南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Advanced Programming

Lab 3 Common Commands in Linux, Makefile

于仕琪，王大兴，廖琪梅, 王薇

topics

- 1. Commands in Linux
  - commands: directory, file .etc.
  - lists of commands, pipelines
- 2. Makefile
  - file: makefile/Makefile
  - command: make, make clean
- 3. Practices
  - commands, makefile
  - branch, loop

1. Commands in Linux

Linux is a family of open-source Unix operating systems based on the Linux Kernel.
There are some popular distributions
such as Ubuntu, Fedora, Debian, openSUSE, and Red Hat.

A Linux command is a program or utility that runs on the Command Line Interface – a console that interacts with the system via texts and processes.

Linux command's general syntax looks like:

CommandName [option(s)] [parameter(s)]

- **CommandName** is the rule that you want to perform.
- **Option** or **flag** modifies a command's operation. To invoke it, use hyphens (-) or double hyphens (--).
- **Parameter** or **argument** specifies any necessary information for the command.

1. Commands in Linux

1.1 Linux directory and file commands:

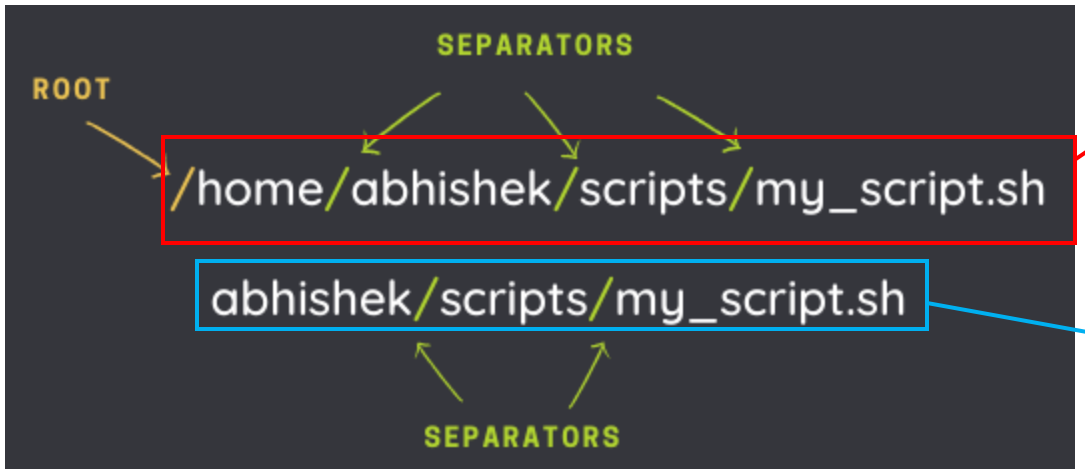
Command	Meaning
pwd	**P**rint the name of current/**w**orking **d**irectory.
cd <directory name>	**C**hange the current **d**irectory.
ls	**L**ist of content of a directory.
mkdir <directory name>	**M**ake a new **dir**ectory under any directory.
rmdir <directory name>	**Rem**ove **dir**ectories without files.
cat <file name>	Display content of the file.
rm <file name>	**Rem**ove a file.
cp <source> <dest>	**Cop**y a file or files to another
mv<source><dest>	**Mov**e a file or files to another directory
cat/tail/head, less/more, nano/vim, file, whereis, echo	

https://www.javatpoint.com/linux-commands

Absolute path and relative path

A **path** is how you refer to files and directories. It gives the location of a file or directory in the Linux directory structure. It is composed of a **name** and **slash** syntax.

If the path starts with slash "/", the first slash denotes root. The rest of the slashes in the path are just separators.



The **absolute path** always starts from the root directory (/).

A **relative path** starts from the current directory.

Two special relative paths:

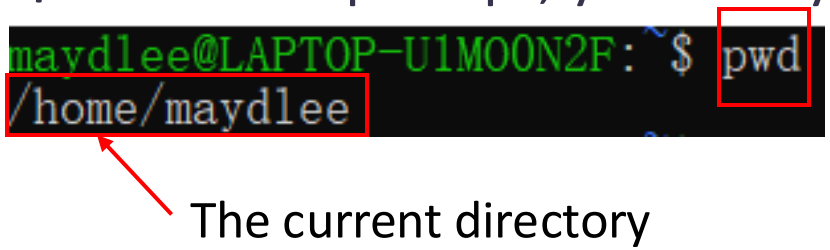
- **. (single dot) denotes the current directory in the path.**
- **.. (two dots) denotes the parent directory, i.e., one level above.**

pwd command

Use the **pwd** command to display the current working directory you are in.

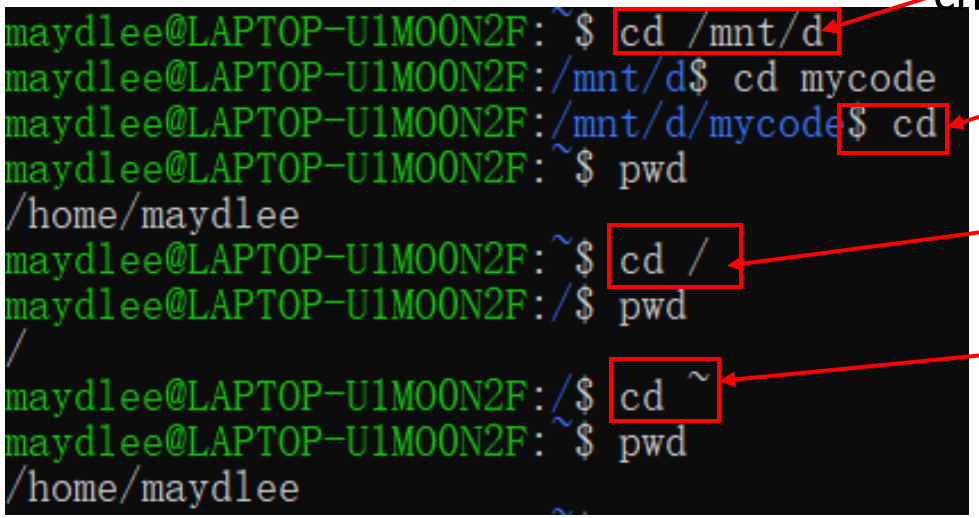
Start Ubuntu, you will see:

