
Makefile
BUILD C/C++ PROGRAMS
SUBRAT KUMAR SWAIN
18YRS IN C++ DEVELOPMENT

AGENDA

- What is going on behind the scene
- Possible Outputs Of a C/C++ Program
- Project Folder Structure Creation
- Understanding Make & Makefile
- Content of Makefile by comparing with C Prog
- Understand Make Terminologies Using RT Example
- Who Will Clean?(PHONY INTRO)

What You Will Get Out Of This Course

MAC
& LINUX
G++ & Clang++

Begin
With High Level
Designing

CROSS
COMPILATION
FEATURE

Command
Line Compilation

Passing
MACRO inside
Makefile

Accessing
Environment
Variables Inside
Makefile

Trick
For Fixing
Linking Issue

One
Make To Call
Other Make

Deployment
Easy

Can
Be Used In Real
Time Project

HLD
DIAGRAM
TO
IMPLEMENTATION

$ ^ < @ %

ENVIRONMENT SETUP

```
┌─subrat@Subrats-MacBook-Air.local ~
└─➤  make -v
GNU Make 3.81
Copyright (C) 2006  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

This program built for i386-apple-darwin11.3.0
```

xcode-select --install

```
➜  make -v
GNU Make 4.2.1
Built for x86_64-pc-linux-gnu
Copyright (C) 1988-2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/
gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

sudo apt-get install build-essential

VIM & VIM TOOLS INSTALLATION

sudo apt-get install vim

```
git clone --depth=1 https://github.com/amix/vimrc.git ~/.vim_runtime
sh ~/.vim_runtime/install_awesome_vimrc.sh
```

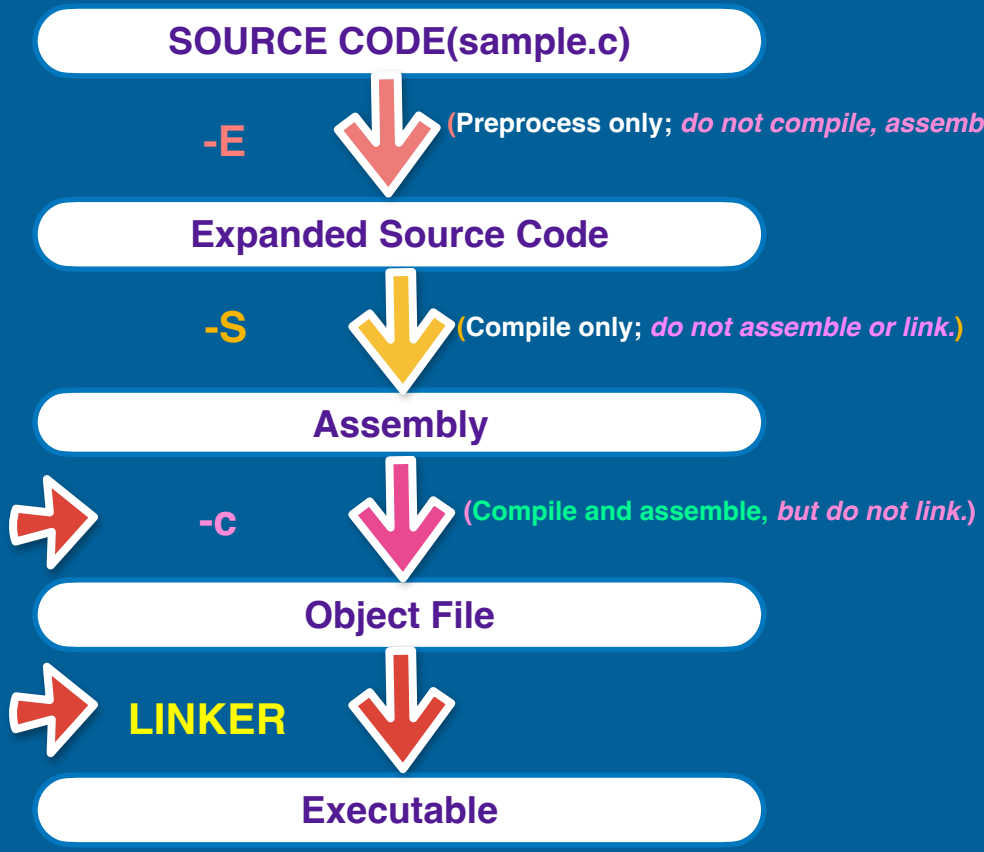
---

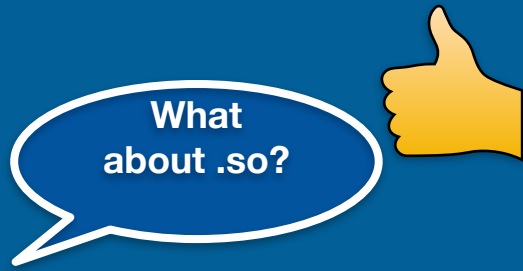
brew instal vim

```
git clone --depth=1 https://github.com/amix/vimrc.git ~/.vim_runtime
sh ~/.vim_runtime/install_awesome_vimrc.sh
```

Refer for Ultimate Vim Configuration: https://github.com/amix/vimrc

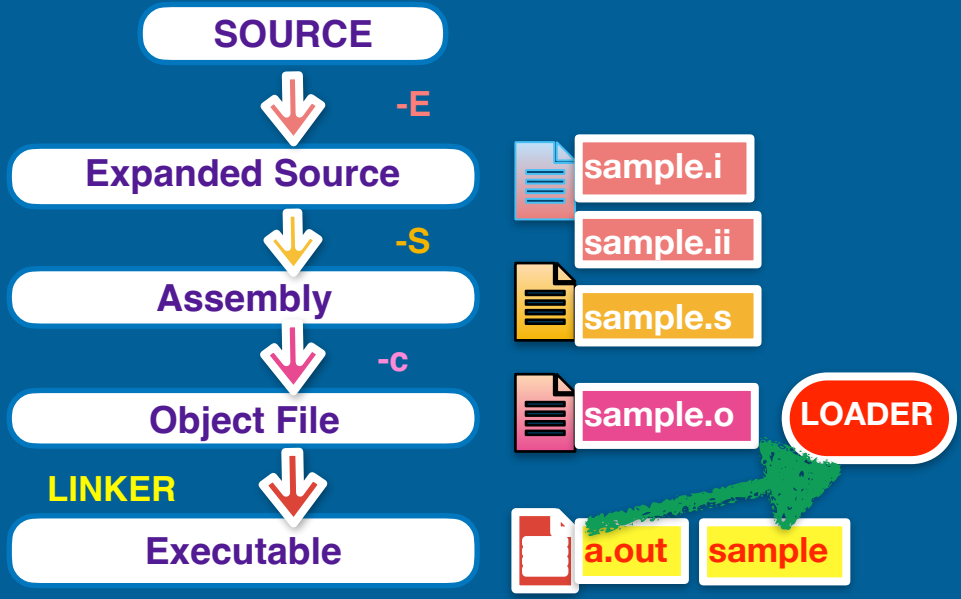
Behind The Scene





Exercise

- Generate all intermediate files
- Generate only expanded source code
- Generate expanded source code for **C++**
- Generate assembly file
- Generate Obj file.
- Generate binary executable.
- Use -S and observe the intermediate files
- Use -c and observe the intermediate files
- Compile using assembly input



-save-temps

Commands To Use

```
┌──subrat@Subrats-MacBook-Air.local ~/udemy/first_demo
└─➤  ls
sample.cpp
┌──subrat@Subrats-MacBook-Air.local ~/udemy/first_demo
└─➤  g++ -E sample.cpp -o sample.ii
┌──subrat@Subrats-MacBook-Air.local ~/udemy/first_demo
└─➤  ls
sample.cpp sample.ii
┌──subrat@Subrats-MacBook-Air.local ~/udemy/first_demo
└─➤  g++ -S sample.ii -o sample.s
┌──subrat@Subrats-MacBook-Air.local ~/udemy/first_demo
└─➤  ls
sample.cpp sample.ii  sample.s
┌──subrat@Subrats-MacBook-Air.local ~/udemy/first_demo
└─➤  g++ -c sample.s -o sample.o
┌──subrat@Subrats-MacBook-Air.local ~/udemy/first_demo
└─➤  ls
sample.cpp sample.ii  sample.o   sample.s
```

```
┌──subrat@Subrats-MacBook-Air.local ~/udemy/first_demo
└─➤  g++ sample.o -o output
┌──subrat@Subrats-MacBook-Air.local ~/udemy/first_demo
└─➤  ls
output     sample.cpp sample.ii  sample.o   sample.s
┌──subrat@Subrats-MacBook-Air.local ~/udemy/first_demo
└─➤  ./output
GNU MAKE!
```

-save-temps

Write Our First Makefile

C

```c
#include <stdio.h>

int main() {
    printf("Hello GNU Make\n");
    return 0;
}
```

Makefile

```
default:
    gcc sample.c -o samplec
```

Write Our First Makefile

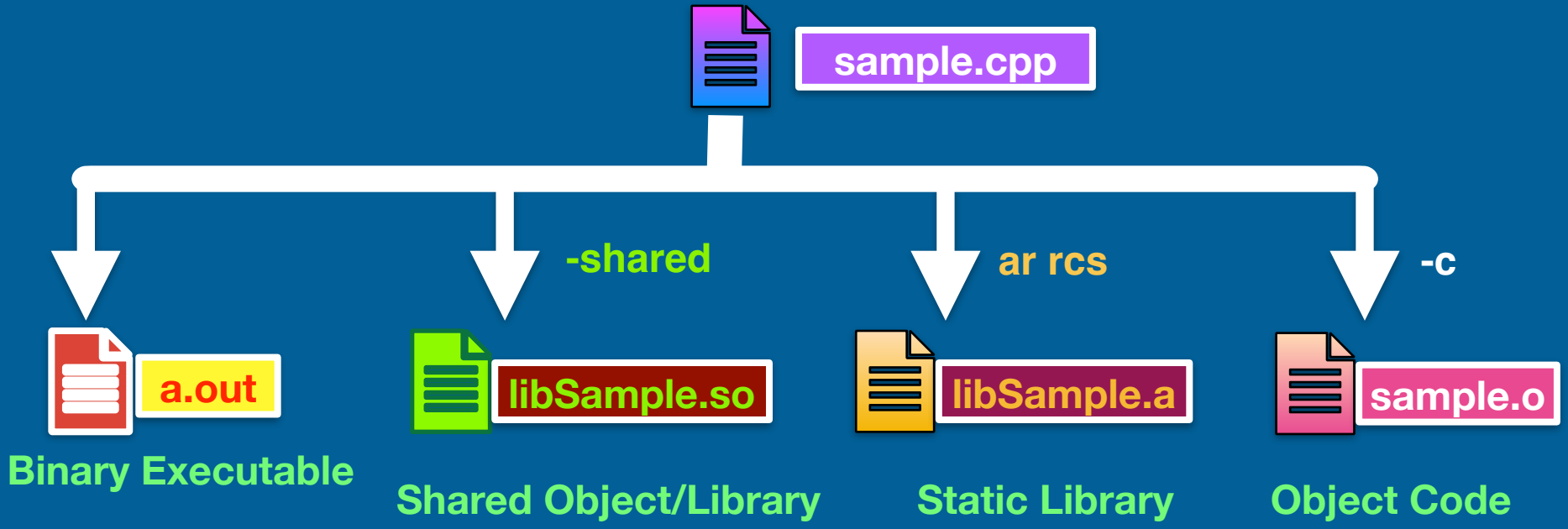
C++

