ROBOTICS CORNER

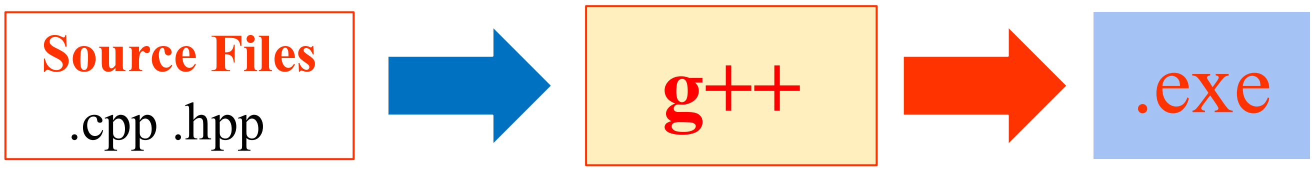
# Build Tools

Mohamed Saied

# Motivation

## Why should we learn build tools ?

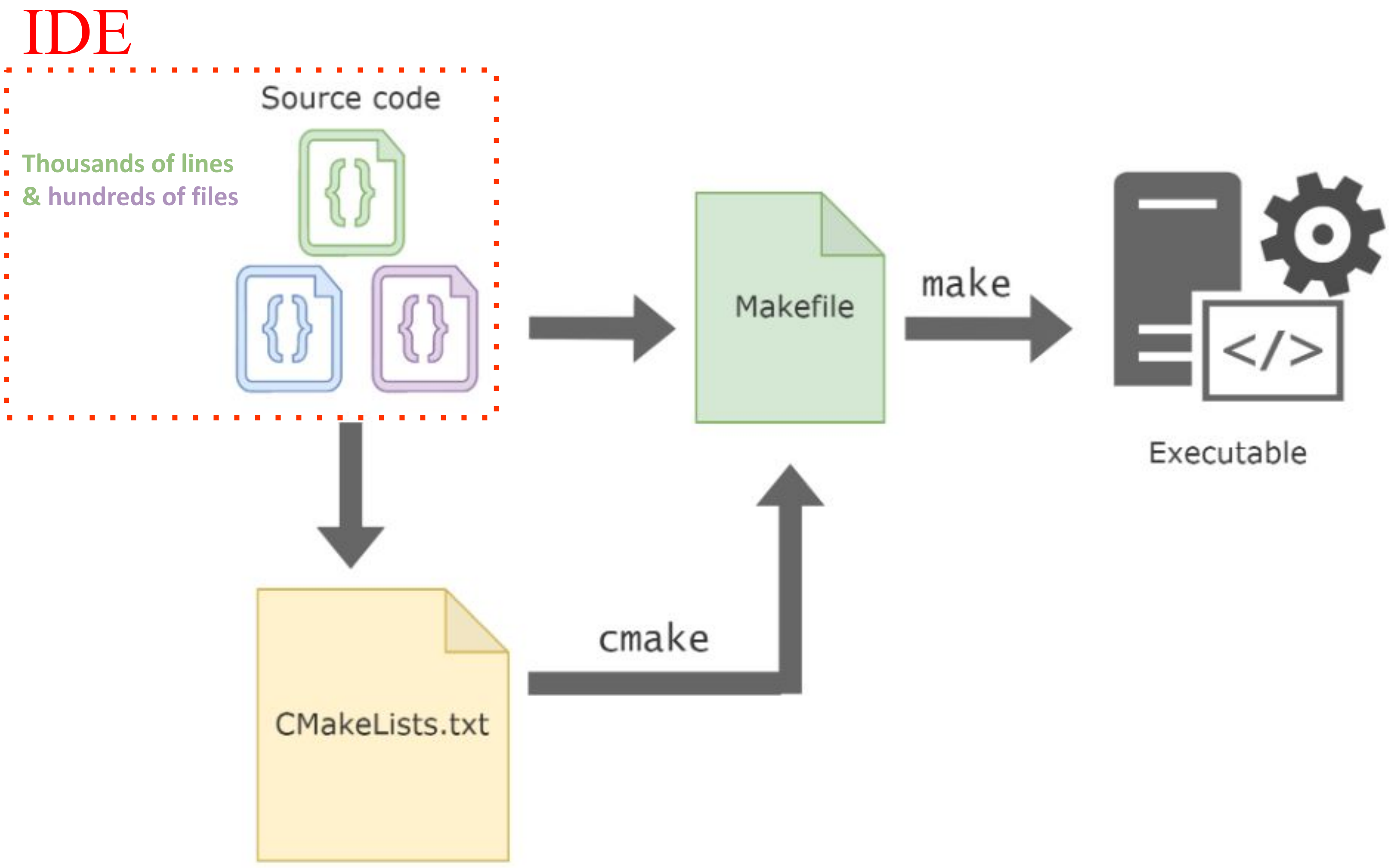
➡ Previous approach ➡ Build Small Programs



```
g++ -c hello World.cpp hello.cpp
g++ -o main.exe helloWorld.o hello.o
```

***Build** and **link** with single lines using **manual** commands up to **10 files***

# Building SW In Industry



## Motivation – continued

- Problems:
  - Long files are harder to manage

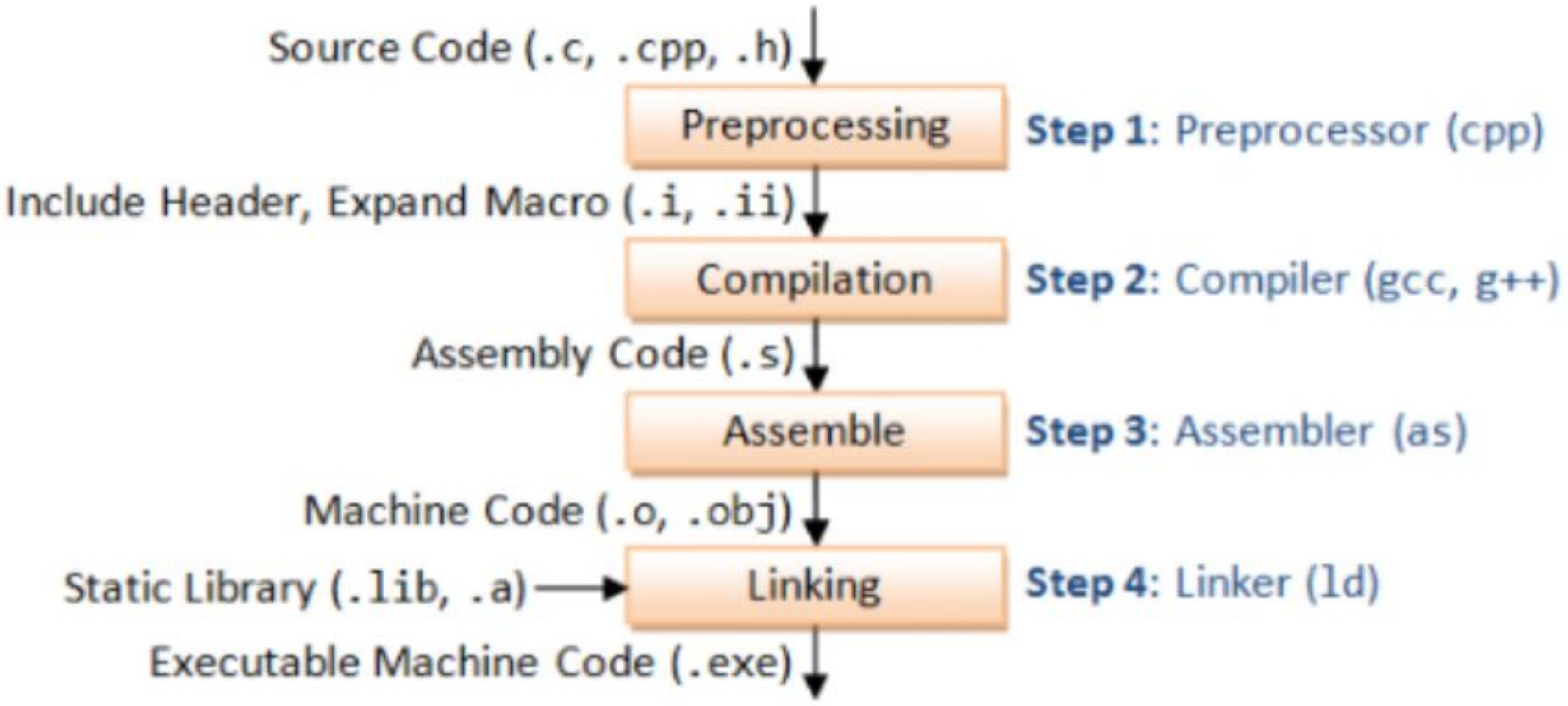
    (for both programmers and machines)
  - Every change requires compilation of all constituent files?
  - Many programmers cannot modify the