$ or **#** is the prompt, you can type command now.



cd command

To navigate through the Linux files and directories, use the **cd** command.



change the directory to the root directory of d drive

change the directory to the home directory

change the directory to the root directory

change the directory to the home directory

Here are some shortcuts to help you navigate:

- **cd ~[username]** goes to another user's home directory.
- **cd ..** moves one directory up.
- **cd -** moves to your previous directory.
- **cd** without an option will take you to the home folder.

ls command

The **ls** command lists files and directories within a system. Running it without a flag or parameter will show the current working directory's content.

```
maydlee@LAPTOP-U1MOON2F:/mnt/d$ cd CMake
maydlee@LAPTOP-U1MOON2F:/mnt/d/CMake$ ls
CMakeCache.txt  CMakeLists.txt  Demo2  HelloWorld.cpp  cmake_install.cmake
CMakeFiles      Demo1           Demo3  Makefile        hello.exe
maydlee@LAPTOP-U1MOON2F:/mnt/d/CMake$ ls Demo1
CMakeCache.txt  CMakeFiles  CMakeLists.txt  Makefile  cmake_install.cmake  hello.exe  main.cpp
maydlee@LAPTOP-U1MOON2F:/mnt/d/CMake$ ls -l
total 188
-rwxrwxrwx 1 maydlee maydlee  14456 Oct 25  2020 CMakeCache.txt
drwxrwxrwx 1 maydlee maydlee   4096 Jun  4  2021 CMakeFiles
-rwxrwxrwx 1 maydlee maydlee     99 Oct 25  2020 CMakeLists.txt
drwxrwxrwx 1 maydlee maydlee   4096 Feb 22  2021 Demo1
drwxrwxrwx 1 maydlee maydlee   4096 Feb 22  2021 Demo2
drwxrwxrwx 1 maydlee maydlee   4096 Jun  4  2021 Demo3
-rwxrwxrwx 1 maydlee maydlee    114 Oct 30  2020 HelloWorld.cpp
-rwxrwxrwx 1 maydlee maydlee   4783 Oct 25  2020 Makefile
-rwxrwxrwx 1 maydlee maydlee   1341 Oct 25  2020 cmake_install.cmake
-rwxrwxrwx 1 maydlee maydlee 160805 Oct 30  2020 hello.exe
```

List subdirectory and files in the current directory

List subdirectory and files in the Demo1 directory

List detail information of subdirectory and files in the current directory

Here are some options you can use with the **ls** command:

- **ls -R** lists all the files in the subdirectories.
- **ls -a** shows hidden files in addition to the visible ones.
- **ls -l (or ll)** shows detail information of subdirectory and files

mkdir command

Use the **mkdir** command to create one or multiple directories at once.

```
maydlee@LAPTOP-U1MOON2F:/mnt/d$ cd examples
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ ls
CMakeLists.txt  a.exe  a.exe.stackdump  main.cpp  matoperation.cpp  matoperation.hpp
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ mkdir demo1 demo2
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ ls
CMakeLists.txt  a.exe  a.exe.stackdump  demo1  demo2  main.cpp  matoperation.cpp  matoperation.hpp
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ mkdir demo1/exercise_demo
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ ls
CMakeLists.txt  a.exe  a.exe.stackdump  demo1  demo2  main.cpp  matoperation.cpp  matoperation.hpp
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ cd demo1
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples/demo1$ ls
exercise_demo
```

Create two subdirectories in the current directory

Create a subdirectory inside the demo1 directory

rmdir command

Use the **rmdir** command to permanently delete an empty directory.