```cpp
#include <iostream>

int main() {
    std::cout << "Hello GNU Make\n";
    return 0;
}
```

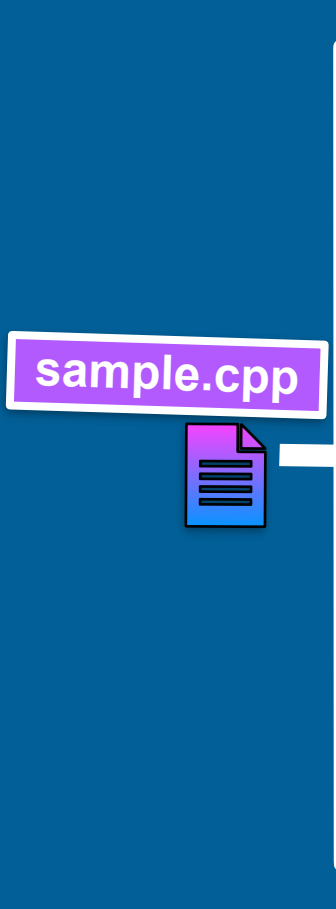
Makefile

```
default:
	g++ sample.cpp -o samplecpp
```

Possible Outputs



Possible Outputs-II



g++ -c sample.cpp -o sample.o

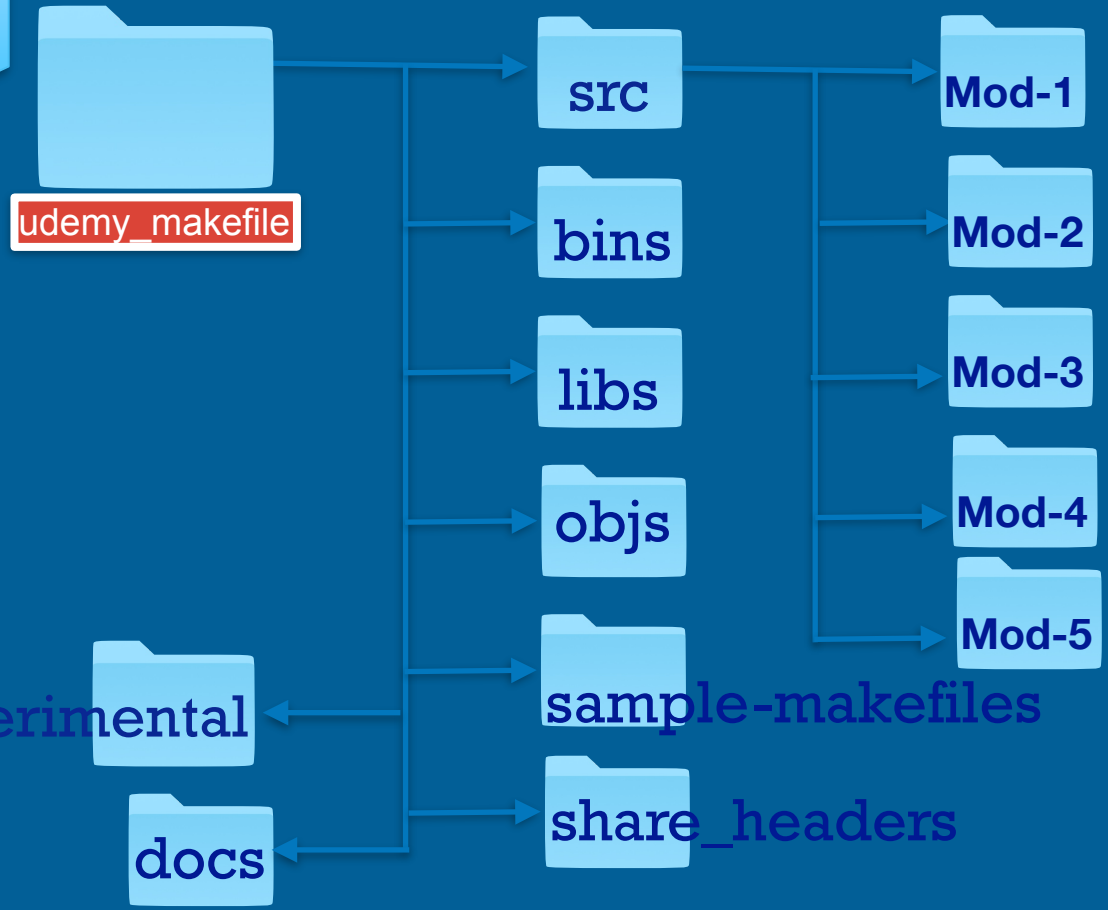
ar rcs libsample.a sample.o

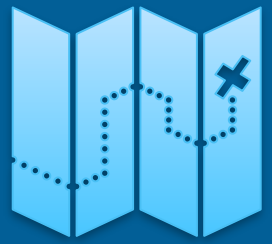
Archive
replace(old files), create(if not exist), sort
(create a sorted index of the library)

g++ -shared sample.cpp -o libsample.so

g++ sample.cpp -o output

Create Project Folder Structure





How?

```
→Oct2020 mkdir udemy_makefile
→Oct2020 cd udemy_makefile/
→udemy_makefile mkdir src objs bins libs sample-makefiles
→udemy_makefile ls
bins  libs  objs  sample-makefiles  src
```

Make Vs Makefile

- Make is a tool
- Makefile is a file written in such a way that Make tool can understand
- Make tool needs Makefile as the input and process as per the instruction written inside it.
- Makefile = makefile - Both are valid

```
→ which make
  /usr/bin/make
→ make ✓
→ makefile ✗
```

Why Makefile

Live Dem

- Makefile is used for automating build process
- Make tool is powerful
- Make tool can detect the changed source file and process/build that file while we do make.

Let me to show you how it is working

TARGET : PREREQUISITE

INSTRUCTION-HOW TO MAKE/PREPARE THE TARGET USiNG PREREQUISITES

WHAT MAKEFILE CONTAINS?

C

```
// Comments
// MACROS
// Variable declaration/definition
// functions declaration/definition
// Instructions/Statements
```

main()

Makefile

