Makefile — Practical Notes

These notes help you understand and use **Makefiles** for building C / C++ projects in Linux.

✅What is `make` and a Makefile?

make is a build automation tool. It executes commands based on rules written in a **Makefile**.

A Makefile: - Compiles source files - Links executables - Rebuilds only changed files (saves time)

Basic Structure of a Rule

```
<target>: <dependencies>
[TAB] <command>
```

Example:

```
main.o: main.c
    gcc -c main.c
```

- **target** → file to be generated
- **dependencies** → files required
- **command** → must start with TAB

  If a dependency is newer than target, the rule runs again.

⭕Example 1: Simple C Program with add, sub, mul using math.h

Files: `add.c`, `sub.c`, `mul.c`, `main.c`, `math.h`

math.h

```c
#ifndef MATH_H
#define MATH_H

int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
```

```
#endif
```

add.c

```
#include <stdio.h>
#include "math.h"
int add(int a, int b) { return a + b; }
```

sub.c

```
#include <stdio.h>
#include "math.h"
int sub(int a, int b) { return a - b; }
```

mul.c

```
#include <stdio.h>
#include "math.h"
int mul(int a, int b) { return a * b; }
```

main.c

```
#include <stdio.h>
#include "math.h"

int main() {
    int a, b, choice;
    printf("Enter a and b: ");
    scanf("%d %d", &a, &b);

    printf("Choose operation (0: add, 1: sub, 2: mul): ");
    scanf("%d", &choice);

    switch(choice) {
        case 0:
            printf("Add = %d\n", add(a,b));
            break;
        case 1:
            printf("Sub = %d\n", sub(a,b));
            break;
```

```
        case 2:
            printf("Mul = %d\n", mul(a,b));
            break;
        default:
            printf("Invalid choice\n");
    }
    return 0;
}
```

Makefile (Example 1)

```
CC=gcc
CFLAGS=-Wall

all: main

main: main.o add.o sub.o mul.o
    $(CC) main.o add.o sub.o mul.o -o main

main.o: main.c math.h
    $(CC) $(CFLAGS) -c main.c

%.o: %.c math.h
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f *.o main
```

Run:

```
make        # compile and build
./main     # run program
make clean  # remove object files and executable
```

⭕Example 2: C Program with Static and Dynamic Library Creation

Files: same as Example 1 (`add.c`, `sub.c`, `mul.c`, `main.c`, `math.h`)

Makefile (Example 2 with static and dynamic library)

```
CC=gcc
CFLAGS=-Wall -fPIC
```

```
AR=ar
LDFLAGS=-shared
LIB_STATIC=libmath.a
LIB_DYNAMIC=libmath.so
TARGET_STATIC=app_static
TARGET_DYNAMIC=app_dynamic

SRC=add.c sub.c mul.c
OBJ=$(SRC:.c=.o)

all: $(TARGET_STATIC) $(TARGET_DYNAMIC)

# Static library
$(LIB_STATIC): $(OBJ)
    $(AR) rcs $@ $^

$(TARGET_STATIC): main.o $(LIB_STATIC)
    $(CC) main.o -L. -lmath -o $@

# Dynamic library
$(LIB_DYNAMIC): $(OBJ)
    $(CC) $(LDFLAGS) -o $@ $^

$(TARGET_DYNAMIC): main.o $(LIB_DYNAMIC)
    $(CC) main.o -L. -lmath -o $@

main.o: main.c math.h
    $(CC) $(CFLAGS) -c main.c

%.o: %.c math.h
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f *.o $(LIB_STATIC) $(LIB_DYNAMIC) $(TARGET_STATIC) $(TARGET_DYNAMIC)
```

Run:

```
make                # build both static and dynamic executables
./app_static        # run static-linked program
LD_LIBRARY_PATH=. ./app_dynamic   # run dynamic-linked program
make clean          # remove object files, libraries, and executables
```