    same file simultaneously
    - Large projects are not implemented in a single module/file

## Motivation – continued

- Solution : **Separation of Concerns**
- Targets:
  - Proper and optimal division of components
  - Minimum compilation when something is changed
  - Easy maintenance of project structure, dependencies and creation

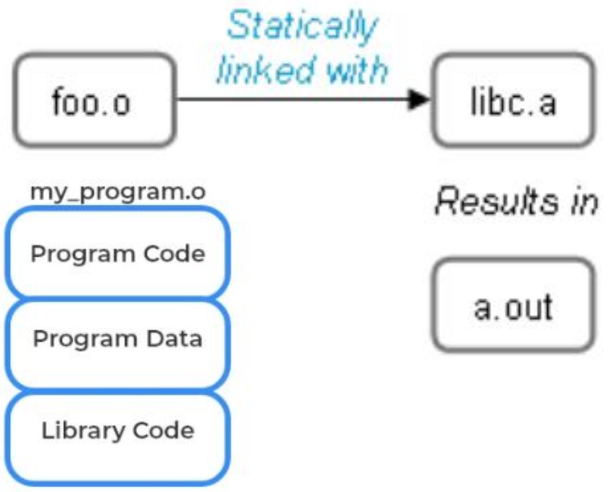
# Recall Build Process


Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

# Recall Build Process Cont.

## Static Linking

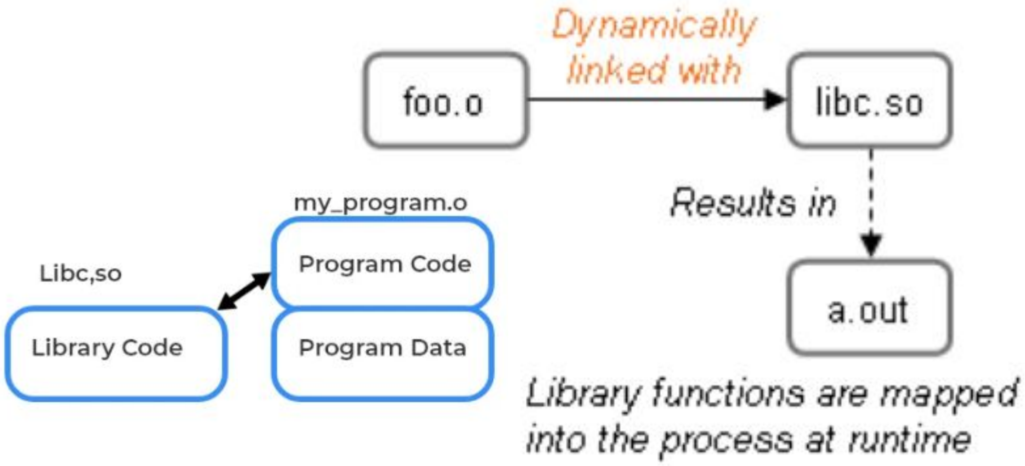
Static linking combines your work with the library into one binary.



The executable is statically linked because a copy of the library is physically part of the executable.

## Dynamic Linking

Dynamic linking creates a combined work at runtime.



The executable is dynamically linked because it contains filenames that enable the loader to find the program's library references at runtime.

## Project maintenance

- Performed in Unix by the `make` utility

  **make utility** is a tool used to automate the process of building programs from source code.

  GNU Make is the most widely used version of the make utility

- A `makefile` is a file (script) containing:
  - Project structure (files, dependencies)
  - Instructions for creation of object code, executables, other tasks