```
# Comments
# MACROS/VARIABLES
# functions
# Instructions/Statements
```

TARGET

MAKEFILE

VARIABLE1
VARIABLE2

define myfunction
……..

endef

Rule

Recipe

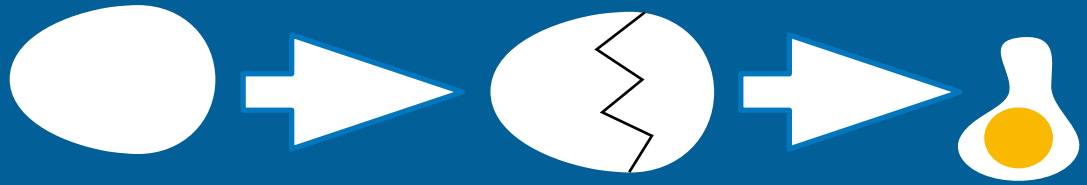
TARGET : PREREQUISITE / DEPENDENCY

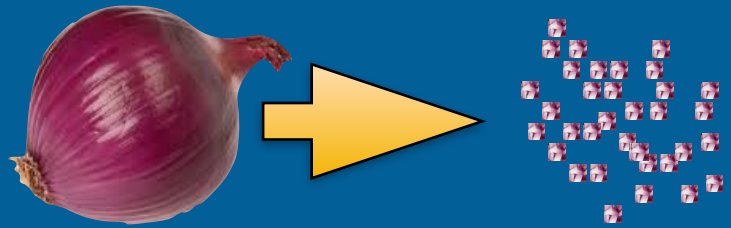
INSTRUCTION-HOW TO MAKE/PREPARE THE TARGET USiNG PREREQUISITES

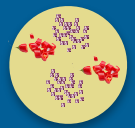
Tab/~~Spaces~~

Real World Make *(Egg Omelet)*

COOK	=	make tool
Procedure	=	Makefile







Real World Makefile

REALTIME EXAMPLE

[EGG OMELET] : USEFUL PARTS OF EGGS Onion-Chopped Chopped-RedChilli Chopped-Tomato

MIX the above items, cook for 5mints and get the EGG OMELET

USEFUL PARTS OF EGGS : EGG1 EGG2

Break the two EGGS and collect the USEFUL PARTS OF EGGS

Onion-Chopped : Onion

Cut the ONION in small pieces and collect the useful parts(Onion-Chopped)

Chopped-RedChilli : RedChilli

Cut the RedChilli in small pieces and collect the useful parts(Chopped CHILLI)

TARGET : PREREQUISITE

INSTRUCTION-HOW TO MAKE/PREPARE THE TARGET USiNG PREREQUISITES

Who Will Clean?

ACTUAL TARGET/
PHYSICAL TARGET

NOT ACTUAL TARGET/ NOT
PHYSICALLY PRESENT

PHONY

Cleaning
OBJ/libs/binaries

Create
build directories
Obj/lib/bin

Clean :

Clean the Kitchen .....

Prepare-Containers :

Collect The Required Containers, Clean and make them READY

.PHONY : Clean Prepare-Containers

TARGET : PREREQUISITE

INSTRUCTION-HOW TO MAKE/PREPARE THE TARGET USiNG PREREQUISITES

Writing Makefile

1. Write a makefile to produce d1,d2,d3 directories
2. Modify above makefile to produce f1.c, f2.c, f3.c files
3. Modify above makefile to delete f3.c and d3
4. Suppress Error.
5. Modify makefile to do step-1 & step-2 in single run
6. Write Comment in makefile

TARGET : PREREQUISITE / DEPENDENCY

INSTRUCTION-HOW TO MAKE/PREPARE THE TARGET USiNG PREREQUISITES

ASSIGNMENT

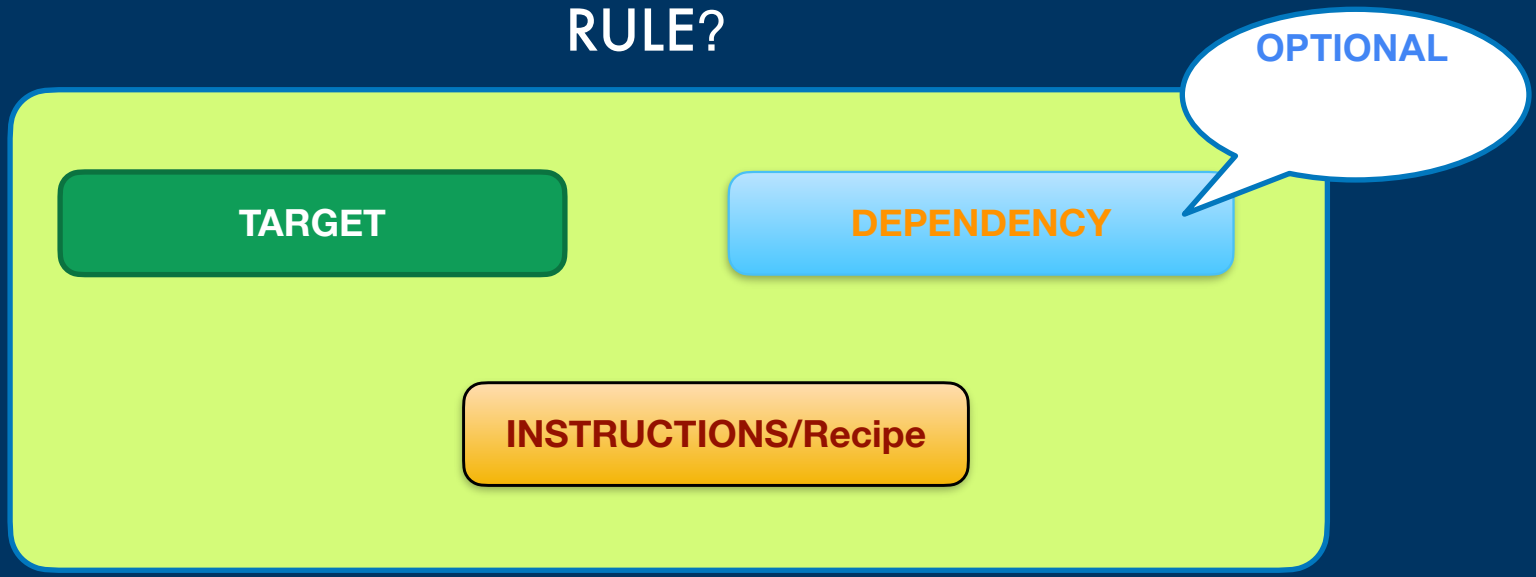
Write a makefile to print date and it shouldn’t show string “date” in output terminal. It should display current date and time.

NOTE

The command that we are using in terminal/command line, that can be used in makefile as instruction!

Exercise

RULE?

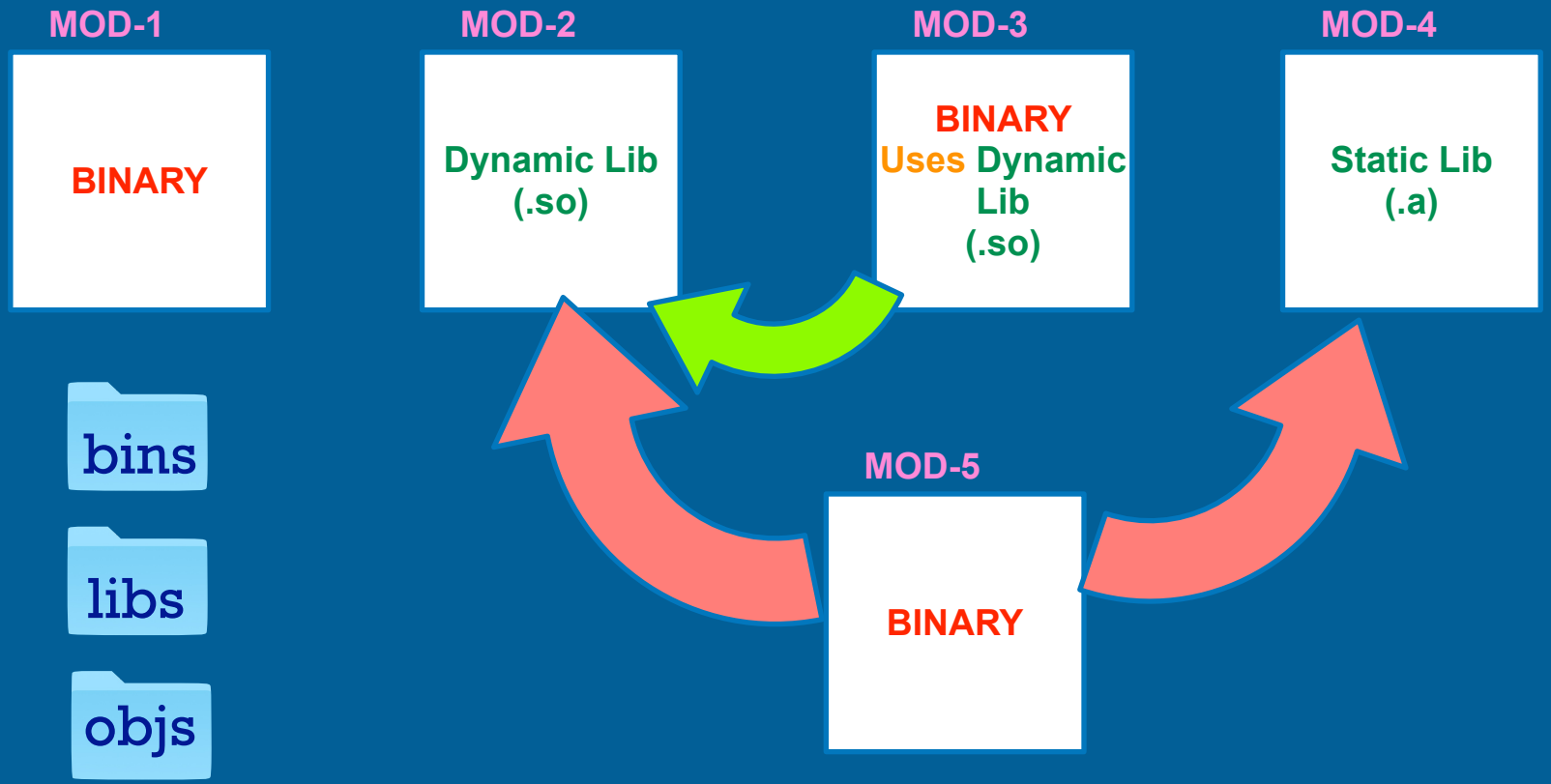


Points To Remember

```
make {TARGET_NAME_TO_EXECUTE_OPT}
```

1. Makefile may have multiple targets.
2. By Default First TARGET gets executed
3. Use .DEFAULT_GOAL to override it

Understanding Requirement



Understanding Module-1

MOD-1

BINARY

Module-1 C IMPLEMENTATION

MOD-1

BINARY

Write A Program To Print Square Root of 15.0

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

int main() {
    printf("%lf\n", sqrt(15.0));
    return 0;
}
```

Makefile

```
default:
	gcc mod1.c -o ../../bins/mod1
```

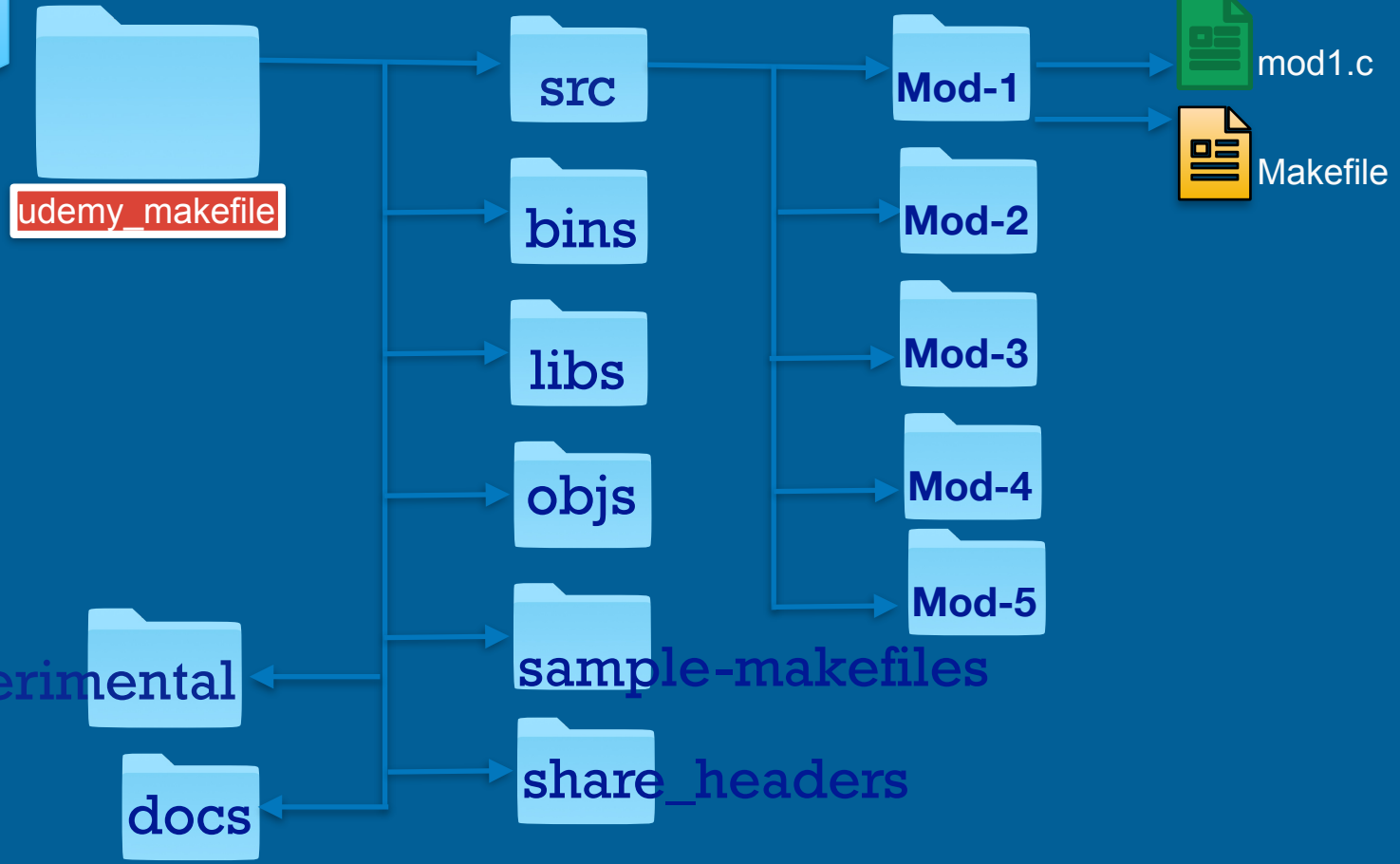
TARGET : PREREQUISITE

INSTRUCTION-HOW TO MAKE/PREPARE THE TARGET USiNG PREREQUISITES

Run make

```
bash-3.2$ make
gcc mod1.c -o ../../bins/mod1
bash-3.2$ ls ../../bins
mod1
```

