Make

HGU

How to build a program consisting of Multiple source files?

We want to execute the following 5 example files.

main.c

```
#include "foo.h"
#include "bar.h"

int main() {
    int x = foo(bar(3));
    printf("main: %d\n",x);
    return 0;
}
```

foo.h

```
#include <stdio.h>

int foo(int x);
```

bar.h

```
#include <stdio.h>

int bar(int x);
```

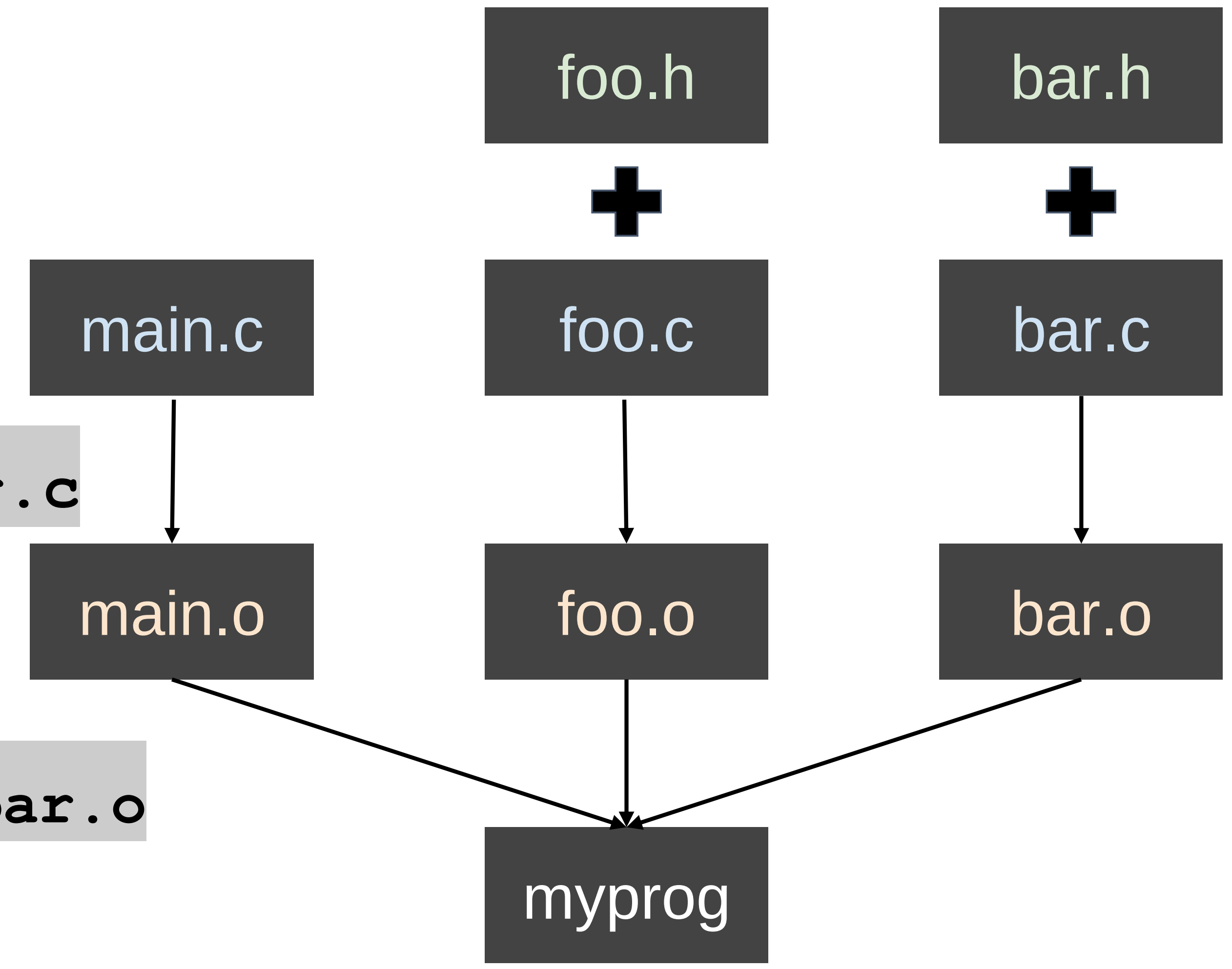
foo.c

```
#include "foo.h"

int foo(int x) {
    printf("foo: %d*2", x);
    return x * 2;
}
```

bar.c

```
#include "bar.h"

int bar(int x) {
    printf("bar: %d+2", x);
    return x + 2;
}
```

How to build a program consisting of Multiple source files?