- Note: A `makefile` script is not limited to C/C++ programs

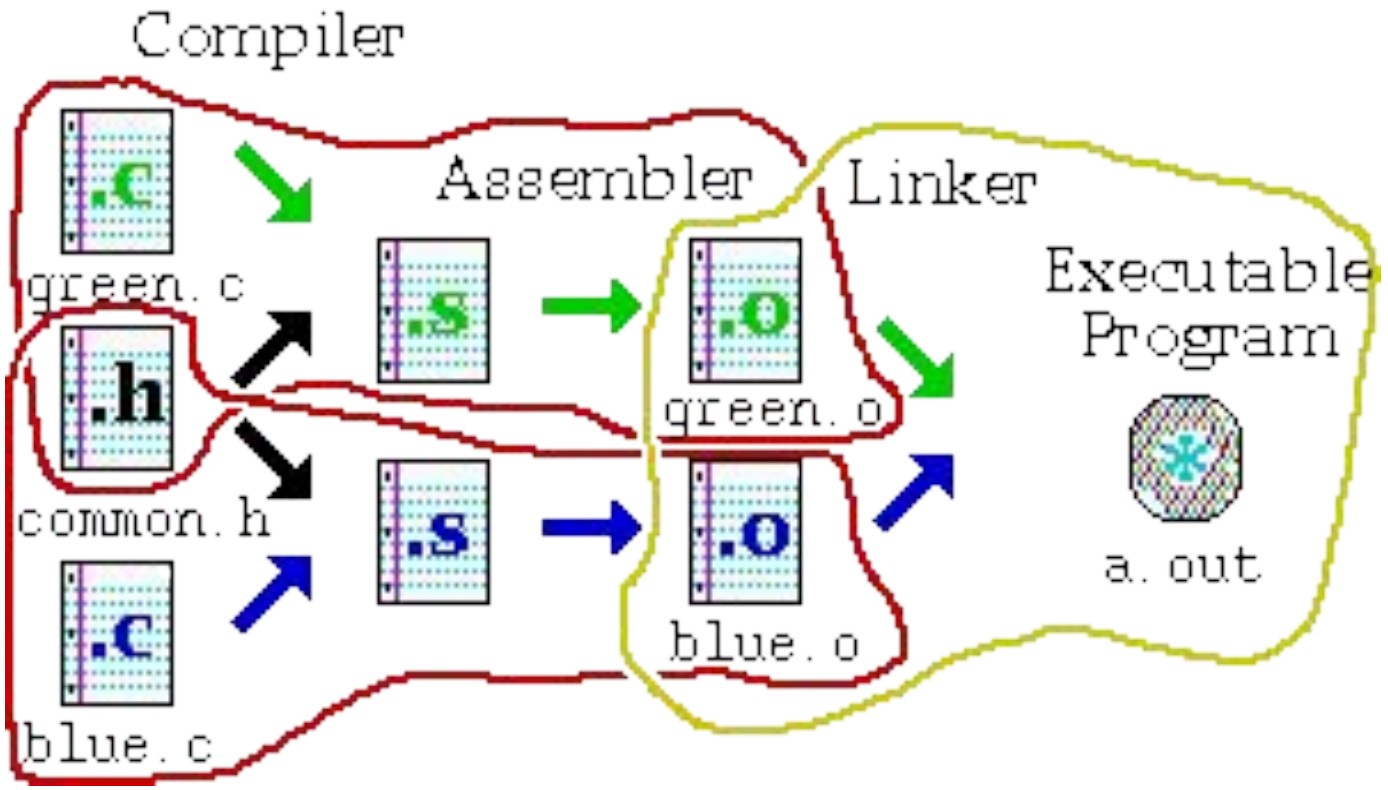
## Project structure

- Project structure and dependencies can be represented as a DAG (= Directed Acyclic Graph)
- Example :
  - Date class:
    - contains 3 files
  - Date.h, Date.cpp, TestDate.cpp
  - Date.h included in both .cpp files
  - Executable should be named testdate

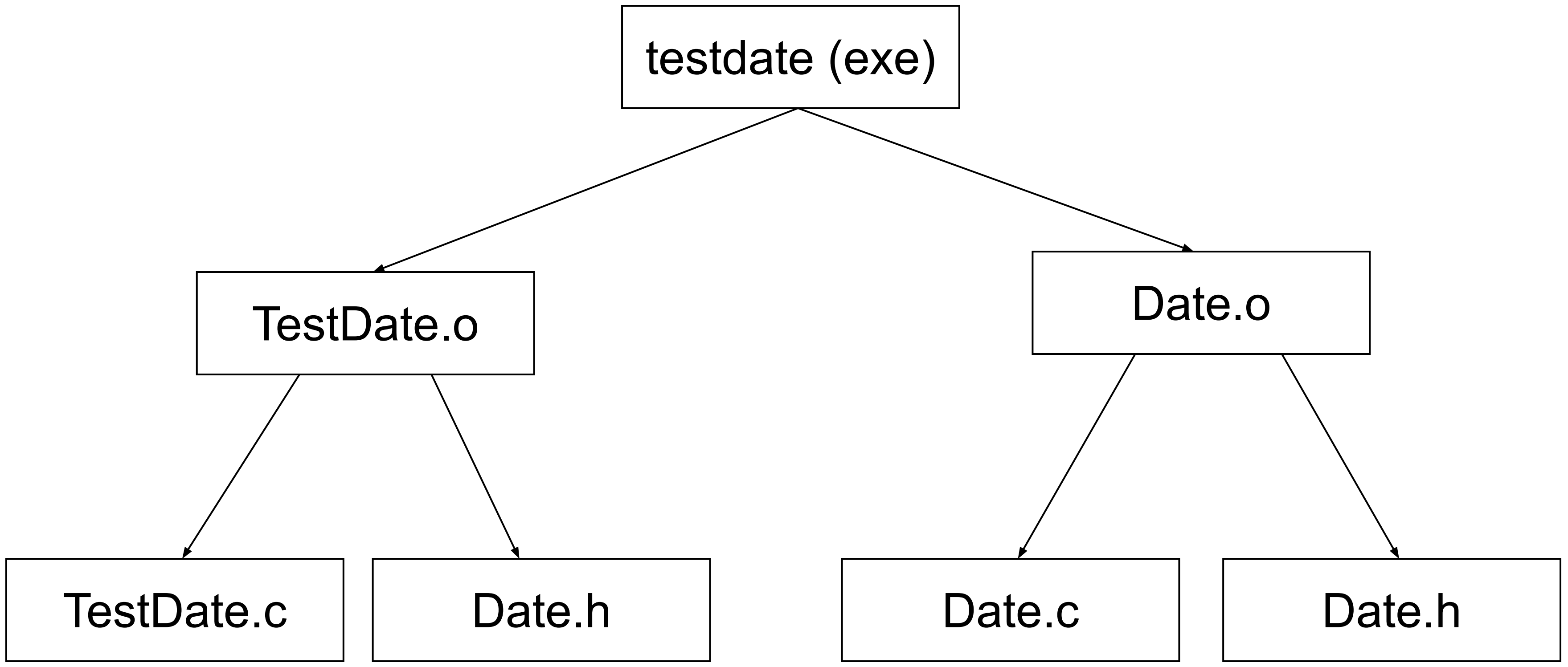
.cpp .h

# Makefiles

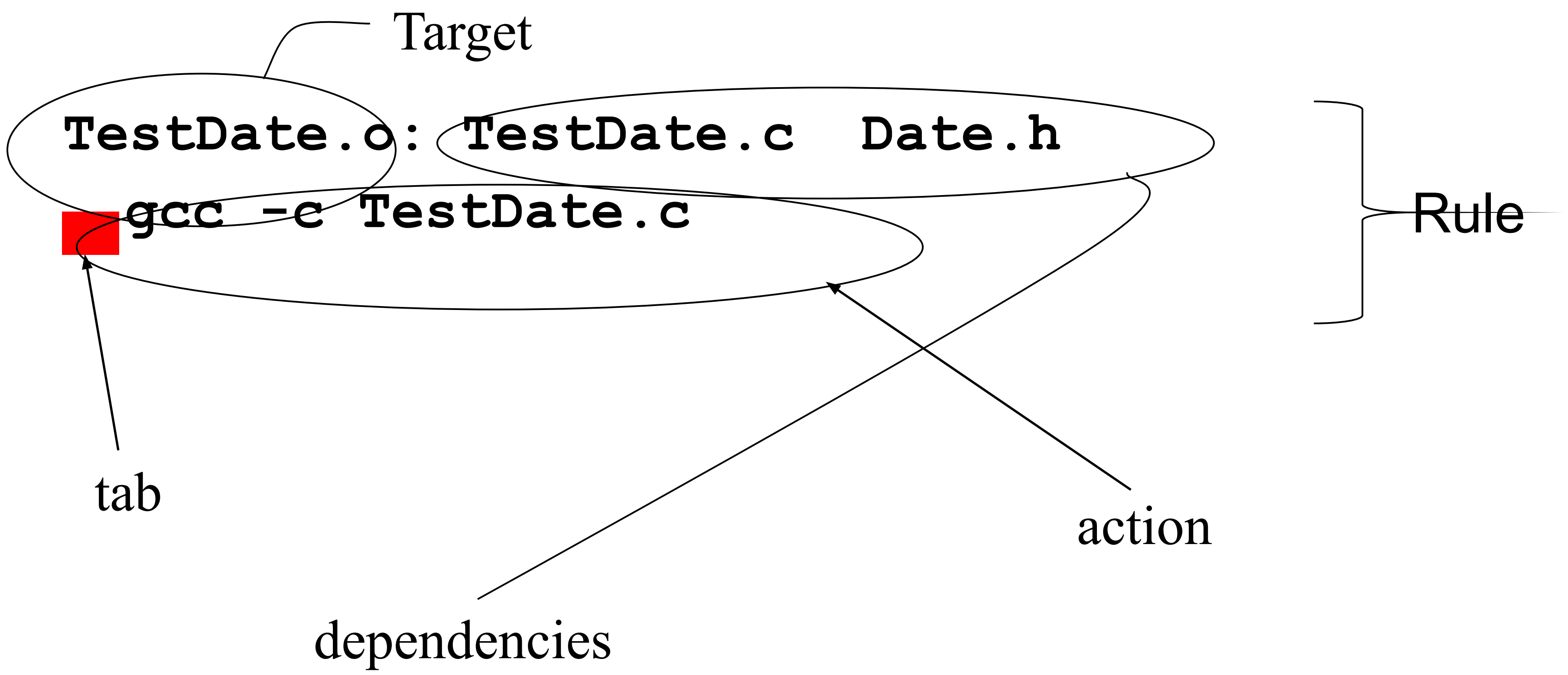
- Provide a way for separate compilation.
- Describe the dependencies among the project files.
- The `make` utility.