Create Project Folder Structure



Makefile

```
default:
	gcc mod1.c -o ../../bins/mod1
```

TARGET : PREREQUISITE

INSTRUCTION-HOW TO MAKE/PREPARE THE TARGET USiNG PREREQUISITES

```
gcc mod1.c -o ../../bins/mod1
bash-3.2$ make
gcc mod1.c -o ../../bins/mod1
bash-3.2$ make
gcc mod1.c -o ../../bins/mod1
bash-3.2$ ls ../../bins
mod1
```

PROBLEM

```
default:
        gcc mod1.c -o ../../bins/mod1
```

INCREMENTAL BUILD IS NOT WORKING

Improve Our Makefile

```
default:
        gcc mod1.c -o ../../bins/mod1
```

```
../../bins/mod1 : mod1.c
        gcc mod1.c -o ../../bins/mod1
```

TARGET : PREREQUISITE

INSTRUCTION-HOW TO MAKE/PREPARE THE TARGET USiNG PREREQUISITES

```
OUTPUT = ../../bins/mod1_bin

$(OUTPUT) : mod1.c
        gcc mod1.c -o $(OUTPUT)
```

Improve Our Makefile

```
OUTPUT = ../../bins/mod1_bin

$(OUTPUT) : mod1.c
        gcc mod1.c -o $(OUTPUT)
```

```
OUTPUT = ../../bins/mod1_bin
INPUT = mod1.c

$(OUTPUT) : $(INPUT)
        gcc $(INPUT) -o $(OUTPUT)
```

STILL BETTER WAY?

```
OUTPUT = ../../bins/mod1_bin
INPUT = mod1.c

$(OUTPUT) : $(INPUT)
        gcc $(INPUT) -o $(OUTPUT)
```

```
OUTPUT = ../../bins/mod1_bin
INPUT = mod1.c

$(OUTPUT) : $(INPUT)
        gcc $(<) -o $(@)
```

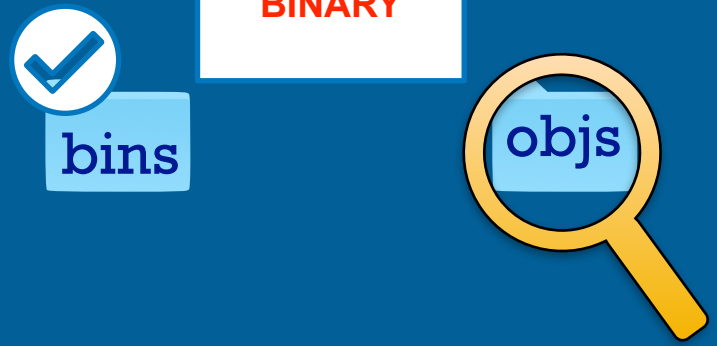
```
$(^)
```

```
$(@) = $@
$(<) = $<
$(^) = $^
```

Revist Module-1

MOD-1

BINARY



Update

MOD-1

BINARY


```
OUTPUT = ../../bins/mod1_bin
INPUT = mod1.c
OBJ1 = ../../objs/mod1.o

$(OUTPUT) : $(OBJ1)
        gcc $< -o $@

$(OBJ1): $(INPUT)
        gcc -c $< -o $@
```

PREVIOUS MAKEFILE

```
OUTPUT = ../../bins/mod1_bin
INPUT = mod1.c

$(OUTPUT) : $(INPUT)
        gcc $(<) -o $(@)
```

Revist Mod-1 Code

MOD-1

BINARY

bin obj

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

int main() {
   printf("%lf\n", sqrt(14.0));
   return 0;
}
```

mod1.c

-lc
libc.a

```
OUTPUT = ../../bins/mod1_bin
INPUT = mod1.c

OBJ1= ../../objs/mod1.o

$(OUTPUT) : $(OBJ1)
        gcc $< -o $@   -lm

$(OBJ1): $(INPUT)
        gcc -c $< -o $@
```

Makefile

libm.a

-lm

libm.a

Unused Variable

mod1.c

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

int main() {
   int a = 10;
   printf("%lf\n", sqrt(14.0));
   return 0;
}
```

-Wall -Werror

```
mod1.c:5:9: warning: unused variable 'a'
```

Makefile

```
OUTPUT = ../../bins/mod1_bin
INPUT = mod1.c

OBJ1= ../../objs/mod1.o

$(OUTPUT) : $(OBJ1)
        gcc $< -o $@  -lm

$(OBJ1): $(INPUT)
        gcc -Wall –c $< -o $@
```

Update Makefile(Compiler Flag)

`-Wall` `-Werror`

Makefile

```
OUTPUT = ../../bins/mod1_bin
INPUT = mod1.c
OBJ1= ../../objs/mod1.o

CFLAGS = -Wall -Werror

$(OUTPUT) : $(OBJ1)
        gcc $< -o $@ -lm

$(OBJ1): $(INPUT)
        gcc $(CFLAGS)-c $< -o $@
```

Update Makefile(Linker Flag)

`-lm` `-lzmq` `-lcurl` `-lxyz`

Makefile

```
OUTPUT = ../../bins/mod1_bin
INPUT = mod1.c
OBJ1= ../../objs/mod1.o
CFLAGS = -Wall -Werror
LDFLAGS = -lm

$(OUTPUT) : $(OBJ1)
        gcc $< -o $@  $(LDFLAGS)

$(OBJ1): $(INPUT)
        gcc $(CFLAGS) -c $< -o $@
```

Module-1 C++ IMPLEMENTATION

MOD-1

BINARY

mod1.cpp

```
#include <cstdio>
#include <cmath>

int main() {
   printf("%lf\n", sqrt(14.0));
   return 0;
}
```

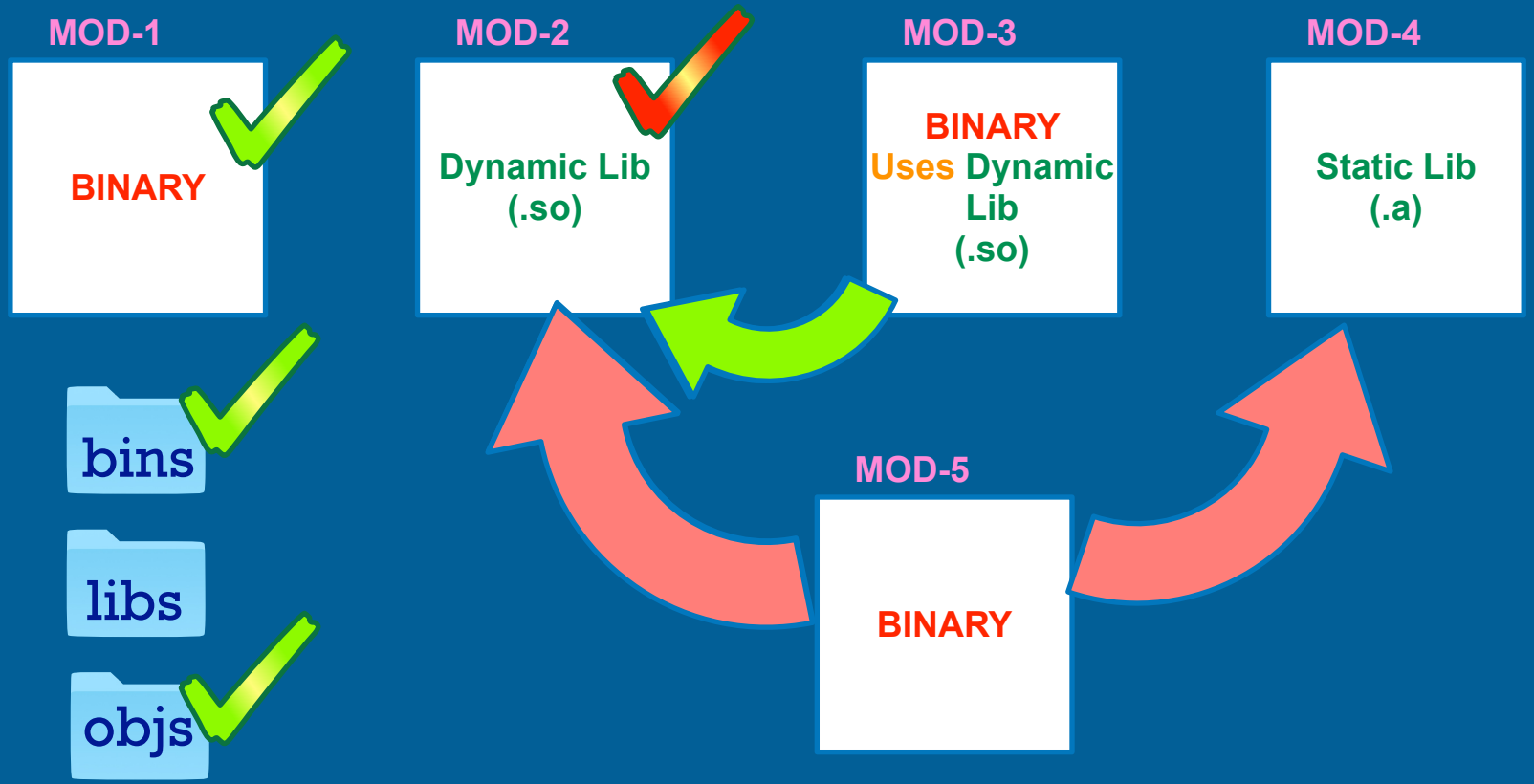
Makefile - Organize In Better Way

```
MODULE_NAME = mod1

TARGET_DIR = ../../bins
TARGET_NAME = mod1_bin
TARGET = $(TARGET_DIR)/$(TARGET_NAME)

OBJ_DIR = ../../objs
OBJ1 = $(OBJ_DIR)/$(MODULE_NAME).o

CFLAGS = -Wall -Werror
LDFLAGS = -lm

$(TARGET): $(OBJ1)
	g++ $< -o $@ $(LDFLAGS)

$(OBJ1): $(MODULE_NAME).c
	g++ $(CFLAGS) -c $< -o $@


clean:
	rm $(TARGET) $(OBJ1)
```