Compile : `gcc –c main.c foo.c bar.c`

Linking : `gcc myprog main.o foo.o bar.o`

How to build a program consisting of Multiple source files?

- Long GCC Command with many options

```
$ gcc -Wall -g -o myprog main.c foo.c bar.c
```

- Useful Options
  - -Wall : Enables warning messages
  - -g : Include debugging information
  - -o : Specifies the name of the output file

How to build a program consisting of Multiple source files?

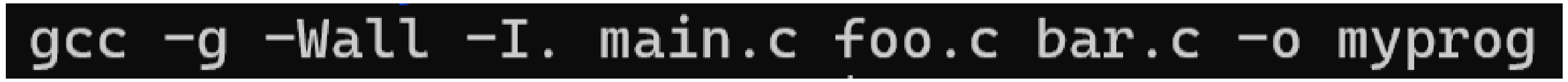
- Create **Makefile**

Makefile

```
all : main.c foo.c bar.c
	gcc -g -Wall main.c foo.c bar.c -o myprog

clean :
	rm myprog
```

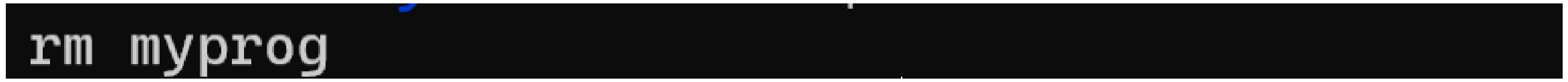
```
$ make
```

→ searches *Makefile* or *makefile* in current directory unless –*f* <*make_filename*> specified



```
gcc -g -Wall -I. main.c foo.c bar.c -o myprog
```

```
$ make clean
```