# DAG



# Rule syntax

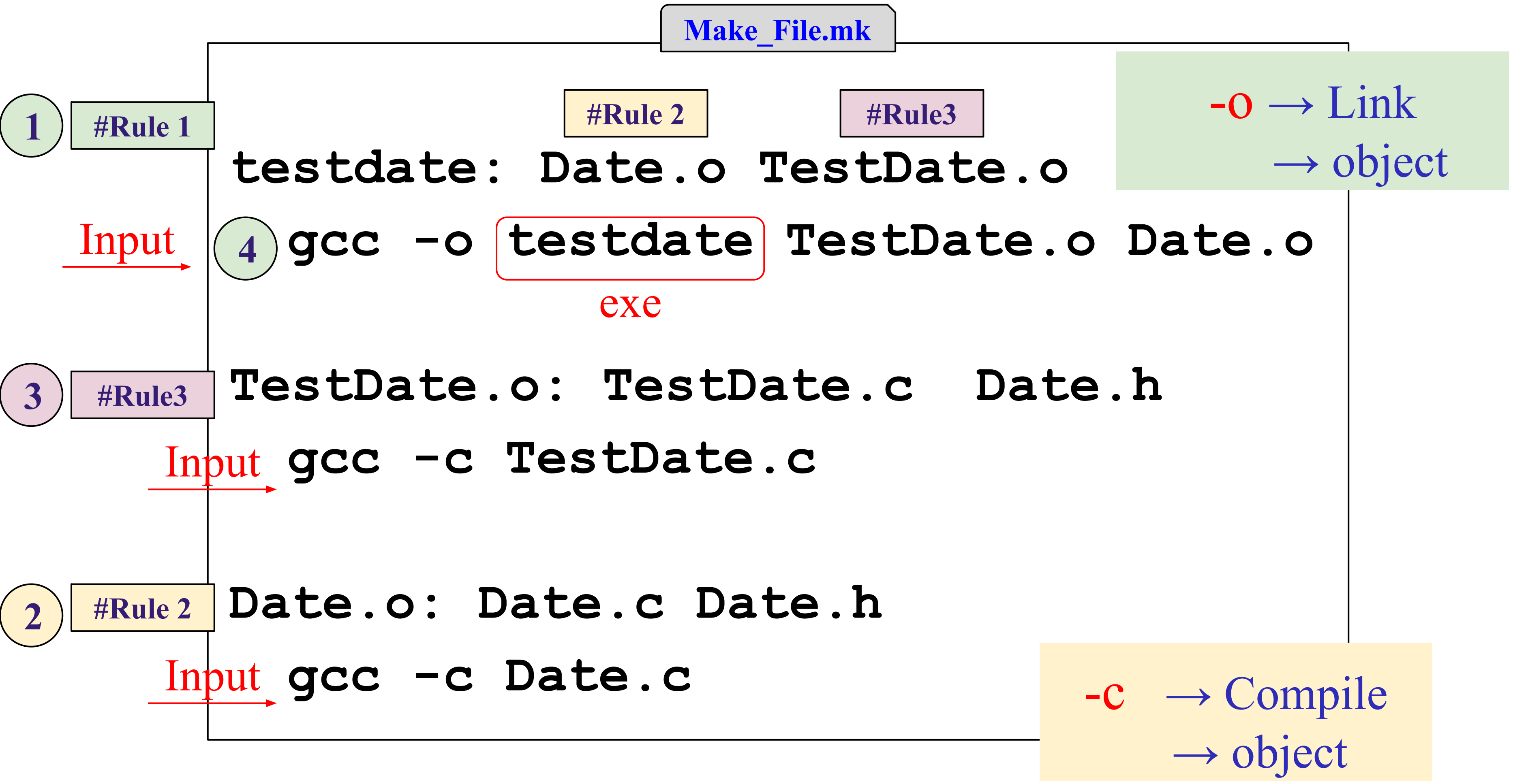


```
TestDate.o: TestDate.c  Date.h
	gcc -c TestDate.c
```

- This rule compiles TestDate.cpp, but does NOT create any executable, just TestDate.o

# Makefile Rules:

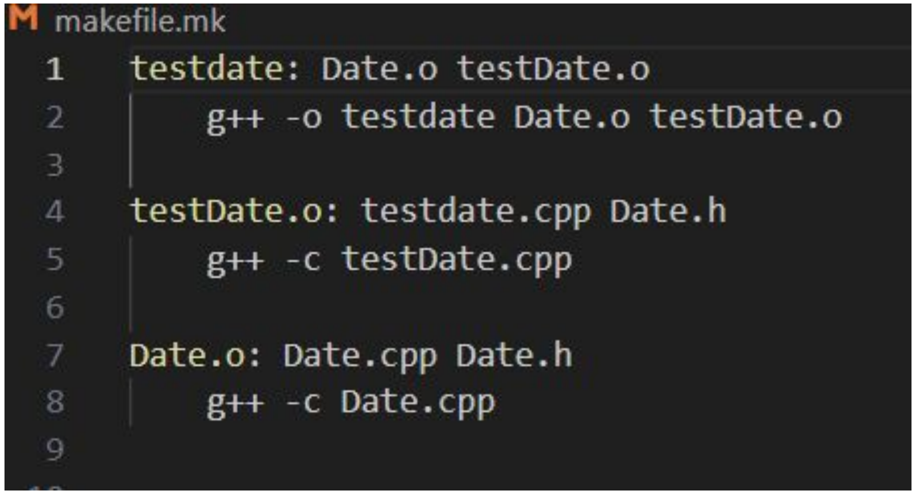
## Targets, Dependencies, and Associated Commands



```
testdate: Date.o TestDate.o
gcc -o testdate TestDate.o Date.o

TestDate.o: TestDate.c  Date.h
gcc -c TestDate.c

Date.o: Date.c Date.h
gcc -c Date.c
```

# Lab 1 *Practical Session 1*

## 1-Lets write our 1st make File



makefile.mk

```
testdate: Date.o testDate.o
	g++ -o testdate Date.o testDate.o

testDate.o: testdate.cpp Date.h
	g++ -c testDate.cpp

Date.o: Date.cpp Date.h
	g++ -c Date.cpp
```