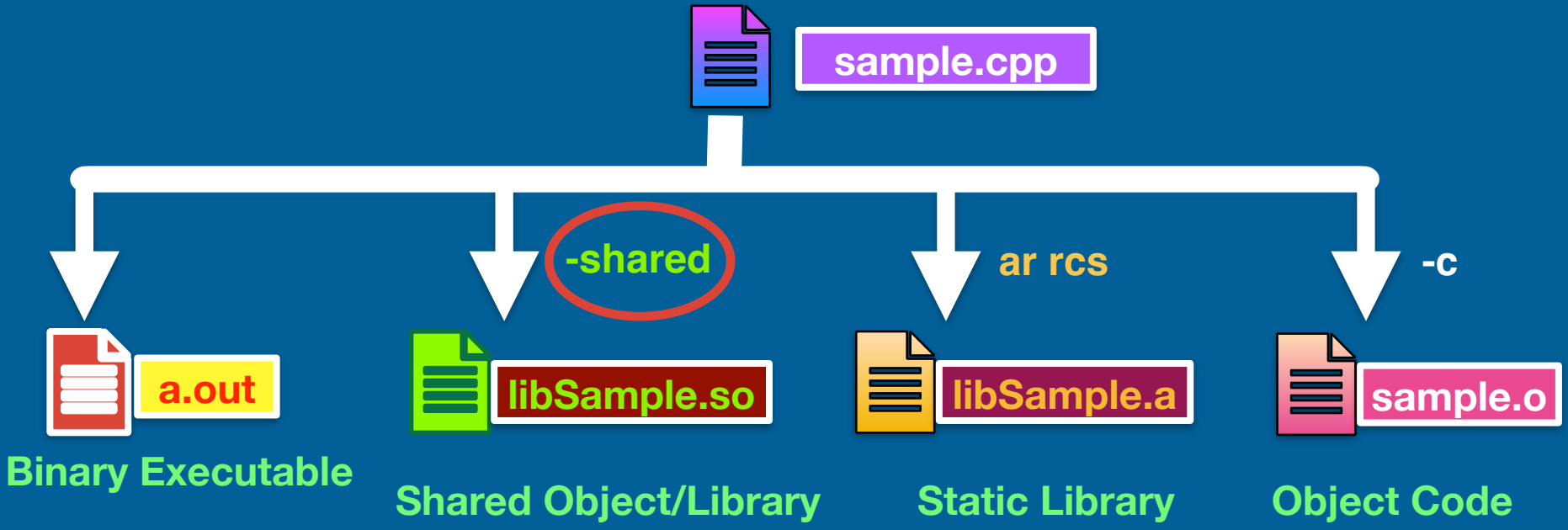
Makefile - Organize In Better Way

```
MODULE_NAME = mod1

TARGET_DIR = ../../bins
TARGET_NAME = mod1_bin
TARGET = $(TARGET_DIR)/$(TARGET_NAME)

OBJ_DIR = ../../objs
OBJ1 = $(OBJ_DIR)/$(MODULE_NAME).o

CFLAGS = -Wall -Werror
LDFLAGS = -lm

$(TARGET): $(OBJ1)
	g++ $< -o $@ $(LDFLAGS)

$(OBJ1): $(MODULE_NAME).cpp
	g++ $(CFLAGS) -c $< -o $@


clean:
	rm $(TARGET) $(OBJ1)
```

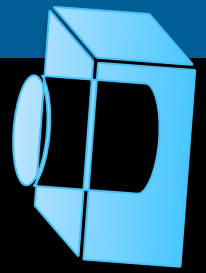
REVIEW - Requirement



Revisit - Possible Outputs



Mod-2 Makefile-2 (.so)



```
MODULE_NAME = mod2
TEST_MODULE = test_mod2
TEST_BIN_DIR = ../../test_bins
TEST_TARGET = $(TEST_BIN_DIR)/$(TEST_MODULE)

TARGET_DIR = ../../libs
TARGET_NAME = lib$(MODULE_NAME).so
TARGET = $(TARGET_DIR)/$(TARGET_NAME)

OBJ_DIR = ../../objs
OBJ1 = $(OBJ_DIR)/$(MODULE_NAME).o

CFLAGS = -fPIC -Wall -Werror
LDFLAGS =

$(TARGET): $(OBJ1)
        g++ -shared $< -o $@ $(LDFLAGS)

$(OBJ1): $(MODULE_NAME).cpp
        g++ $(CFLAGS) -c $< -o $@

test:  $(TEST_MODULE).cpp $(TARGET)
        g++ $< -o $(TEST_TARGET) -L$(TARGET_DIR) -l$(MODULE_NAME)
clean:
        rm $(TARGET) $(OBJ1) $(TEST_TARGET)
build_dir:
        mkdir -p $(TEST_BIN_DIR)
```

When -fPIC?

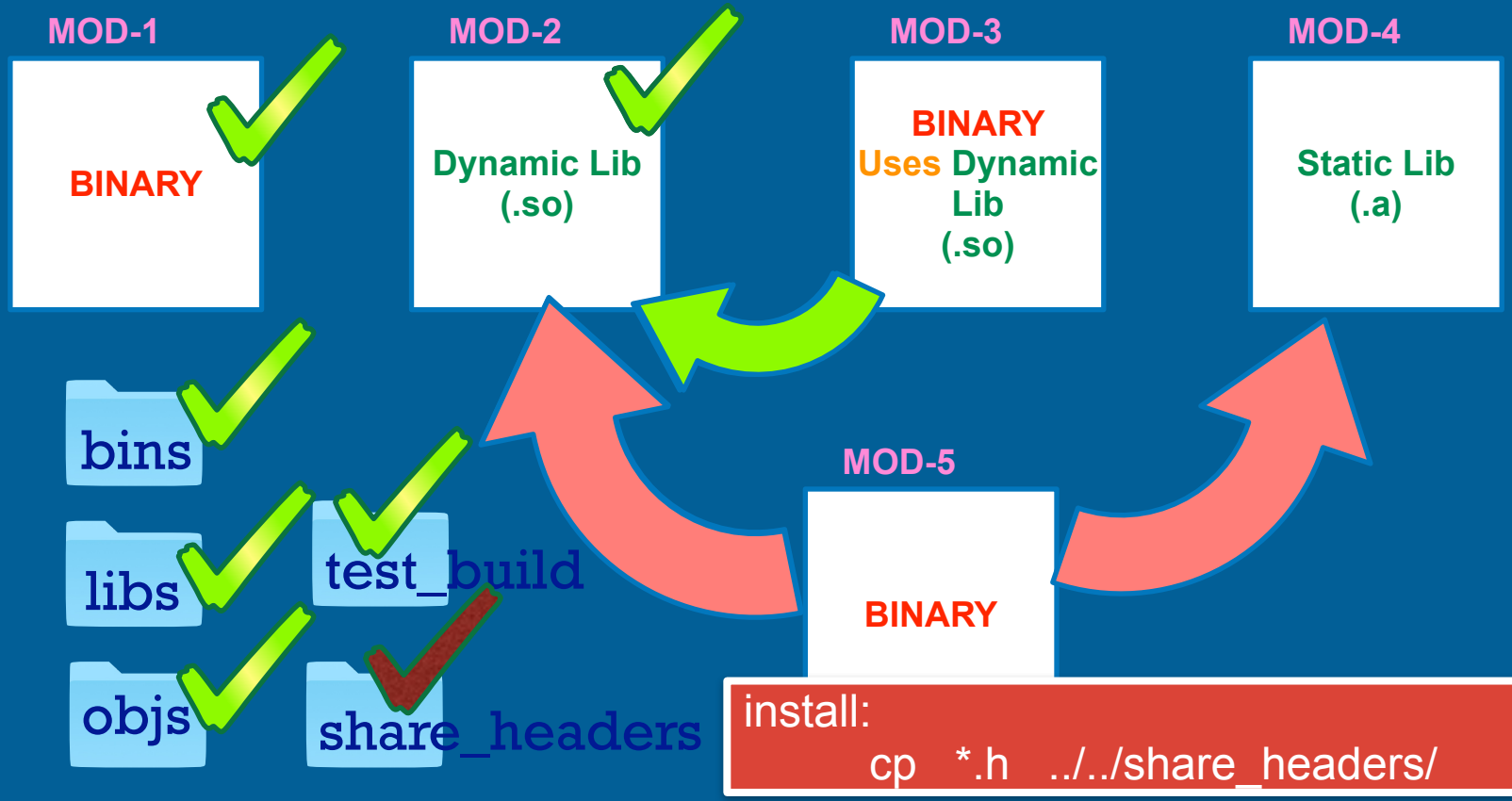
- Compile or Linking stage?
- PROOF

When -fPIC?