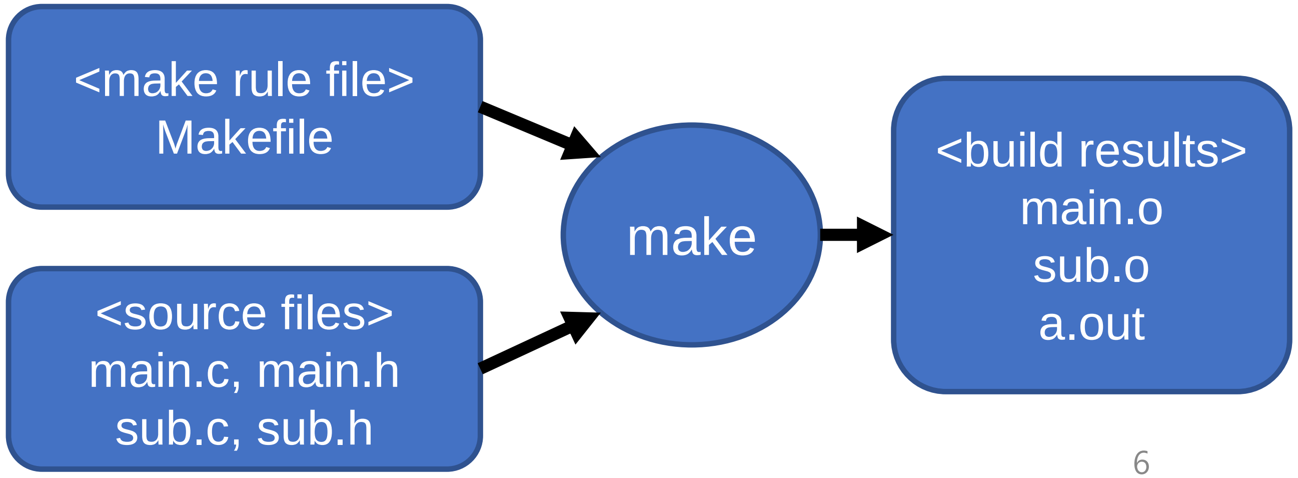


```
rm myprog
```

Make

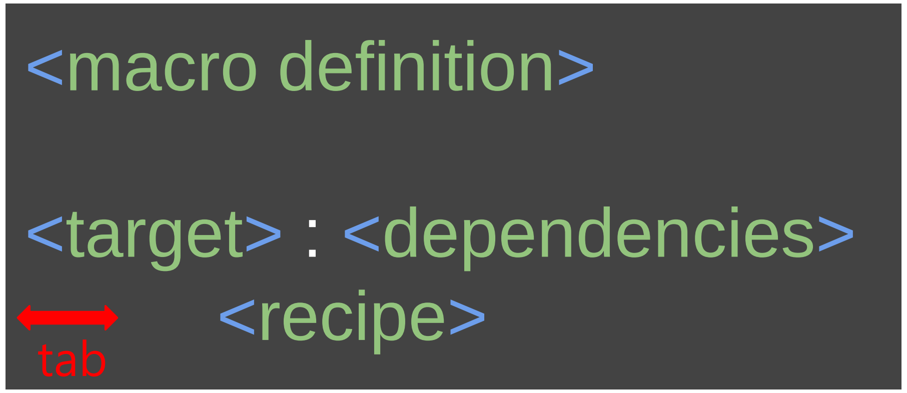
- Make is a **build automation tool** used in software development to control the build process.
- Configuration: The build process is defined in a file '**Makefile**'.
- Configuration Rule:
  - Every rule consists of <target file>, <dependency file>, and <command>
- Running make
  - **$ make**

  or
  - **$ make –f <make_rule_file>**



Makefile

- What is Makefile?



```
<macro definition>

<target> : <dependencies>
	<recipe>
```