# **g++**
Compile and link c++

# **gcc**
build c & c++
cannot link c++ files

# **-o** to specify executable file name (Compile and link)

# **-c** to compile only (no linking)

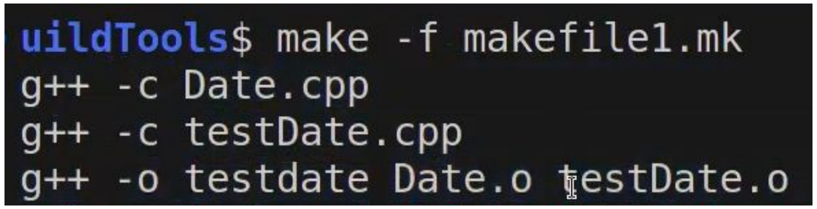
# Continue . . Lab 1 2-Build Project

**Make Commands:**

make #build any file name is makefile.mk

make -f filename.mk # If file name is not makefile” or “Makefile”

1

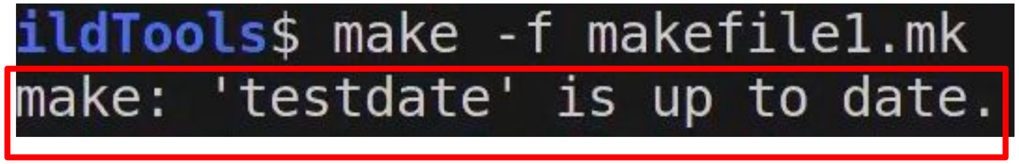


```
uildTools$ make -f makefile1.mk
g++ -c Date.cpp
g++ -c testDate.cpp
g++ -o testdate Date.o testDate.o
```

If you do make after build without adding any change,

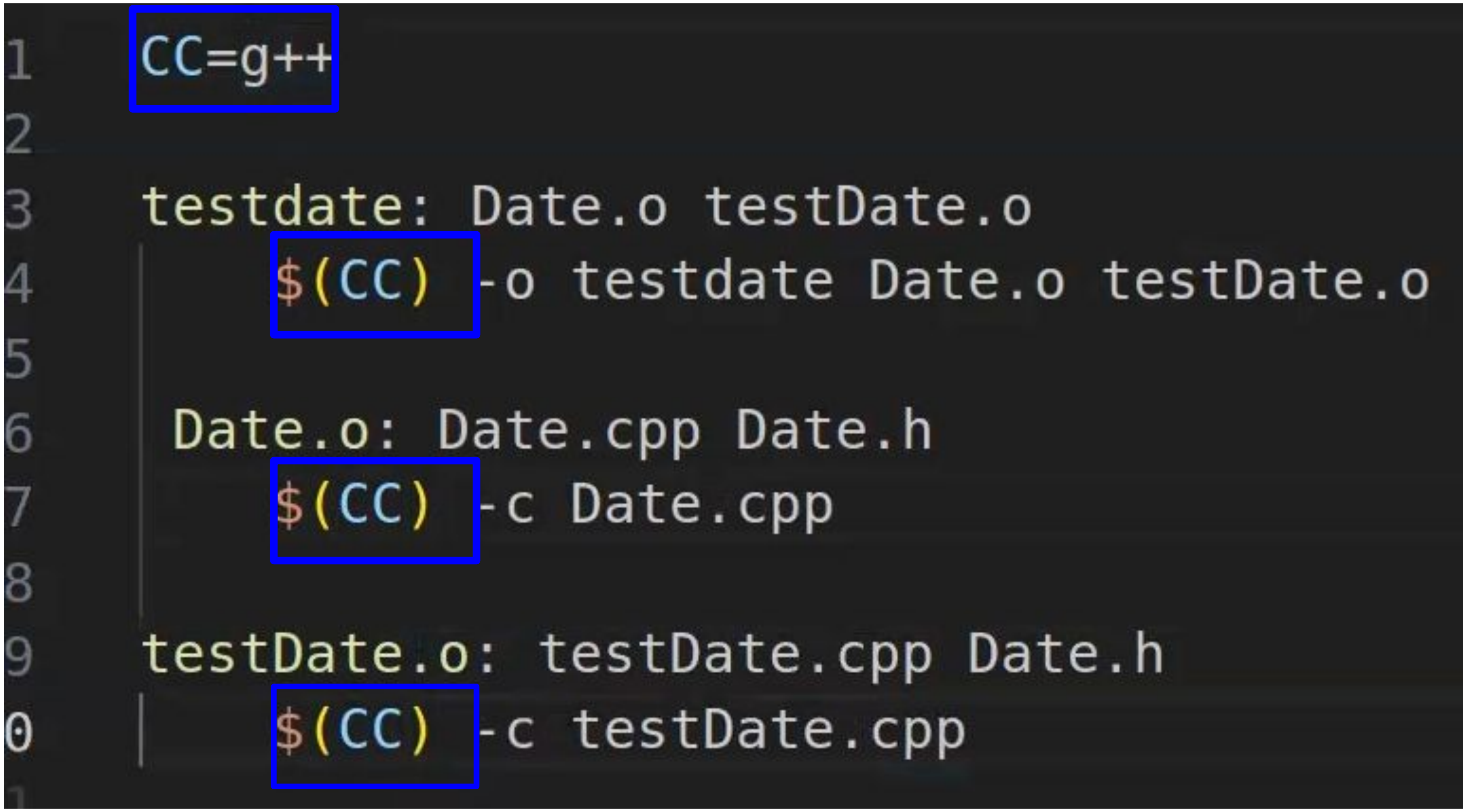
*you may get this message*

2



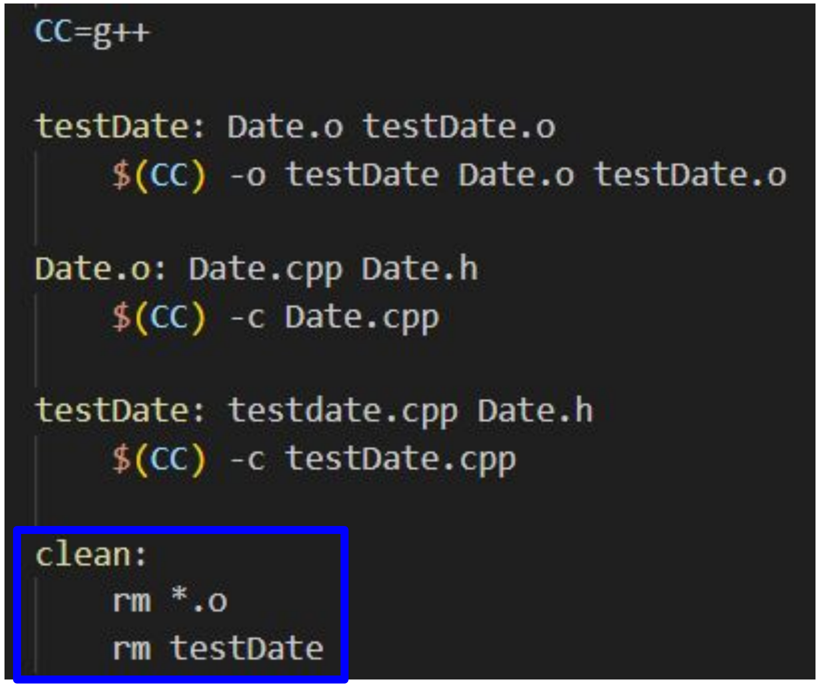
```
ildTools$ make -f makefile1.mk
make: 'testdate' is up to date.
```