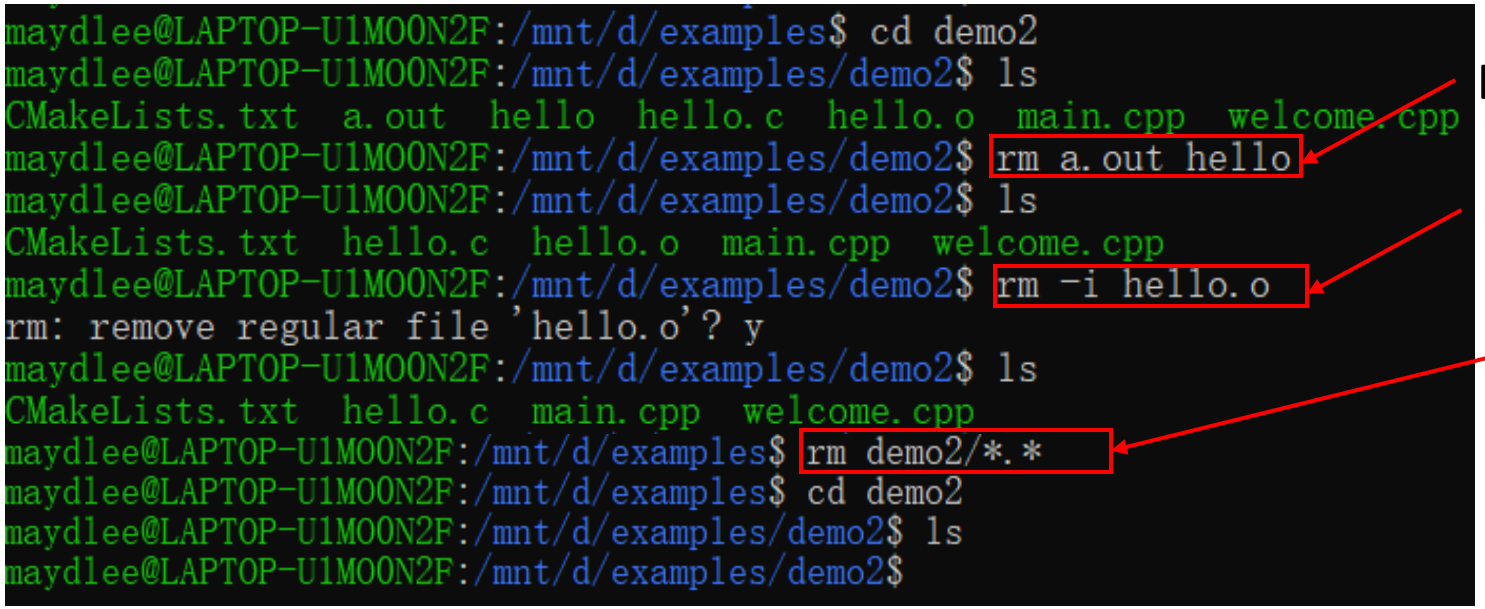
```
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ mkdir demo1 demo2
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ ls
CMakeLists.txt  a.exe  a.exe.stackdump  demo1  demo2  main.cpp  matoperation.cpp  matoperation.hpp
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ mkdir demo1/exercise_demo
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ ls
CMakeLists.txt  a.exe  a.exe.stackdump  demo1  demo2  main.cpp  matoperation.cpp  matoperation.hpp
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ cd demo1
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples/demo1$ ls
exercise_demo
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples/demo1$ cd ..
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ rmdir demo1
rmdir: failed to remove 'demo1': Directory not empty
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ rmdir demo1/exercise_demo
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ rmdir demo1
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ ls
CMakeLists.txt  a.exe  a.exe.stackdump  demo2  main.cpp  matoperation.cpp  matoperation.hpp
```

Delete demo1 in the current directory

First delete the directory in demo1, then delete demo1

rm command

The **rm** command is used to delete files within a directory. Make sure that the user performing this command has write permissions.



Delete two files without confirmation

Delete a file with confirmation

Delete all the files in demo2

Here are some acceptable options you can add:

- **-i** prompts system confirmation before deleting a file.
- **-f** allows the system to remove without a confirmation.
- **-r** deletes files and directories recursively.

cp command and mv command

The **cp** command is used to copy a file or directory.

```
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ ls
CMakeLists.txt  a.exe  a.exe.stackdump  demo2  main.cpp  matoperation.cpp  matoperation.hpp
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ cp main.cpp demo2
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ ls demo2
main.cpp
```

Copy a file into demo2

The **mv** command is used to move a file or a directory form one location to another location.

Move a file into demo2

```
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ mv CMakeLists.txt demo2
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ ls demo2
CMakeLists.txt  main.cpp
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ ls
a.exe  a.exe.stackdump  demo2  main.cpp  matoperation.cpp  matoperation.hpp
```

The CMakeLists.txt is not in the examples directory because it is moved into demo2.

Use mv command to rename a file

```
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ mv main.cpp test_main.cpp
maydlee@LAPTOP-U1MOON2F:/mnt/d/examples$ ls
a.exe  a.exe.stackdump  demo2  matoperation.cpp  matoperation.hpp  test_main.cpp
```

cat command

Concatenate, or **cat**, is one of the most frequently used Linux commands. It lists, combines, and writes file content to the standard output. To run the cat command, type **cat** followed by the file name and its extension.

```
maydlee@LAPTOP-U1MOON2F:/mnt/d/CMake$ cat HelloWorld.cpp
#include <iostream>
using namespace std;

int main()
{
        cout << "Hello World!" << endl;

        return 0;
}
```

Here are other ways to use the cat command:

- **cat > filename.txt** creates a new file.
- **cat filename1.txt filename2.txt > filename3.txt**
  merges **filename1.txt** and **filename2.txt** and stores the output in **filename3.txt**.
- **tac filename.txt** displays content in reverse order.