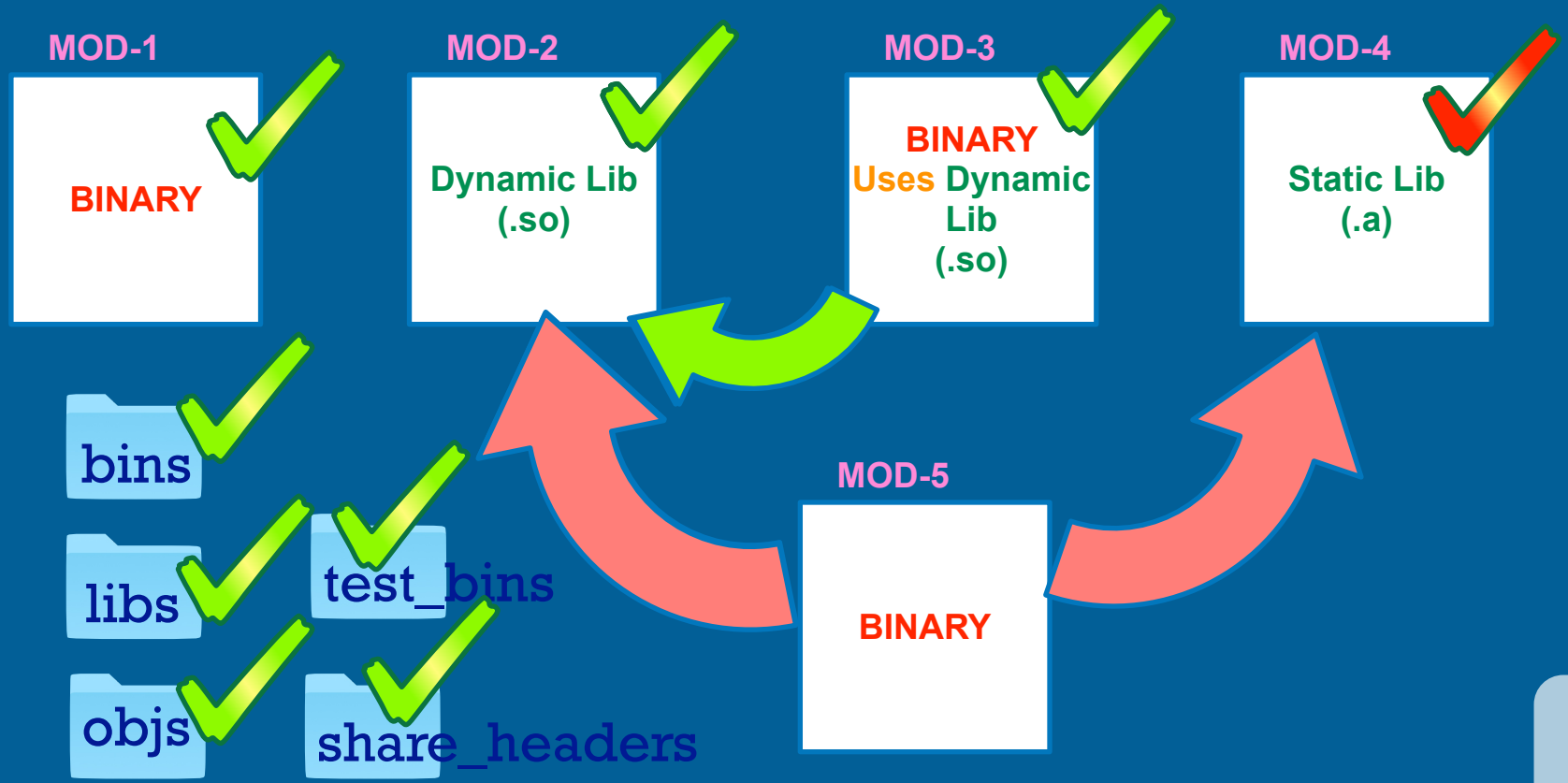
g++ or gcc compiler

clang++ or clang compiler

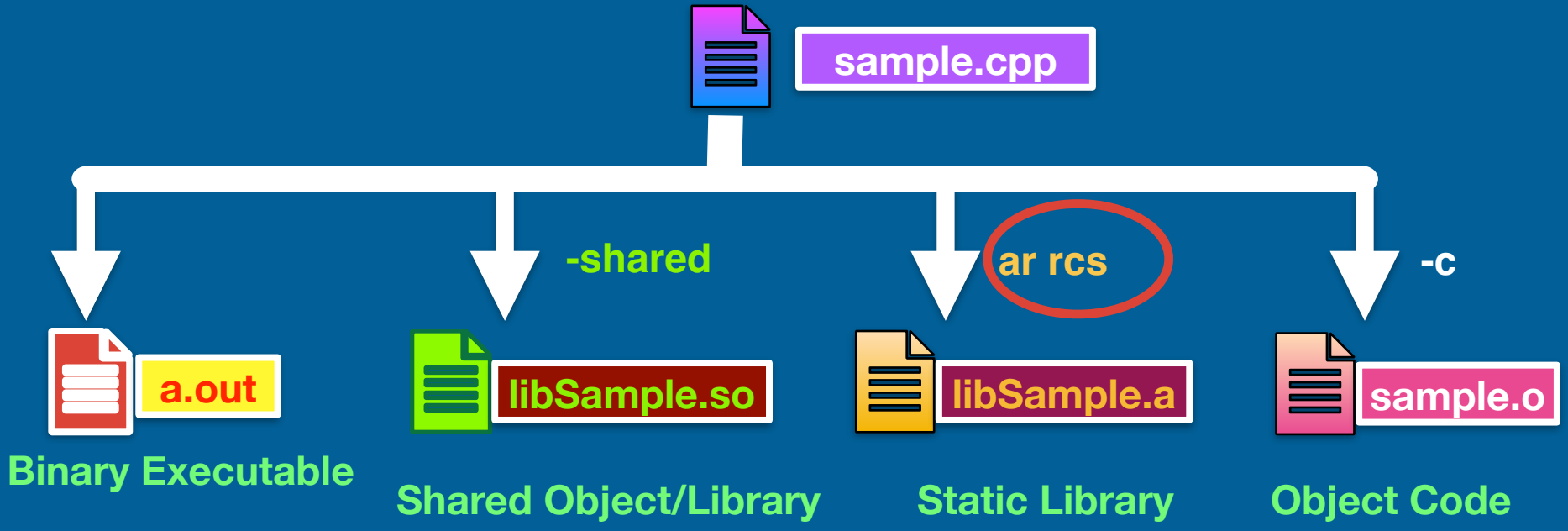
REVIEW - Requirement



REVIEW - Requirement



Revisit - Possible Outputs



ar rcs libsample.a sample.o

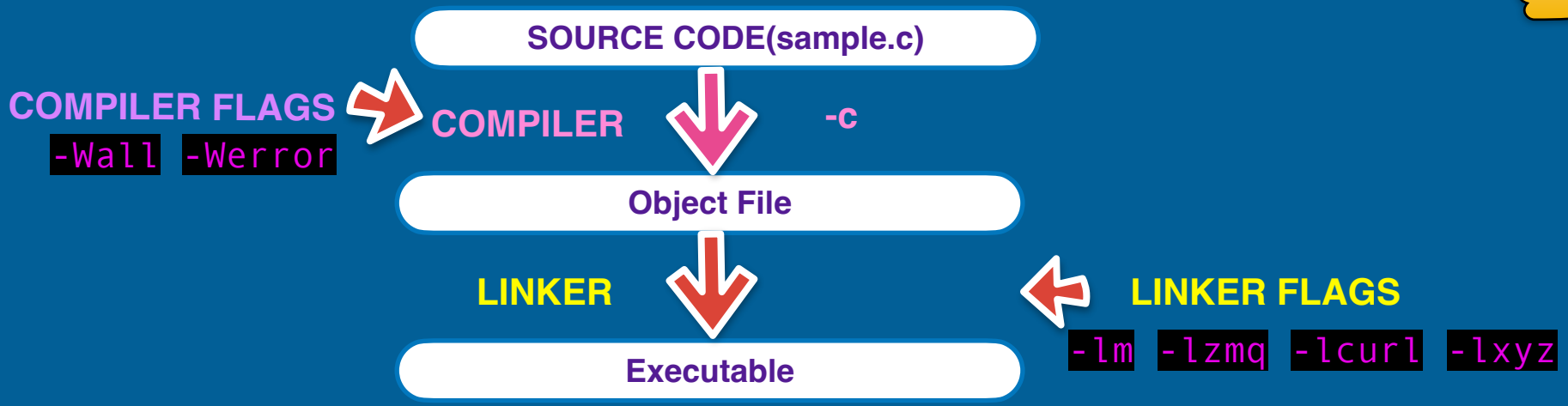
Mod-4 Makefile-4 (.a)

```
MODULE_NAME = mod4

TARGET_DIR = ../../libs
TARGET_NAME = lib$(MODULE_NAME).a
TARGET = $(TARGET_DIR)/$(TARGET_NAME)

OBJ_DIR = ../../objs
OBJ1 = $(OBJ_DIR)/$(MODULE_NAME).o

CFLAGS = -fPIC -Wall -Werror
LDFLAGS =

$(TARGET): $(OBJ1)
	ar rcs $@ $^

$(OBJ1): $(MODULE_NAME).cpp
	g++ $(CFLAGS) -c $< -o $@

clean:
	rm $(TARGET) $(OBJ1)
```

ASSIGNMENT

- Write test code to test mod4.a
- Makefile to generate test binary.

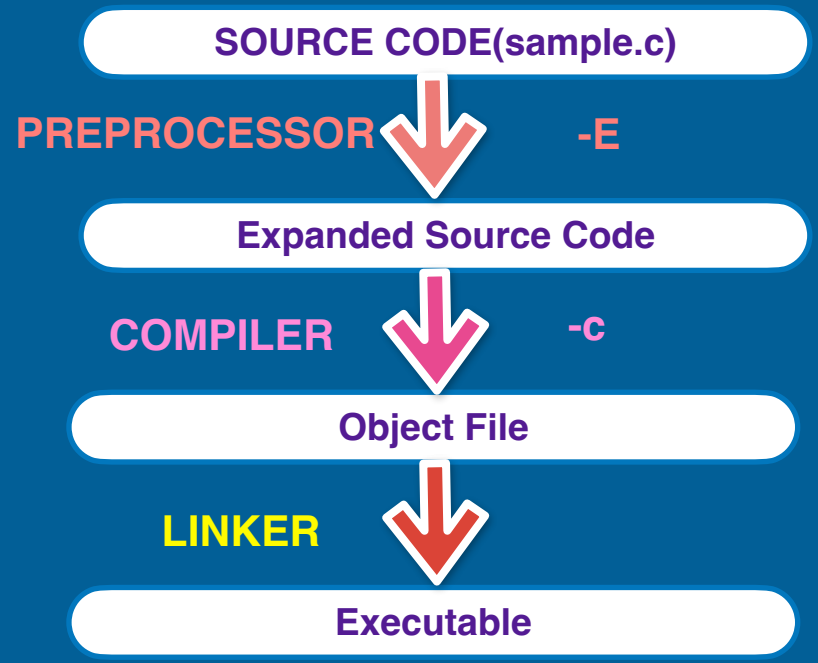
Remember - For Makefile




```
gcc $(CFLAGS) -c $< -o $@
gcc $< -o $@ $(LDFLAGS)
```

What abut passing a MACRO from out side of our program?

What About Macro?



-DMY_NAME=\"Subrat\"

macro.c

```c
#include <stdio.h>

int main() {

    printf("%s\n", MY_NAME);
}
```

```
└─➤  gcc macro.c
macro.c:5:20: error: use of undeclared identifier
'MY_NAME'
    printf("%s\n", MY_NAME);
```

```
└─➤ gcc -DMY_NAME=\"Subrat\" macro.c
```

Introducing -std flag

WAP TO DEMONASTRATE MULTITHREADING

= VS :=

Lazy Initialization

Instant Initialization

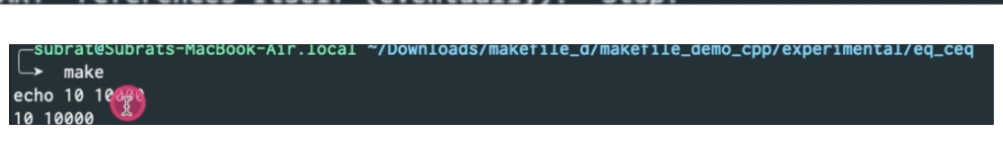
RECURSIVE Variable error

= VS :=

```
1 makefile
VAR1=10
VAR2=30
VAR3=40

VAR1=$(VAR1) 10000

deafult:
    echo $(VAR1)

~
NORMAL  makefile                                    66%   6:1
"makefile" 9L, 69C written
```

```
VAR1 := $(VAR1) 10000
```

```
subrat@Subrats-MacBook-Air.local ~/Downloads/makefile_d/makefile_demo_cpp/experimental/eq_ceq
➜  make
echo  10000
10000
subrat@Subrats-MacBook-Air.local ~/Downloads/makefile_d/makefile_demo_cpp/experimental/eq_ceq
➜  make
makefile:5: *** Recursive variable 'VAR1' references itself (eventually).  Stop.
subrat@Subrats-MacBook-Air.local
➜
```



Makefile Template

- Add cross compilation feature
- Better way of accessing bins, libs, share_headers directories
- Copy headers to shared headers during building for .so and .a

Thing to remember

No space/ space

Space

[TARGET] : [Dependency - 1] [Dependency - 2] .....