## Continue . . Lab 1 - - 3-Add a variable

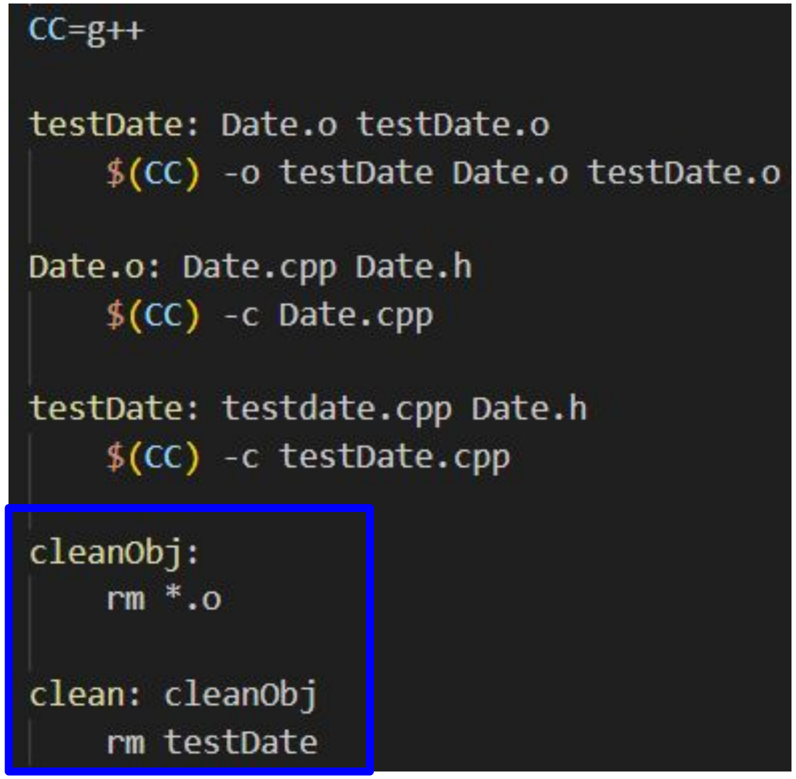


```
CC=g++

testdate: Date.o testDate.o
	$(CC) -o testdate Date.o testDate.o

 Date.o: Date.cpp Date.h
	$(CC) -c Date.cpp

testDate.o: testDate.cpp Date.h
	$(CC) -c testDate.cpp
```

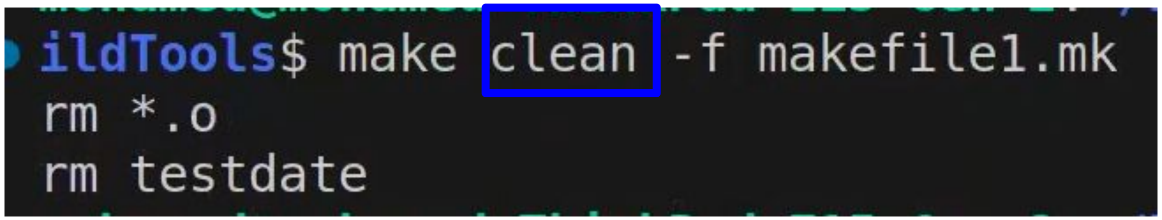
## Continue . . Lab 1 - - 4-Add Clean Rule



```
CC=g++

testDate: Date.o testDate.o
	$(CC) -o testDate Date.o testDate.o

Date.o: Date.cpp Date.h
	$(CC) -c Date.cpp

testDate: testdate.cpp Date.h
	$(CC) -c testDate.cpp

clean:
	rm *.o
	rm testDate
```

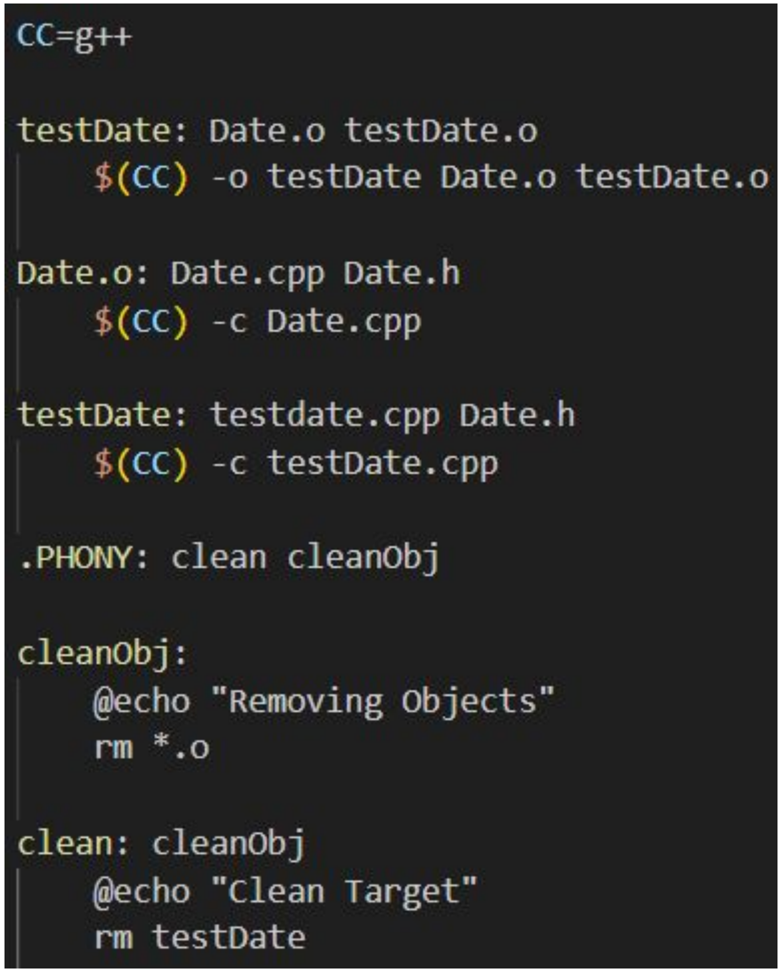


```
CC=g++

testDate: Date.o testDate.o
	$(CC) -o testDate Date.o testDate.o

Date.o: Date.cpp Date.h
	$(CC) -c Date.cpp

testDate: testdate.cpp Date.h
	$(CC) -c testDate.cpp

cleanObj:
	rm *.o

clean: cleanObj
	rm testDate
```

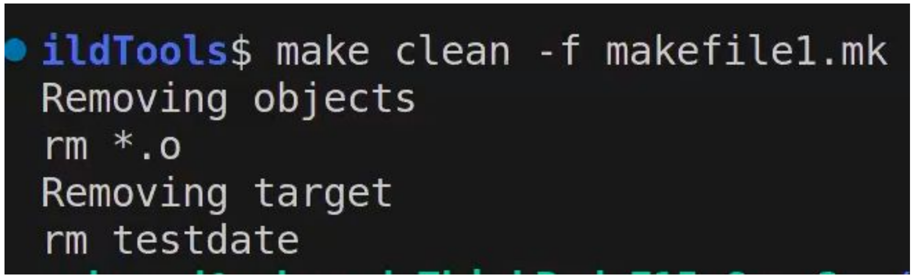


```
ildTools$ make clean -f makefile1.mk
rm *.o
rm testdate
```