1.2 lists of commands, pipelines

- 1.2.1 lists of commands *https://www.gnu.org/software/bash/manual/bash.html#Lists*
  - An **AND list** has the form： command1 **&&** command2
    - ✓ **command2 is executed if, and only if, command1 returns an exit status of zero (success).**
  - An **OR list** has the form： command1 **||** command2
    - ✓ **command2 is executed if, and only if, command1 returns a non-zero exit status.**
  - **Command Sequence**: command1 **;** command2
    - ✓ **Commands separated by a ';' are executed sequentially;**
  - TIPS: *https://www.gnu.org/software/bash/manual/bash.html#Exit-Status*
    - ✓ For the shell's purposes, a command which **exits with a zero exit status has succeeded**. A non-zero exit status indicates failure.
    - ✓ The exit status of the last command is available in the special parameter **$?**

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/build$ ls
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/build$ cp *.cpp ../
cp: cannot stat '*.cpp': No such file or directory
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/build$ echo $?
1
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/build$ cp *.cpp ../ && echo and_list
cp: cannot stat '*.cpp': No such file or directory
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/build$ cp *.cpp ../ || echo or_list
cp: cannot stat '*.cpp': No such file or directory
or_list
```

1.2 lists of commands, pipelines

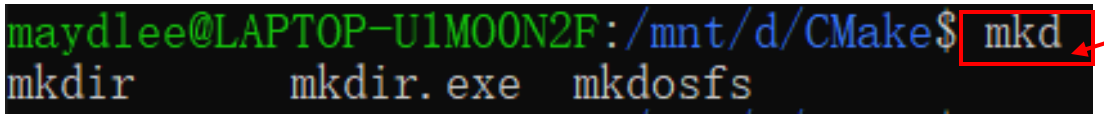
- 1.2.2 pipelines https://www.gnu.org/software/bash/manual/bash.html#Pipelines
  - A pipeline is a sequence of one or more commands separated by one of the control operators '|' or '|&'.
  - The output of each command in the pipeline is connected via a pipe to the input of the next command. That is, each command reads the previous command's output. This connection is performed before any redirections specified by command1.

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/src$ cat lab3_1.cpp
#include <iostream>
#include <iomanip>
using namespace std;

int main(){
    char x=0xff;
    unsigned char y=0xff;
    cout<<hex<<x<<endl;
    cout<<hex<<y<<endl;
}ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/src$ cat lab3_1.cpp | grep cout
    cout<<hex<<x<<endl;
    cout<<hex<<y<<endl;
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/src$ cat lab3_1.cpp | grep include
#include <iostream>
#include <iomanip>
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/src$ cat lab3_1.cpp | grep include | gre
p iostream
#include <iostream>
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/src$
```

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/build$ ls ..
build  inc  lab3_1.c  lab3_2.c  src  test
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/build$ echo .. | ls
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/build$ echo .. | xargs ls
build  inc  lab3_1.c  lab3_2.c  src  test
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/build$
```

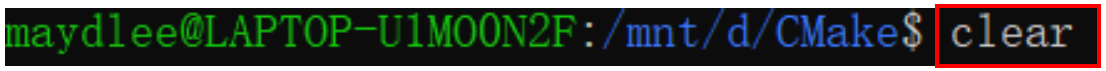
TIPS: Shortcut keys

- **Up** and **down** arrow keys can list the commands you typed.
- **Tab** key can complete the command. For a long command, you can type first few letters and press Tab key to complete the command or list alternate commands.



Type the first few letters of a command, and then press Tab key. If there is completion, press Tab key again, it will list the alternate commands.

clear is a standard Unix computer operating system command that is used to clear the terminal screen.



gcc & g++

gcc and **g++** are GNU C or C++ compilers respectively, which issued for preprocessing, compilation, assembly and linking of source code to generate an executable file.

Type command **gcc** or **g++ --help**, you can get the common options of the gcc or g++. **g++** accepts mostly the same options as **gcc**.

```
maydlee@LAPTOP-U1MOON2F: $ gcc --help
Usage: gcc [options] file...
Options:
  -pass-exit-codes         Exit with highest error code from a phase.
  --help                   Display this information.
  --target-help            Display target specific command line options.
  --help={common|optimizers|params|target|warnings|[^]{joined|separate|undocumented}}[,...].
                           Display specific types of command line options.
  (Use '-v --help' to display command line options of sub-processes).
  --version                Display compiler version information.
  -std=<standard>          Assume that the input sources are for <standard>.
  --sysroot=<directory>    Use <directory> as the root directory for headers
                           and libraries.
  -B <directory>           Add <directory> to the compiler's search paths.
  -v                       Display the programs invoked by the compiler.
  -###                     Like -v but options quoted and commands not executed.
  -E                       Preprocess only; do not compile, assemble or link.
  -S                       Compile only; do not assemble or link.
  -c                       Compile and assemble, but do not link.
  -o <file>                Place the output into <file>.
  -pie                     Create a dynamically linked position independent
                           executable.
  -shared                  Create a shared library.
```

gcc & g++

-c Compile or assemble the source files, but do not link. The ultimate output is in the form of an object file for each source file. The object file name for a source file is made by replacing the suffix **.c** with **.o**.

-o *<file>* Place output in file *file*. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code. If **-o** is not specified, the default is to put an executable file in *a.out*.

gcc source_file.c -o program_name ***or*** ***gcc source_file.o -o program_name***

```
#include <stdio.h>
int main()
{
    printf("Hello World!\n");

    return 0;
}
```

gcc & g++

With one step to generate an executable target file:

gcc file_name or ***g++ file_name***

This command is used to compile and create an executable file *a.out* (default target name).

```
#include <iostream>
using namespace std;
int main()
{
	cout << "Hello World!!!" << endl;

	return 0;
}
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ gcc hello.c
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ ls
a.out  hello  hello.c  helloworld.cpp
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ ./a.out
Hello World!
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ rm a.out hello
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ g++ helloworld.cpp
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ ls
a.out  hello.c  helloworld.cpp
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ ./a.out
Hello World!!!
```

gcc & g++

compile multiple files

You can compile the files one by one and then link them to an executable file.

Another choice is using one step to list all the .c(or .cpp) files after gcc(or g++) command and create an executable file named a.out.