- Build Automation
  : execute a set of rules

- Incremental Build
  : rebuild the updated files only

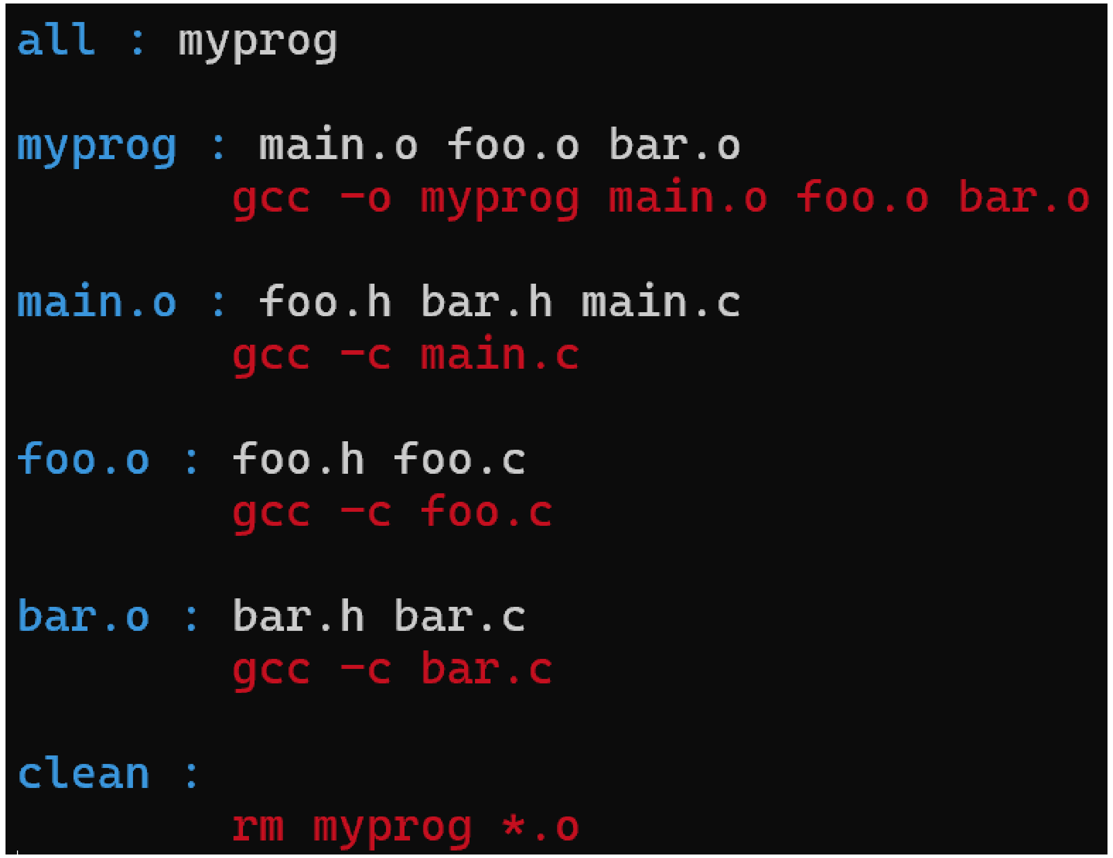
Run by

```
$ make [<target>]
```

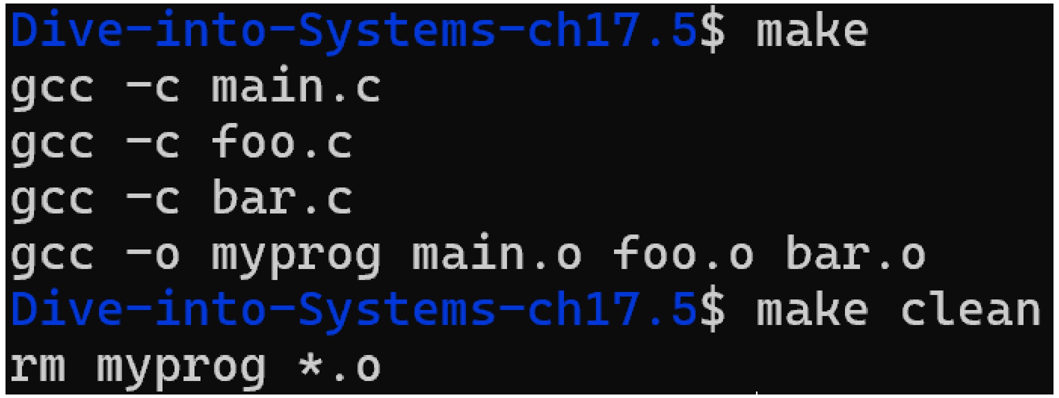
If **`<target>`** is empty in command, target the first **`<target>`**

Makefile Example #1

- Build Automation : list the rule for each dependencies
  - Compare the modified **date** of dependency files and target file