# Continue . . Lab 1 - - - 5-Add PHONY



```
CC=g++

testDate: Date.o testDate.o
	$(CC) -o testDate Date.o testDate.o

Date.o: Date.cpp Date.h
	$(CC) -c Date.cpp

testDate: testdate.cpp Date.h
	$(CC) -c testDate.cpp

.PHONY: clean cleanObj

cleanObj:
	@echo "Removing Objects"
	rm *.o

clean: cleanObj
	@echo "Clean Target"
	rm testDate
```



```
ildTools$ make clean -f makefile1.mk
Removing objects
rm *.o
Removing target
rm testdate
```

# Macros

- Macros are used to simplify and automate
  - Often used like constants in programs
- We can compress identical dependencies and use built-in macros to get another (shorter) equivalent makefile :

Left Hand Side      Right Hand Side

```
testdate: TestDate.o Date.o
   g++ -o $@ TestDate.o Date.o

Date.o: Date.cpp Date.h
   gcc -c &<
```

Macro $@ Represents the name of output file.

$@ = `testdate`

Macro $< Represents filename of the first prerequisite.

$< = `Date.cpp`

# Macros

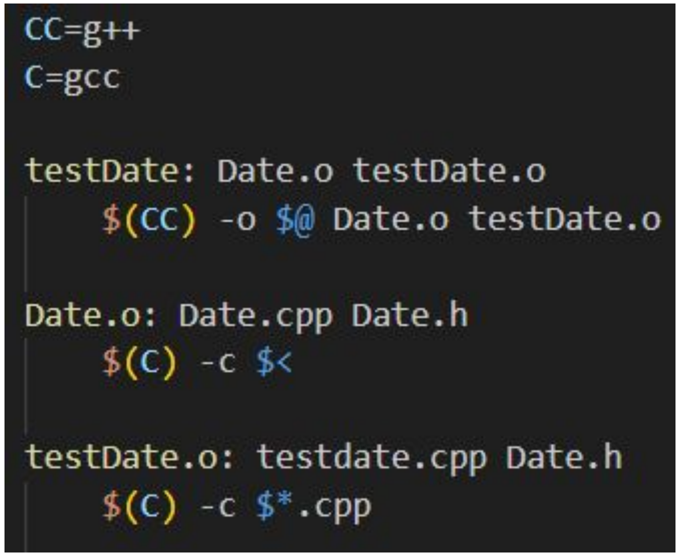
```
TestDate.o: Testdate.cpp Date.h
   gcc -c $*.cpp
```

Macro $* represents target prefixes:

$* = TestDate, if target is TestDate.o

$* =  Date, if target is Date.o

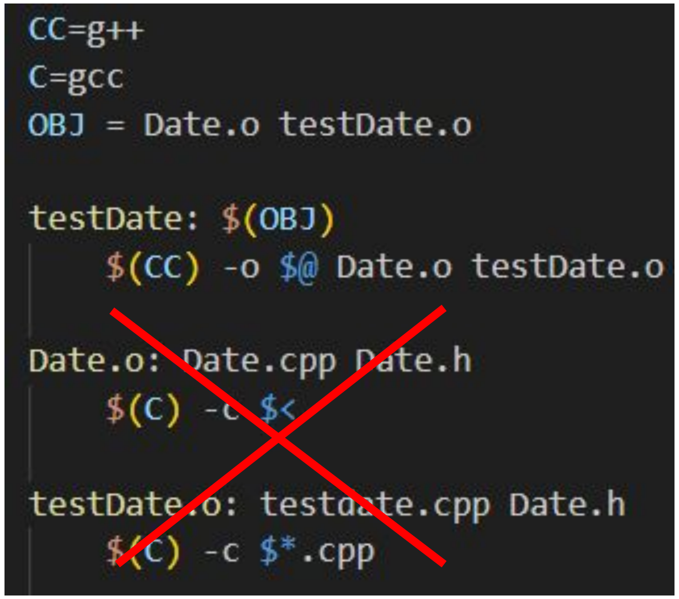
```
target … : prerequisites …
        recipe
        …
        …
```



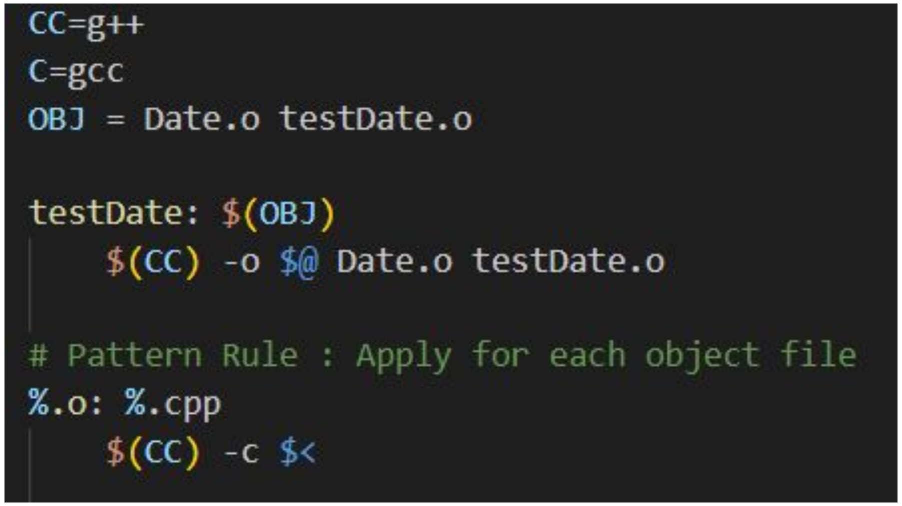
```
CC=g++
C=gcc

testDate: Date.o testDate.o
    $(CC) -o $@ Date.o testDate.o

Date.o: Date.cpp Date.h
    $(C) -c $<

testDate.o: testdate.cpp Date.h
    $(C) -c $*.cpp
```

## Generic Rules

To avoid making build rule for each object



```
CC=g++
C=gcc
OBJ = Date.o testDate.o

testDate: $(OBJ)
	$(CC) -o $@ Date.o testDate.o

Date.o: Date.cpp Date.h
	$(C) -c $<

testDate.o: testdate.cpp Date.h
	$(C) -c $*.cpp
```



```
CC=g++
C=gcc
OBJ = Date.o testDate.o

testDate: $(OBJ)
	$(CC) -o $@ Date.o testDate.o

# Pattern Rule : Apply for each object file
%.o: %.cpp
	$(CC) -c $<
```

acts as for loop , applied for each object dependency

% replaces with file name

# Lab 2 *Practice Session 2*

Make you make file & use the following:

1. varabiles
2. macros
3. pattern rule

## Conditional statements - example

```
sum: main.o sum.o
   gcc -o sum main.o sum.o

main.o: main.c sum.h
   gcc -c main.c

#deciding which file to compile to create sum.o
ifeq ($(USE_SUM), 1)

sum.o: sum1.c sum.h
   gcc -c sum1.c -o $@

else
sum.o: sum2.c sum.h
   gcc -c sum2.c -o $@

endif
```

## `make` options

**`-f`** *filename* – when the makefile name is not standard

**`-t`** – (touch) mark the targets as up to date

**`-q`** – (question) are the targets up to date, exits with 0 if true

**`-n`** – print the commands to execute but do not execute them

**/* `-t`, `-q`, and `-n`, cannot be used together * /**

**`-s`** – silent mode : Don't print commands

**`-k`** – keep going – compile all the prerequisites even if not able to link them !!

# VPATH

- `VPATH` variable – defines directories to be searched if a file is not found in the current directory.