```
//area.h
#define PI 3.1415

double compute_area(double r);
```

```
//area.c
#include "area.h"

double compute_area(double r)
{
   return PI * r * r;
}
```

```
//main.c
#include <stdio.h>
#include "area.h"

int main()
{
   double r, area;

   printf("Please input a radius:");
   scanf("%lf", &r);

   area = compute_area(r);

   printf("The area of %lf is %.4lf\n", r, area);

   return 0;
}
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ gcc -c area.c
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ gcc -c main.c
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ ls
area.c  area.h  area.o  hello.c  helloworld.cpp  main.c  main.o
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ gcc -o main main.o area.o
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ ls
area.c  area.h  area.o  hello.c  helloworld.cpp  main  main.c  main.o
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ ./main
Please input a raduis:4.8
The area of 4.800000 is 72.3802
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ gcc area.c main.c
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab_example/lab03$ ./a.out
Please input a raduis:2.3
The area of 2.300000 is 16.62
```

Makefile

What is a Makefile?

Makefile is a tool to simplify and organize compilation. **Makefile is a set of commands with variable names and targets .** You can compile your project(program) or only compile the update files in the project by using Makefile.

Suppose we have four source files as follows:

multifiles > functions.h > ...

```h
#pragma once

#define N 5

void printinfo();
int factorial(int n);
```

multifiles > printinfo.cpp > ...

```cpp
#include <iostream>
#include "functions.h"

void printinfo()
{
    std::cout << "Let's go!" << std::endl;
}
```

multifiles > factorial.cpp > factorial(int)

```cpp
#include "functions.h"

int factorial(int n)
{
    if(n == 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

multifiles > main.cpp > ...

```cpp
#include <iostream>
#include "functions.h"
using namespace std;

int main()
{
    printinfo();

    cout << "The factorial of "<< N << " is:" << factorial(N) << endl;

    return 0;
}
```

Normally, you can compile these files by the following command:

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ g++ -o testfiles main.cpp printinfo.cpp factorial.cpp
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ ./testfiles
Let's go!
The factorial of 5 is:120
```

or `g++ *.cpp`

How about if there are hundreds of files to compile? If only one source file is modified, need we compile all the files? Makefile will help you.

The name of makefile must be either **makefile** or **Makefile** without extension. You can write makefile in any text editor. A rule of makefile including three elements: **targets**, **prerequisites** and **commands**. There are many rules in the makefile.

A makefile consists of a set of rules. A rule including three elements: **target**, **prerequisites** and **commands**.

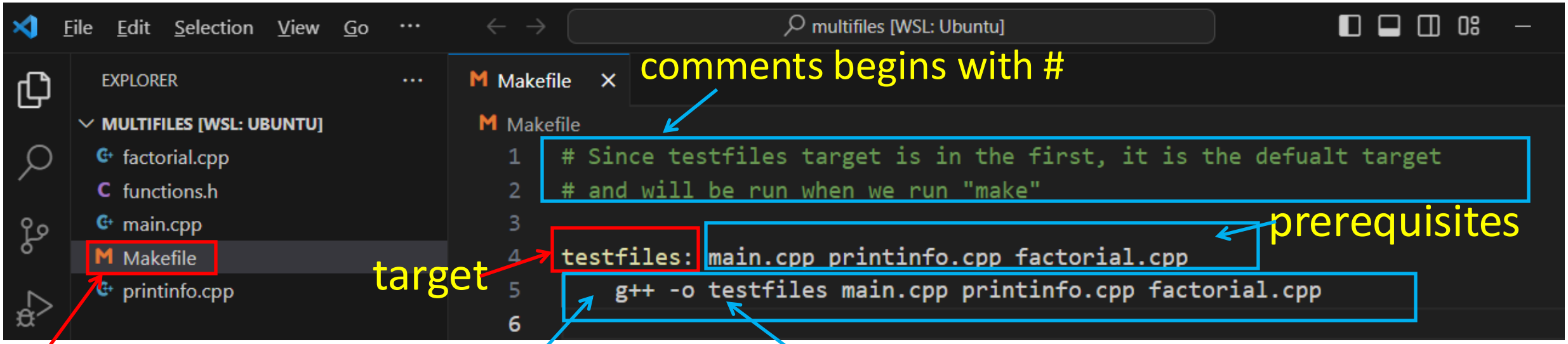
```
targets : prerequisites
<TAB> command
```

- The **target** is an object file, which is generated by a program.

Typically, there is only one per rule.

- The **prerequisites** are file names, separated by spaces, as input to create the target.
- The **commands** are a series of steps that make carries out.

These need to start with a **tab character**, not spaces.



Place the **Makefile** together with your programs.

start with <TAB>

commands

g++ is compiler name, -o is linker flag and testfiles is binary file name.

Type the command **make** in VScode

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ make
```

If you don't install make in VScode, the information will display on the screen.

```
Command 'make' not found, but can be installed with:
sudo apt install make          # version 4.2.1-1.2, or
sudo apt install make-guile    # version 4.2.1-1.2
```

Install it first according to the instruction.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ make
g++ -o testfiles main.cpp printinfo.cpp factorial.cpp
```

Run the commands in the **makefile** automatically.

Run your program

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ ./testfiles
Let's go!
The factorial of 5 is:120
```

output

Define Macros/Variables in the makefile

To improve the efficiency of the **makefile**, we use variables.

variables

```
Makefile
1   # Using variables in makefile
2   CXX = g++
3   TARGET = testfiles
4   OBJ = main.o printinfo.o factorial.o
5   $(TARGET) : $(OBJ)
6       $(CXX) -o $(TARGET) $(OBJ)
```

start with <TAB>

Write target, prerequisite and commands by variables using '$()'

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ make
g++     -c -o main.o main.cpp
g++     -c -o printinfo.o printinfo.cpp
g++     -c -o factorial.o factorial.cpp
g++ -o testfiles main.o printinfo.o factorial.o
```

Compile and link the source file one by one

Note: Deletes all the .o files and executable file created previously before using make command. Otherwise, it'll display:

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ make
make: 'testfiles' is up to date.
```

If only one source file is modified, we need not compile all the files. So, let's modify the makefile.