(Linking Stage)

g++ [Dependency - 1] [Dependency - 2] -o [TARGET]

Tab/Space ✓ ✗

$<

$^

$@

```
g++ $< -o $@ $(LDFLAGS)
```

Dependency.o

(Compiling Stage)

Dependency1 : [Dependency-1's DEPENDENCIES]

g++ -c [Dependency-1's DEPENDENCIES] -o Dependency1

```
g++ $(CFLAGS) -c $< -o $@
```

Dependency2 : [Dependency-2's DEPENDENCIES]

g++ -c [Dependency-2's DEPENDENCIES] -o Dependency2

PROJECT

UI CONTROLLER

BINARY

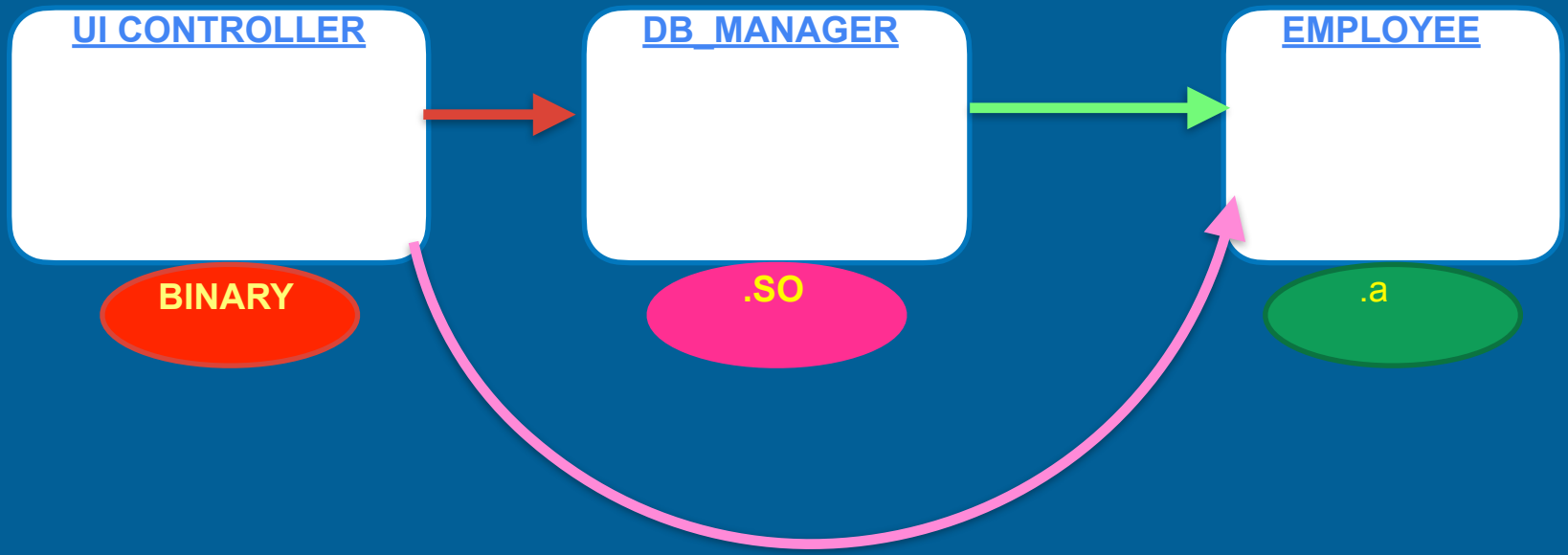
DB_MANAGER

.SO

EMPLOYEE

.a

MOUDLES



SOURCE CODE DEMO

BUILD & RUN

- Use the sample makefiles & modify as per our requirement
- Fix some issues on the fly

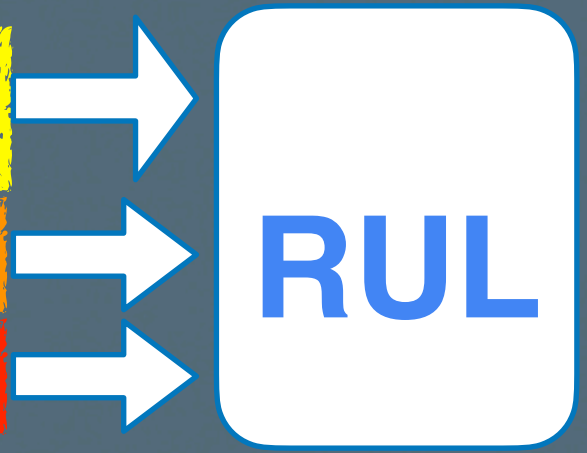
STEPS:

- Employee ======>.A
- DB_MGR. =======> .SO
- UI. ==============> BINARY EXECUTABLE

One Makefile To CALL OTHER MAKEFILES

- How to pass environment variable to Makefile - Use **export** in **terminal** from where we are going to do make
- How to pass a variable from one Makefile to other Makefile - Use **export** inside **Makefile**

```
    as shown below
DEP_LIBS = -L$(LIBRARY_DIR) -lemp

# RPATH IS USED FOR LINKING USER DEFINED LIBS IN SPECIFIC PATH DURING
# BUILDING THE MODULE.It is needed for the User of the .so. \
    # Unit test binary may use it.
#RPATH="-Wl,-rpath,$(TARGET_DIR):$(TARGET_DIR)/3rd_party_lib"
RPATH="-Wl,-rpath,$(TARGET_DIR)"

LDFLAGS = $(DEP_LIBS) $(RPATH)

all: $(TARGET)

$(TARGET): $(ALL_OBJS)
        $(LDXX) -shared -o $@ $^ $(LDFLAGS)
        make install

$(OBJ_1): $(SOURCE_1).cpp
        $(CXX) $(CCFLAGS) -o $@ -c $<

$(OBJ_2): $(SOURCE_2).cpp
        $(CXX) $(CCFLAGS) -o $@ -c $<

build_dir:
        @echo Creating object directory if not exist
Makefile
```



Using Function & Patterns

```
SRCS := $(wildcard *.cpp)

BINS := $(SRCS:%.cpp=%)
OBJS := $(SRCS:%.cpp=%.o)
```

substitution reference

Source file	OBJ file	Binary
Sample.cpp	Sample.o	Sample
Sample1.cpp	Sample1.o	Sample1

Exercise

WRITE A MAKEFILE
Build all C++ Source Files

1. Detect all the file with extension .cpp
2. Store inside a variable SRCS
3. Determine output binary file name
4. Store inside a variable BINS
5. Write rules to build all the source files to corresponding target.

How To Generate Object files?

Detect all Source Files(.cpp / .c)

```
SRCS := $(wildcard *.cpp *.c)
```

CODE

```
SRCS := $(wildcard *.cpp)
BINS := $(SRCS:%.cpp=%)
OBJS := $(SRCS:%.cpp=%.o)

all: $(BINS) $(OBJS)

%: %.o
	@echo "Generating binaries"
	g++ $< -o $@
%.o : %.cpp
	@echo "Generating object files"
	g++ -c $< -o $@

clean:
	rm $(BINS) $(OBJS)
```

Deep Look Into makefile

Modify Makefile of db_mgr to detect .cpp files automatically.

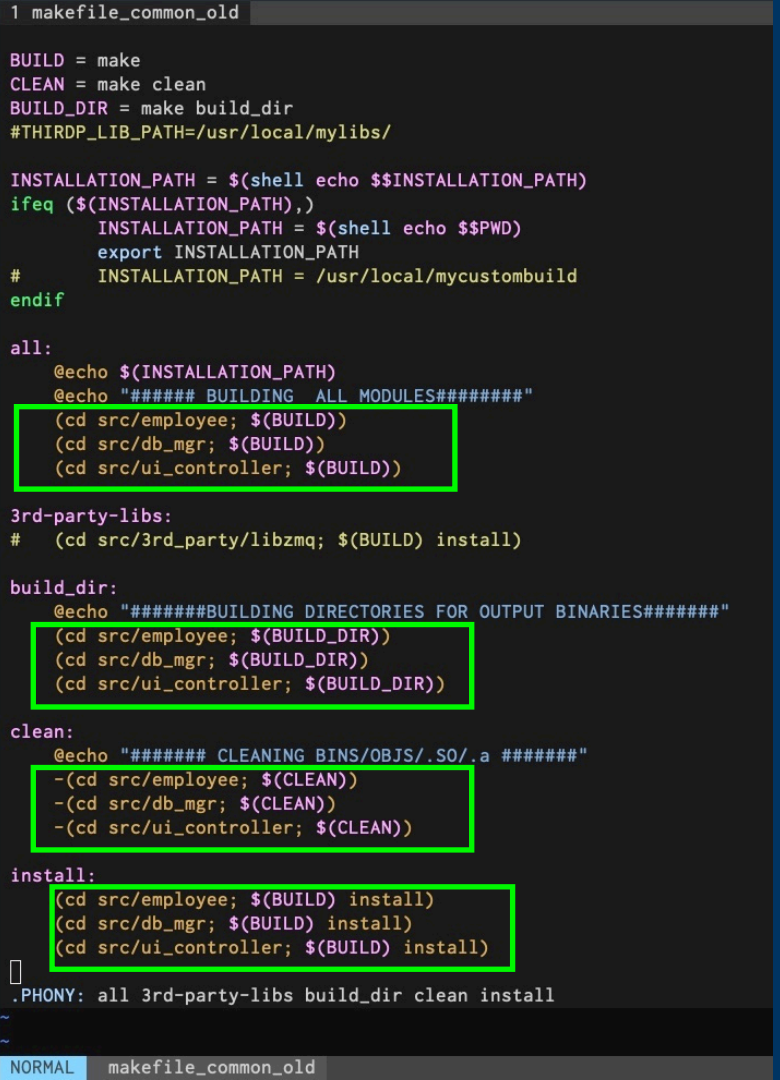
ASSIGNMENT

Modify Makefile of EMPLOYEE, UI_CONTROLLER to detect source files automatically.