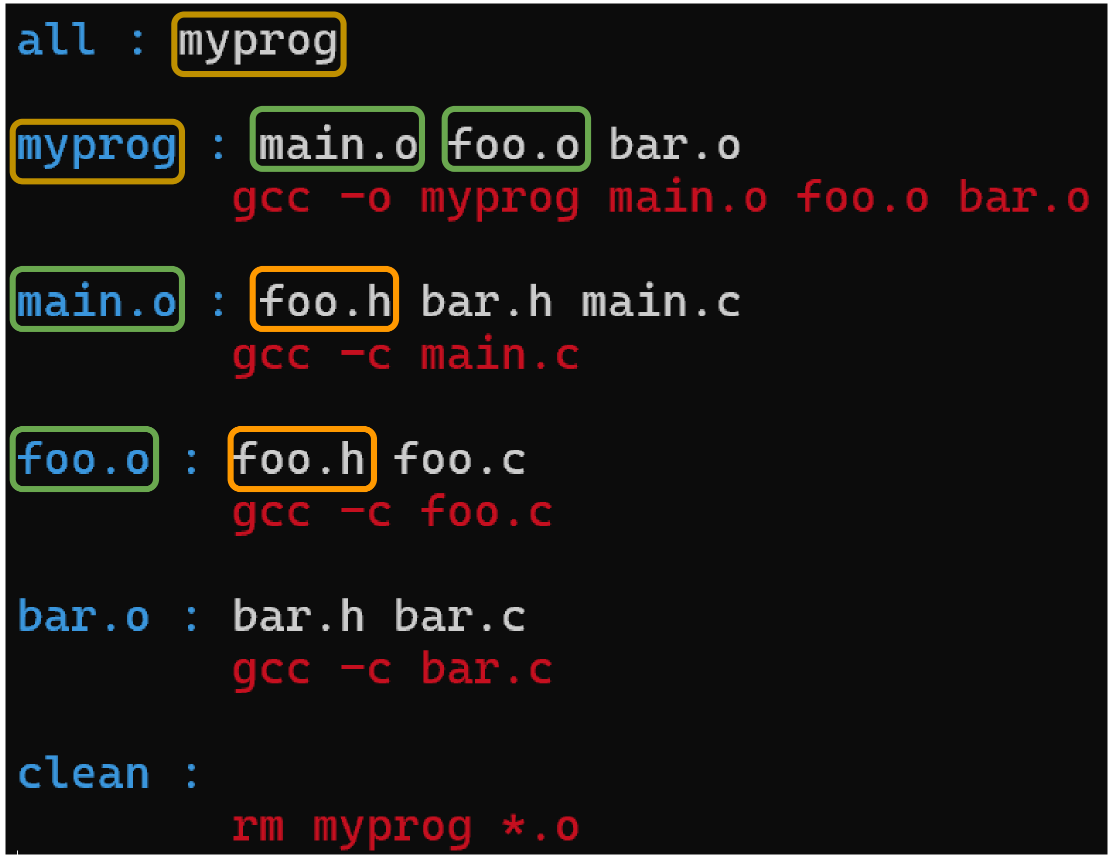
```
all : myprog

myprog : main.o foo.o bar.o
        gcc -o myprog main.o foo.o bar.o

main.o : foo.h bar.h main.c
        gcc -c main.c

foo.o : foo.h foo.c
        gcc -c foo.c

bar.o : bar.h bar.c
        gcc -c bar.c

clean :
        rm myprog *.o
```



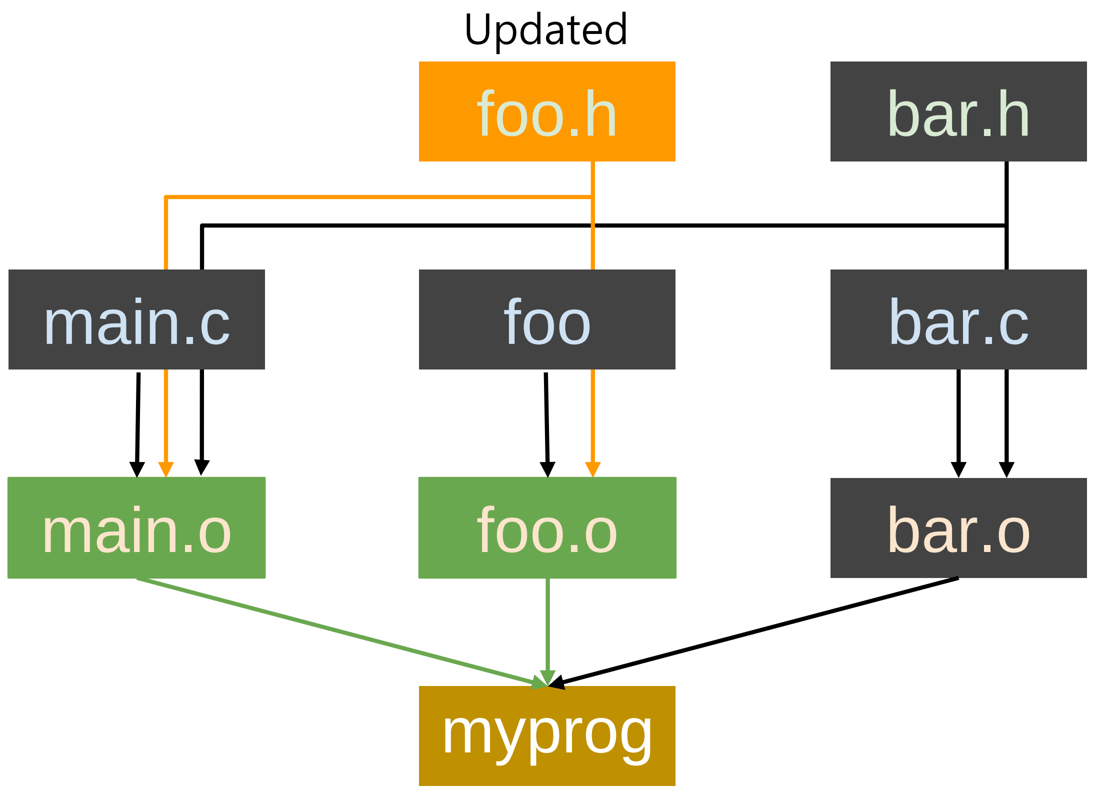
```
Dive-into-Systems-ch17.5$ make
gcc -c main.c
gcc -c foo.c
gcc -c bar.c
gcc -o myprog main.o foo.o bar.o
Dive-into-Systems-ch17.5$ make clean
rm myprog *.o
```