Makefile

```
1   # Using variables in makefile
2   CXX = g++
3   TARGET = testfiles
4   OBJ = main.o printinfo.o factorial.o
5   $(TARGET) : $(OBJ)
6       $(CXX) -o $(TARGET) $(OBJ)
7
8   main.o : main.cpp
9       $(CXX) -c main.cpp
10
11  printinfo.o : printinfo.cpp
12      $(CXX) -c printinfo.cpp
13
14  factorial.o : factorial.cpp
15      $(CXX) -c factorial.cpp
```

targets

If main.cpp is modified, it is compiled by make.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ make
g++ -c main.cpp
g++ -o testfiles main.o printinfo.o factorial.o
```

All the .cpp files are compiled to the .o files, so we can modify the makefile like this:

```
# Using several ruls and several targes

CXX = g++
TARGET = testfiles
OBJ = main.o printinfo.o factorial.o

# options pass to the compiler
# -c generates the object file
# -Wall displays compiler warning
CFLAGES = -c -Wall

$(TARGET) : $(OBJ)
    $(CXX) $^ -o $@

%.o : %.cpp
    $(CXX) $(CFLAGES) $< -o $@
```

$(TARGET) : $(OBJ)
    $(CXX) -o $@ $(OBJ)

This is a model rule, which indicates that all the .o objects depend on the .cpp files

$@: the target file

$^: all the prerequisites files

$<: the first prerequisite file

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ make
g++ -c -Wall main.cpp -o main.o
g++ -c -Wall printinfo.cpp -o printinfo.o
g++ -c -Wall factorial.cpp -o factorial.o
g++ main.o printinfo.o factorial.o -o testfiles
```

```
%.o : %.cpp
    $(CXX) $(CFLAGS) $<
```

or

```
$(CXX) $(CFLAGES) $^
```

Using phony target to clean up compiled results automatically

```
# Using several ruls and several targes

CXX = g++
TARGET = testfiles
OBJ = main.o printinfo.o factorial.o

# options pass to the compiler
# -c generates the object file
# -Wall displays compiler warning
CFLAGES = -c -Wall

$(TARGET) : $(OBJ)
    $(CXX) -o $@ $(OBJ)

%.o : %.cpp
    $(CXX) $(CFLAGES) $^

.PHONY : clean
clean:
    rm -f *.o $(TARGET)
```

Because **clean** is a label not a target, the command **make clean** can execute the clean part. Only **make** command can not execute clean part.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ make clean
rm -f *.o testfiles
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ make
g++ -c -Wall main.cpp
g++ -c -Wall printinfo.cpp
g++ -c -Wall factorial.cpp
g++ -o testfiles main.o printinfo.o factorial.o
```

After clean, you can run make again

Adding **.PHONY** to a target will prevent making from confusing the phony target with a file name.

Functions in makefile

wildcard: search file
for example:

Search all the .cpp files in the current directory, and return to SRC

```
SRC = $(wildcard ./*.cpp)
```

```
SRC = $(wildcard ./*.cpp)
target:
	@echo $(SRC)
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ make
./printinfo.cpp ./factorial.cpp ./main.cpp
```

All .cpp files in the current directory

patsubst(pattern substitution): replace file
$(patsubst original pattern, target pattern, file list)

for example:

Replace all .cpp files with .o files

```
OBJ = $(patsubst  %.cpp,  %.o, $(SRC))
```

```
SRC = $(wildcard ./*.cpp)
OBJ = $(patsubst %.cpp, %.o, $(SRC))
target:
    @echo $(SRC)
    @echo $(OBJ)
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ make
./printinfo.cpp ./factorial.cpp ./main.cpp
./printinfo.o ./factorial.o ./main.o
```

Replace all .cpp files with .o files

```
# Using functions

SRC = $(wildcard ./*.cpp)
OBJS = $(patsubst %.cpp, %.o, $(SRC))
TARGET = testfiles

CXX = g++
CFLAGES = -c -Wall

$(TARGET) : $(OBJS)
    $(CXX) -o $@ $(OBJS)
%.o : %.cpp
    $(CXX) $(CFLAGES) $<

.PHONY : clean
clean:
    rm -f *.o $(TARGET)
```

vs

```
OBJ = main.o printinfo.o factorial.o
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ make
g++ -c -Wall printinfo.cpp
g++ -c -Wall factorial.cpp
g++ -c -Wall main.cpp
g++ -o testfiles  ./printinfo.o  ./factorial.o  ./main.o
```

Use Options to Control Optimization

-O1, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

-O2,Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O1, this option increases both compilation time and the performance of the generated code.

-O3, Optimize yet more. O3 turns on all optimizations specified by -O2.

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

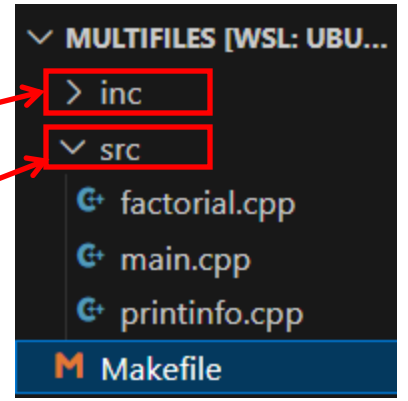
https://blog.csdn.net/xinianbuxiu/article/details/51844994