  `VPATH = `*`dir`*` : `*`dir`*` …`

  /* `VPATH = src:../headers */`

- `vpath` directive (lower case!) – more selective directory search:

  `vpath `*`pattern directory`*

  `/ vpath %.h headers /`

- `GPATH:`

  `GPATH` – if you want targets to be stored in the same directory as their dependencies.

## make operation

- Project's dependency tree is constructed
- Target of first rule is to be created
- Go down the dependency tree to see if there is a target that should be recreated:
  - target file is older than one of its dependencies
  - recreate the target file according to the action specified, on our way up the tree. Consequently, more files may need to be recreated
- If something is changed, linking is usually necessary
  - Top target in dependency tree

## make operation - continued

- make operation ensures minimum compilation, when the project structure is written properly
- Do not write something like:

```
testdate: TestDate.c Date.c Date.h
 gcc  -o testdate TestDate.c Date.c
```

  which requires compilation of all files when something is changed
  - Example: If Date.cpp changes, why compile TestDate.cpp?

## Passing parameters to makefile

- Parameters can be passed to a makefile by specifying them along with their values in the command line.
- For example:

```
make PAR1=1 PAR2=soft1
```

will call the **`makefile`** with 2 parameters: **`PAR1`** is assigned the value “**`1`**” and **`PAR2`** is assigned the value “**`soft1`**”. The same names should be used within the makefile to access these variables (using the usual “$(VAR_NAME)” syntax)

# Passing parameters - continued

- Note that assigning a value to a variable within the makefile overrides any value passed from the command line.
- For example:

  command line : `make PAR=1`

  in the makefile:

  ```
  PAR = 2
  ```

- **`PAR`** value within the **`makefile`** will be 2, overriding the value sent from the command line

# Example

- For example, if you have the following source files in some project of yours:
  - ccountln.h
  - ccountln.c
  - fileops.h
  - fileops.c
  - process.h
  - process.c
  - parser.h
  - parser.c
- You could compile every C file and then link the object files generated, or use a single command for the entire thing.
  - This becomes unfriendly when the number of files increases; hence, use Makefiles!
- NOTE: you don't NEED to compile **`.h`** files explicitly.

## Example (2)

- One by one:
  - `gcc -g -Wall -ansi -pedantic -c ccountln.c`
  - `gcc -g -Wall -ansi -pedantic -c parser.c`
  - `gcc -g -Wall -ansi -pedantic -c fileops.c`
  - `gcc -g -Wall -ansi -pedantic -c process.c`

- This will give you four object files that you need to link and produce an executable:
  - `gcc ccountln.o parser.o fileops.o process.o -o ccountln`

## Example (3)

- You can do this as well:
  - gcc -g -Wall -ansi -pedantic ccountln.c parser.c fileops.c process.c -o ccountln

- Instead of typing this all on a command line, again: use a Makefile.

## Example (4)

```
# Simple Makefile with use of gcc could look like this
CC=gcc
CFLAGS=-g -Wall -ansi -pedantic
OBJ:=ccountln.o parser.o process.o fileops.o
EXE=ccountln

all: $(EXE)

$(EXE): $(OBJ)
   $(CC) $(OBJ) -o $(EXE)

ccountln.o: ccountln.h ccountln.c
   $(CC) $(CFLAGS) -c ccountln.c
...
```

# makefiles content

Makefiles content

-  rules : implicit, explicit
-  variables (macros)
-  directives (conditionals)
-  # sign –  comments everything till the end of the line
-  \ sign - to separate one command line on two rows

## Sample makefile

- Makefiles main element is called a *rule*:

```
target : dependencies
TAB  commands         #shell commands
```

**Example:**

```
my_prog : eval.o main.o
  g++ -o my_prog eval.o main.o

eval.o : eval.c eval.h
  g++ -c eval.c
main.o : main.c eval.h
  g++ -c main.c
```

```
_______________________________
# -o to specify executable file name
# -c to compile only (no linking)
```

# Variables

The old way (no variables)

```
my_prog : eval.o main.o
	g++ -o my_prog eval.o main.o
eval.o : eval.c eval.h
	g++ -c -g eval.c
main.o : main.c eval.h
	g++ -c -g main.c
```

A new way (using variables)

```
C = g++
OBJS = eval.o main.o
HDRS = eval.h

my_prog : eval.o main.o
	$(C) -o my_prog $(OBJS)
eval.o : eval.c
	$(C) -c -g eval.c
main.o : main.c
	$(C) -c -g main.c
$(OBJS) : $(HDRS)
```

<u>Defining variables on the command line:</u>

Take precedence over variables defined in the makefile.

```
make C=cc
```

# Implicit rules

- Implicit rules are standard ways for making one type of file from another type.
- There are numerous rules for making an ***.o*** file – from a ***.c*** file, a ***.p*** file, etc. `make` applies the first rule it meets.
- If you have not defined a rule for a given object file, `make` will apply an implicit rule for it.

**<u>Example:</u>**

Our makefile

```
my_prog : eval.o main.o
   $(C) -o my_prog $(OBJS)
$(OBJS) : $(HEADERS)
```

The way `make` understands it

```
my_prog : eval.o main.o
    $(C) -o my_prog $(OBJS)
$(OBJS) : $(HEADERS)
eval.o : eval.c
    $(C) -c eval.c
main.o : main.c
    $(C) -c main.c
```

## Defining implicit rules

```
%.o : %.c
   $(C) -c -g $<

C = g++
OBJS = eval.o main.o
HDRS = eval.h

my_prog : eval.o main.o
   $(C) -o my_prog $(OBJS)
$(OBJS) : $(HDRS)
```

<u>Avoiding implicit rules - empty commands</u>

**`target: ;`**      #Implicit rules will not apply for this target.

# Automatic variables

Automatic variables are used to refer to specific part of rule components.

```
target : dependencies
TAB    commands     #shell commands
```

```
eval.o : eval.c eval.h
 g++ -c eval.c
```

`$@` - The name of the target of the rule (`eval.o`).

`$<` - The name of the first dependency (`eval.c`).

`$^` - The names of all the dependencies (`eval.c eval.h`).

`$?` - The names of all dependencies that are newer than the target

# Phony targets

<u>Phony targets:</u>

Targets that have no dependencies.

Used only as names for commands that you want to execute.

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

or

```
.PHONY : clean
clean:
   rm $(OBJS)
```

---

To invoke it: `make clean`

<u>Typical phony targets:</u>

`all` – make all the top level targets

```
 .PHONY : all
 all: my_prog1 my_prog2
```

`clean` – delete all files that are normally created by `make`

`print` – print listing of the source files that have changed

# Conditionals (directives)

Possible conditionals are:

`if    ifeq   ifneq   ifdef   ifndef`

All of them should be closed with `endif`.

Complex conditionals may use `elif` and `else`.

**Example:**

```
libs_for_gcc = -lgnu
normal_libs =
ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)          #no tabs at the beginning
else
  libs=$(normal_libs)           #no tabs at the beginning
endif
```