Modify Our Common Makefile

Modify makefile present in project root folder to Call other makefiles

- Patterns and functions ✓
- $ Vs $$? ✓
- Write our own function and call it. ✓



WHAT WE ARE DOING?

1. Changing the directory.
2. Running make with proper argument.

Ex:

make
make clean
make build_dir

make -C DIRECTORY_NAME TARGET_NAME

WRITE A FUNCTION

1. Detect all modules/directories in src folder.
2. Change the directory.
3. Run make with proper argument.

Function

WRITE FUNCTION DEFINITION

```
define makeallmodules
    # INSTRUCTIONS TO EXECUTE
                $(1)          $(2)
endef
```

CALL FUNCTION

```
$( call makeallmodules , PARAM1, PARAM2 )
```

```
1 makefile_common_old

BUILD = make
CLEAN = make clean
BUILD_DIR = make build_dir
#THIRDP_LIB_PATH=/usr/local/mylibs/

INSTALLATION_PATH = $(shell echo $$INSTALLATION_PATH)
ifeq ($(INSTALLATION_PATH),)
        INSTALLATION_PATH = $(shell echo $$PWD)
        export INSTALLATION_PATH
#       INSTALLATION_PATH = /usr/local/mycustombuild
endif

all:
    @echo $(INSTALLATION_PATH)
    @echo "###### BUILDING  ALL MODULES########"
    (cd src/employee; $(BUILD))
    (cd src/db_mgr; $(BUILD))
    (cd src/ui_controller; $(BUILD))

3rd-party-libs:
#   (cd src/3rd_party/libzmq; $(BUILD) install)

build_dir:
    @echo "#######BUILDING DIRECTORIES FOR OUTPUT BINARIES#######"
    (cd src/employee; $(BUILD_DIR))
    (cd src/db_mgr; $(BUILD_DIR))
    (cd src/ui_controller; $(BUILD_DIR))

clean:
    @echo "####### CLEANING BINS/OBJS/.SO/.a #######"
    -(cd src/employee; $(CLEAN))
    -(cd src/db_mgr; $(CLEAN))
    -(cd src/ui_controller; $(CLEAN))

install:
    (cd src/employee; $(BUILD) install)
    (cd src/db_mgr; $(BUILD) install)
    (cd src/ui_controller; $(BUILD) install)

.PHONY: all 3rd-party-libs build_dir clean install
~
~
NORMAL  makefile_common_old
```

WRITE A FUNCTION

1. Detect all modules/directories in src folder.
2. Change the directory.
3. Run make with proper argument.

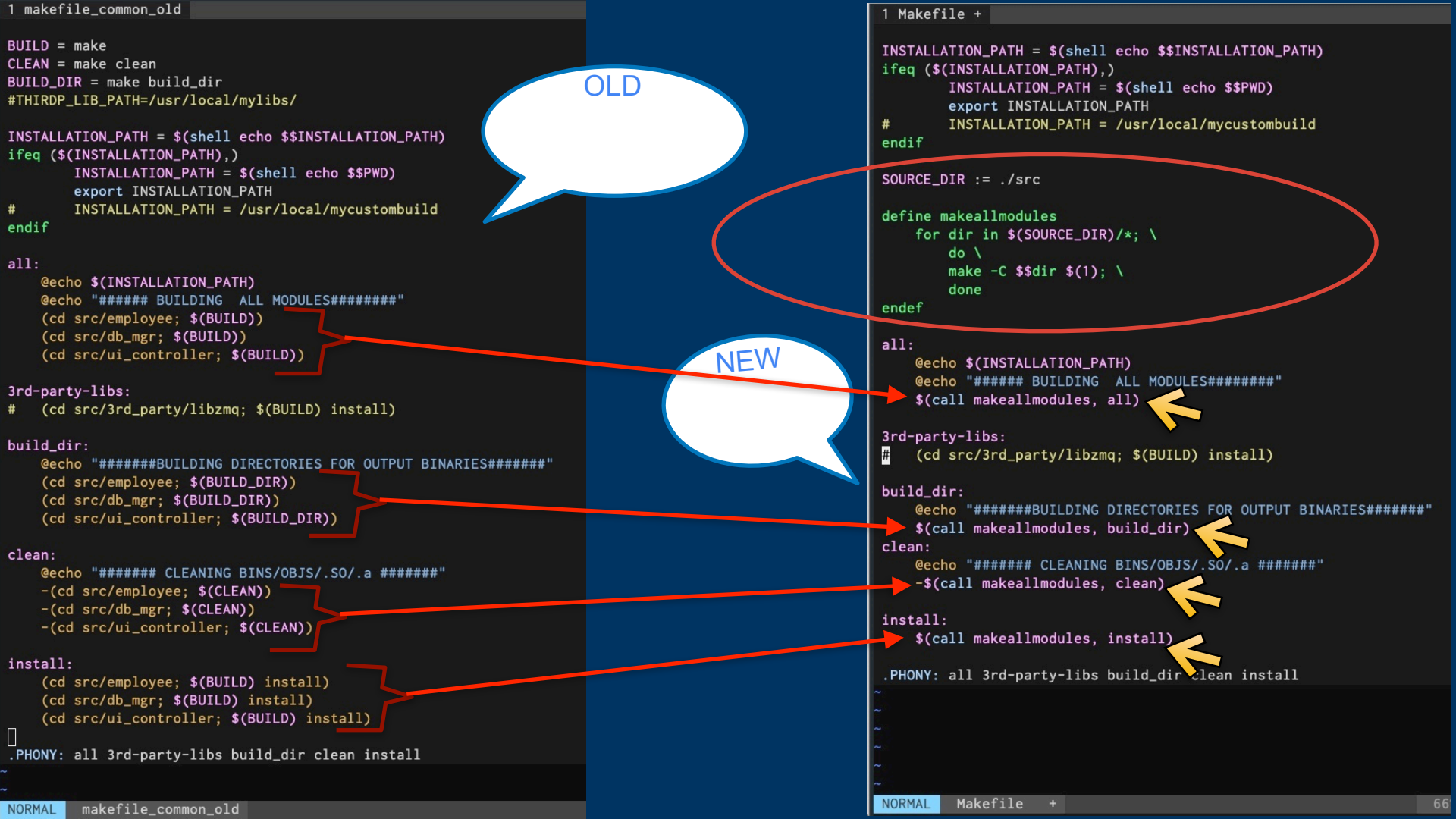
```
SOURCE_DIR := ./src

define makeallmodules
    for dir in $(SOURCE_DIR)/*; do make -C $$dir $(1); done
endef
```

OR

```
SOURCE_DIR := ./src

define makeallmodules
    for dir in $(SOURCE_DIR)/*; \
        do \
        make -C $$dir $(1); \
        done
endef
```

CALL FUNCTION

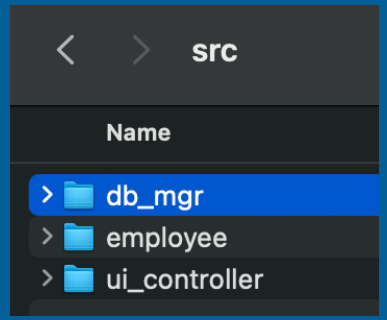
```
$(call makeallmodules, all)
$(call makeallmodules, build_dir)
$(call makeallmodules, clean)
```


1 makefile_common_old
BUILD = make
CLEAN = make clean
BUILD_DIR = make build_dir
#THIRDP_LIB_PATH=/usr/local/mylibs/
INSTALLATION_PATH = $(shell echo $$INSTALLATION_PATH)
ifeq ($(INSTALLATION_PATH),)
INSTALLATION_PATH = $(shell echo $$PWD)
export INSTALLATION_PATH
# INSTALLATION_PATH = /usr/local/mycustombuild
endif
all:
@echo $(INSTALLATION_PATH)
@echo "###### BUILDING ALL MODULES########"
(cd src/employee; $(BUILD))
(cd src/db_mgr; $(BUILD))
(cd src/ui_controller; $(BUILD))
3rd-party-libs:
(cd src/3rd_party/libzmq; $(BUILD) install)
build_dir:
@echo "#######BUILDING DIRECTORIES FOR OUTPUT BINARIES#######"
(cd src/employee; $(BUILD_DIR))
(cd src/db_mgr; $(BUILD_DIR))
(cd src/ui_controller; $(BUILD_DIR))
clean:
@echo "####### CLEANING BINS/OBJS/.SO/.a #######"
-(cd src/employee; $(CLEAN))
-(cd src/db_mgr; $(CLEAN))
-(cd src/ui_controller; $(CLEAN))
install:
(cd src/employee; $(BUILD) install)
(cd src/db_mgr; $(BUILD) install)
(cd src/ui_controller; $(BUILD) install)
.PHONY: all 3rd-party-libs build_dir clean install
NORMAL makefile_common_old
OLD
NEW
1 Makefile +
INSTALLATION_PATH = $(shell echo $$INSTALLATION_PATH)
ifeq ($(INSTALLATION_PATH),)
INSTALLATION_PATH = $(shell echo $$PWD)
export INSTALLATION_PATH
# INSTALLATION_PATH = /usr/local/mycustombuild
endif
SOURCE_DIR := ./src
define makeallmodules
for dir in $(SOURCE_DIR)/*; \
do \
make -C $$dir $(1); \
done
endef
all:
@echo $(INSTALLATION_PATH)
@echo "###### BUILDING ALL MODULES########"
$(call makeallmodules, all)
3rd-party-libs:
(cd src/3rd_party/libzmq; $(BUILD) install)
build_dir:
@echo "#######BUILDING DIRECTORIES FOR OUTPUT BINARIES#######"
$(call makeallmodules, build_dir)
clean:
@echo "####### CLEANING BINS/OBJS/.SO/.a #######"
-$(call makeallmodules, clean)
install:
$(call makeallmodules, install)
.PHONY: all 3rd-party-libs build_dir clean install
NORMAL Makefile +

PROBLEM

1. Need to run make 2 times.
2. Bcz db_mgr is depending upon libemp.a.
3. During build process, db_mgr is building first while libemp.a is not created.
4. So, how to get rid of this issue?

```
SOURCE_DIR := ./src

define makeallmodules
    for dir in $(SOURCE_DIR)/*; \
      do \
      make -C $$dir $(1); \
      done
endef
```



Solution

Add the dependency in Makefile of db_mgr

Demonstrate in LINUX

Copy the code with makefile to Linux box
Demonstrate without any modification it is working

FOR C PROGRAMMERS

Write C programs

In makefile:

Replace g++ with gcc
Replace $(CXX) with $(CC)
Replace $(LDXX) with $(LD)
Replace *.cpp with *.c(WILDCARD function)


Thank You!
All The Best