```
SRC_DIR = ./src
SOURCE  = $(wildcard $(SRC_DIR)/*.cpp)
OBJS    = $(patsubst %.cpp, %.o, $(SOURCE))
TARGET  = testfiles
INCLUDE = -I./inc

# Options pass to complier
# -c: generates the object file
# -Wall: displays complier warnings
# -O0: no optimization
# -O1: dafault optimization
# -O2: represents the second level optimization
# -O3: represents the highest level optimization

CXX      = g++
CFLAGES = -c -Wall
CXXFLAGES = $(CFLAGES) -O3

$(TARGET) : $(OBJS)
    $(CXX) -o $@ $(OBJS)
%.o : %.cpp
    $(CXX) $(CXXFLAGES) $< -o $@ $(INCLUDE)

.PHONY :clean
clean:
    rm -f $(SRC_DIR)/*.o $(TARGET)
```

-I means search file(s) in the specified folder i.e. inc folder

All .h files are in inc

All .cpp files are in src




```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ make
g++ -c -Wall -O3  src/printinfo.cpp -o src/printinfo.o -I./inc
g++ -c -Wall -O3  src/factorial.cpp -o src/factorial.o -I./inc
g++ -c -Wall -O3  src/main.cpp -o src/main.o -I./inc
g++ -o testfiles  ./src/printinfo.o  ./src/factorial.o  ./src/main.o
maydlee@LAPTOP-U1MO0N2F:/mnt/d/makefile/multifiles$ ls
Makefile  inc  src  testfiles
```

GNU Make Manual

http://www.gnu.org/software/make/manual/make.html

Exercises 1. commands and command list

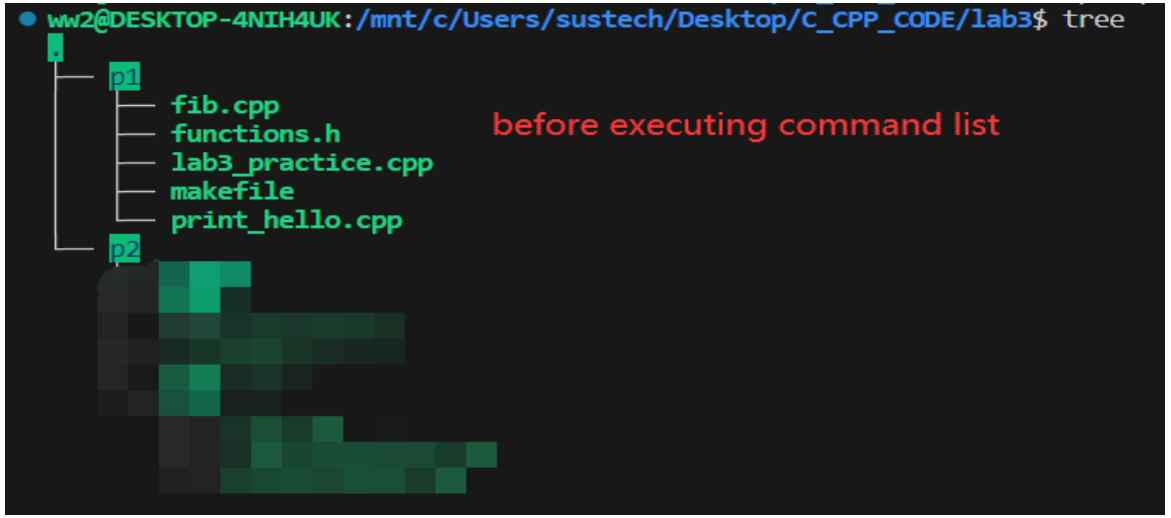
The existing directory structure is shown in the upper right image. There are different types of C/C++files in the “p1” directory while the directory structure under p2 is unknown.

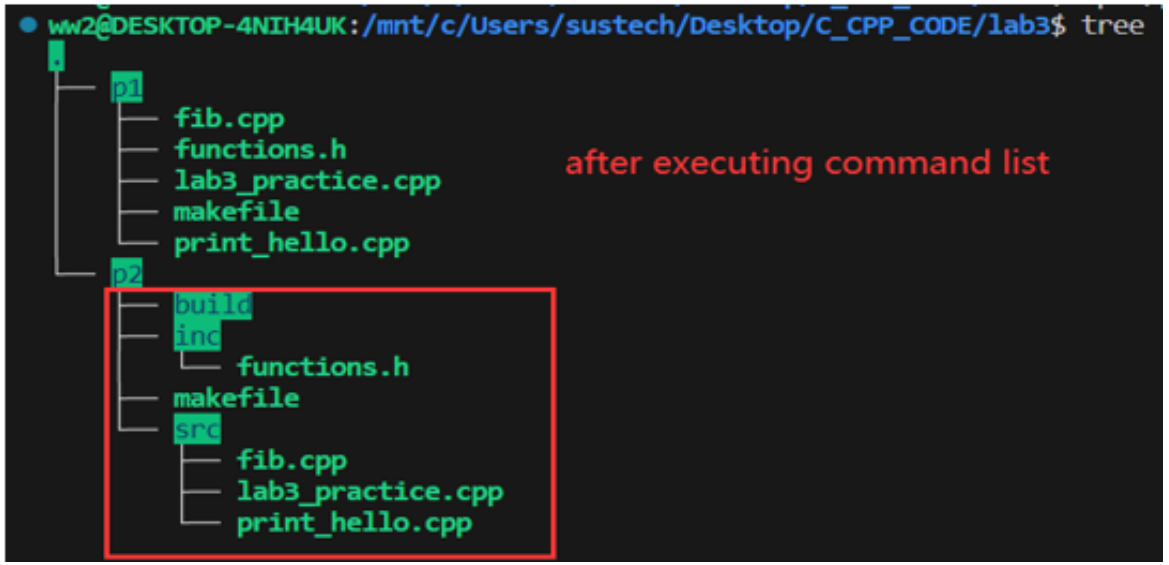
Task. use the command list to create subdirectories as needed and place files of different types into different subdirectories in the “p2” directory (as shown in the lower right image).Place the header file in p2/inc, and the cpp source file in p2/src, and create p2/build.

NOTE:
1. when using commands, if there is already an “inc” subdirectories, do not create “inc” repeatly. If there is no “inc” subdirectories, create it;
The same requirement also applies to “src” and “build”.

2. File copying work should only be performed after the destination subdirectories have been created.

3. Use as few command lists as possible to complete this exercise (options include “and list”, “or list”, “command sequence”, which can be combined as needed)





Exercises 2. Makefile and make(1)

create a makefile, run it by command “make” or “make clean” to complete following tasks:

1. compile your project(program) or only compile the update files in the project by running “make”based on makefile to generate the executable file “lab3_practice”.

notes: the object file *.o and the executable file “lab3_practice” should be in the directory “build”

2. remove all the files in the directory “build” by running “make clean”

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/p2$ tree
.
├── build
├── inc
│   └── functions.h
├── makefile
└── src
    ├── fib.cpp
    ├── lab3_practice.cpp
    └── print_hello.cpp

3 directories, 5 files
```

