case of compiling several files separately then combine

```
M Makefile1
1  %.o : %.c
2      gcc -Wall -std=c99 -c -o $@ $<
3
4  linkTestAssignment: test_assignment1.o assignment1.o
5      gcc -Wall -std=c99 -o linkTestAssignment test_assignment1.o assignment1.o
● (base) Assignment1_supportFiles % make -f Makefile1 linkTestAssignment
gcc -Wall -std=c99 -c -o test_assignment1.o test_assignment1.c
gcc -Wall -std=c99 -c -o assignment1.o assignment1.c
gcc -Wall -std=c99 -o linkTestAssignment test_assignment1.o assignment1.o
```

Take the advantage of the automatic variables and written as an implicit rule

Makefile example: linkTestAssignment

Tried to write a new makefile with name “Makefile1” with rules that will compile the code into an executable called “linkTestAssignment” in the following way:

```
gcc -Wall -std=c99 -c -o test_assignment1.o test_assignment1.c
gcc -Wall -std=c99 -c -o assignment1.o assignment1.c
gcc -Wall -std=c99 -o linkTestAssignment test_assignment1.o assignment1.o
```

-c: use the -c flag to tell the compiler to skip the link stage, in the case of compiling several files separately then combine

M Makefile1

```
1  CC = gcc
2  CFLAGS = -Wall -std=c99
3
4  linkTestAssignment: test_assignment1.o assignment1.o
5  |      gcc -Wall -std=c99 -o linkTestAssignment test_assignment1.o assignment1.o
```

Take the advantage of the predefined implicit rule and redefined the variables

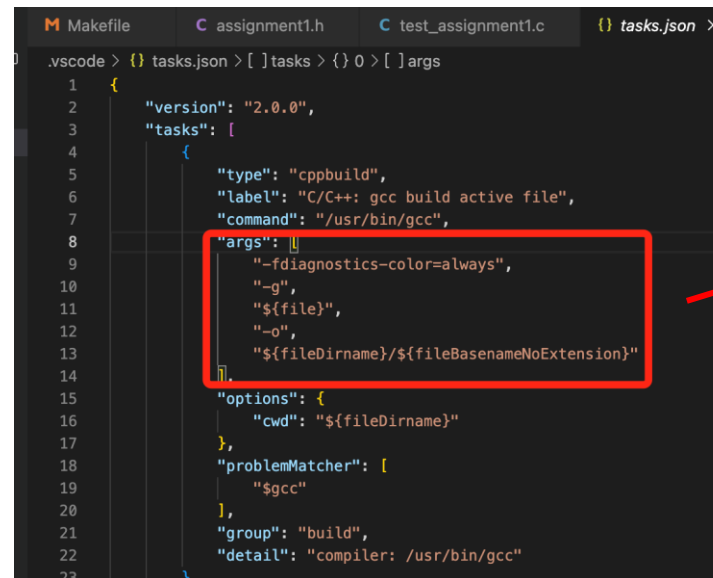
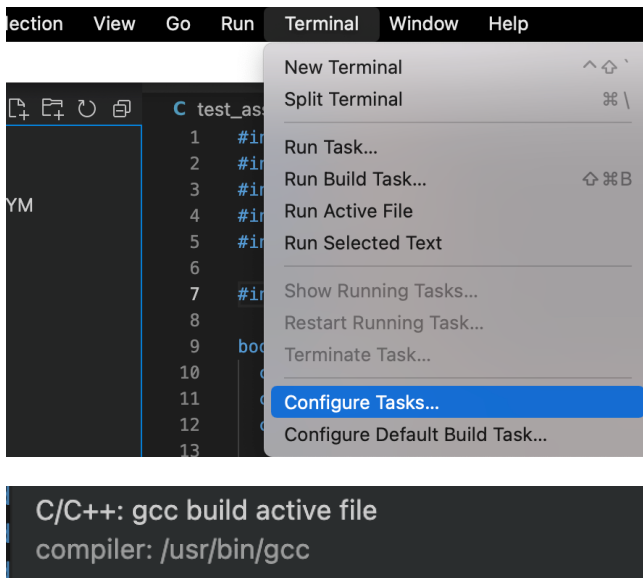
```
● (base) Assignment1_supportFiles % make -f Makefile1 linkTestAssignment
gcc -Wall -std=c99 -c -o test_assignment1.o test_assignment1.c
gcc -Wall -std=c99 -c -o assignment1.o assignment1.c
gcc -Wall -std=c99 -o linkTestAssignment test_assignment1.o assignment1.o
```

an alternative way to compile assignment:

Modify the task.json file by adding assignment1.c in the args attribute

Make sure your active editing window is showing the test_assignment1.c file (where the main() is), then: Terminal -> Run Build Task

Then: Run -> Start Debugging / Run Without Debugging



```
"args": [  
    "-Wall",  
    "-std=c99",  
    "-g",  
    "${file}",  
    "${fileDirname}/assignment1.c",  
    "-o",  
    "${fileDirname}/${fileBasenameNoExtension}"  
],
```

For Windows, the last line is:
\${fileDirname}\\\${fileBasenameNoExtension}.exe

You don't need to modify the last line!!!

Header files

A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files.

.c files – code that implements the functions declared in the header file

Macro Definitions:

```
#ifndef ASSIGNMENT2_H
#define ASSIGNMENT2_H
// header file content
#endif
```

it's used to avoid the same header file being included multiple times within the same C file, which would lead to multiple-definition errors.

Header files

There are two types of header files:

Pre-existing header files:

Files which are already available in C/C++ compiler we just need to import them. Include pre-existing header files by using the syntax:

```
#include <filename.h>
```

User-defined header files:

These files are defined by the user and can be imported using “#include”. Include User-defined header files by using the syntax:

```
#include "filename.h"
```

Exercise for you

Download the following files from Canvas > Lab02:
main.c, minmax.c, minmax.h

Take a look at each file to see how the minmax.h header file is included, and how the function definitions are separated from their function header

Write a Makefile with a rule that will compile the code into an executable called findMinMax

Header files

Commonly used Pre-existing header files:

<stdio.h> : have many standard library functions for file input and output

<math.h> : support are mathematical related functions in c

<time.h>: interact with system time

<string.h>: support string handling function