before running make

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/p2$ make
g++ -I ./inc  -c src/fib.cpp -o build/fib.o
g++ -I ./inc  -c src/print_hello.cpp -o build/print_hello.o
g++ -I ./inc  -c src/lab3_practice.cpp -o build/lab3_practice.o
g++  -o ./build/lab3_practice  ./build/fib.o  ./build/print_hello.o  ./build/lab3_practice.o
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/p2$ tree
.
├── build
│   ├── fib.o
│   ├── lab3_practice
│   ├── lab3_practice.o
│   └── print_hello.o
├── inc
│   └── functions.h
├── makefile
└── src
    ├── fib.cpp
    ├── lab3_practice.cpp
    └── print_hello.cpp

3 directories, 9 files
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/p2$
```

after running make

Exercises 2. Makefile and make(2)

```
//print_hello.cpp
#include <iostream>
#include "functions.h"
using namespace std;
void print_hello(){
   cout<<"Hello World!"<<endl;
}
```

```
//functions.h
void print_hello();
int fib(int n);
```

```
//fib.cpp
#include "functions.h"
int factorial(int n) {
   if(1==n)  return 1;
   else  return n * fib(n-1);
}
```

```
//lab3_practice.cpp
#include <iostream>
#include "functions.h"
using namespace std;

int main(){
   print_hello();
   cout<<"This is main:"<<endl;
   cout<<"The factorial of 5 is: "
         <<fib(5)<<endl;
   return 0;
}
```

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/p2$ ./build/lab3_practice
Hello World!
This is main:
The factorial of 5 is: 120
```

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/p2$ make
g++ -I ./inc  -c src/lab3_practice.cpp -o build/lab3_practice.o
g++  -o ./build/lab3_practice  ./build/fib.o  ./build/print_hello.o  ./build/lab3_practice.o
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab3/p2$ ./build/lab3_practice
Hello World!
This is main:
The factorial of 6 is: 720
```

3. edit the source file “lab3_practice.cpp” (change parameter 5 to another number), save it, then run “make” again, which object file would be updated in the process?

Exercises

3. Run the following source code and explain the result.

You need to explain the reason to a SA to pass the test.

```
#include <iostream>
using namespace std;

int main()
{
   for(size_t n = 2; n >= 0; n--)
      cout << "n = " << n << "  ";

   return 0;

}
```

Exercises

4. Run the following source code and explain the result.

You need to explain the reason to a SA to pass the test.

```
#include <iostream>
using namespace std;

int main()
{
   int n = 5;
   int sum;
   while(n >0){
      sum += n;
      cout << "n = " << n << "  ";
      cout << "sum = " << sum << "  ";
   }
   return 0;
}
```

```
#include <iostream>
using namespace std;

int main()
{
   unsigned int n = 5;
   int sum;
   while(n >0){
      sum += n;
      cout << "n = " << (n-=2) <<endl;
      cout << "sum = " << sum << "  ";
   }
   return 0;
}
```

Exercises

5. Run the following source code and explain the result.

You need to explain the reason to a SA to pass the test.

```
#include <iostream>
using namespace std;

int main()
{
   int n,fa;

   do{
      fa *= n;
      n++;
   }while(n <= 10);

   cout << "fa = " << fa << endl;

   return 0;
}
```