Makefile Example #1

- Incremental Build : rebuilt updated file only

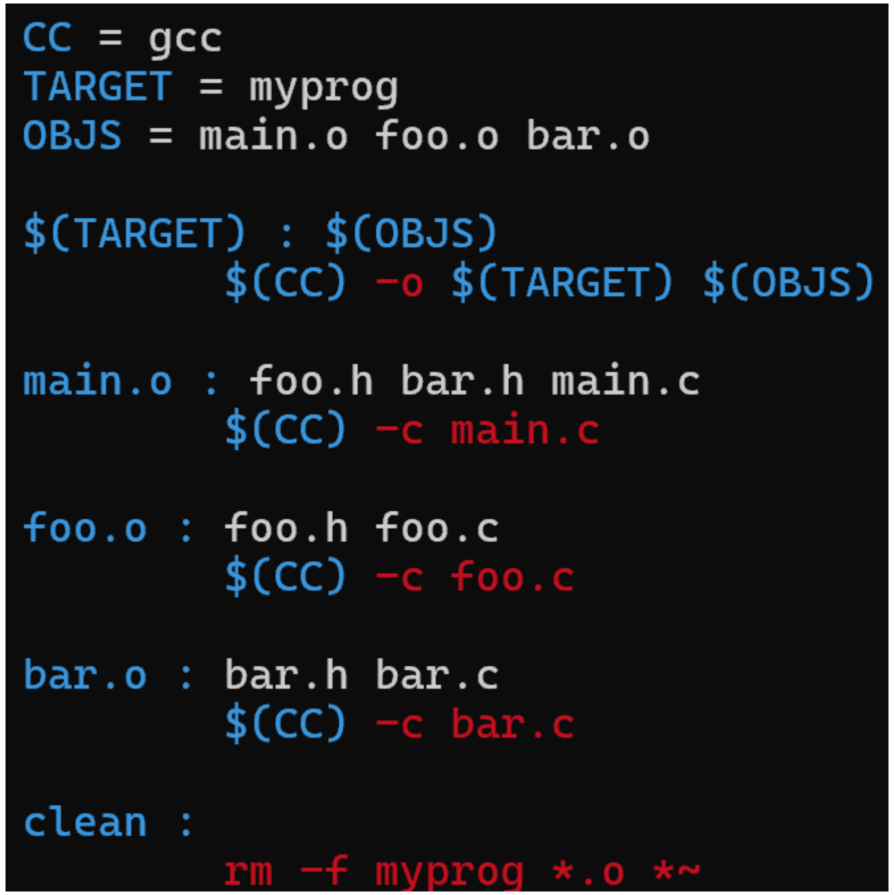


```
all : myprog

myprog : main.o foo.o bar.o
        gcc -o myprog main.o foo.o bar.o

main.o : foo.h bar.h main.c
        gcc -c main.c

foo.o : foo.h foo.c
        gcc -c foo.c

bar.o : bar.h bar.c
        gcc -c bar.c

clean :
        rm myprog *.o
```



Makefile Example #2

- A Simple Generic Makefile: : Use the variables with Macros definition
  - Macro Definition Syntax: Macro_name = Macro contents
  - Variable naming syntax : $(Macro_name)



```
CC = gcc
TARGET = myprog
OBJS = main.o foo.o bar.o

$(TARGET) : $(OBJS)
        $(CC) -o $(TARGET) $(OBJS)

main.o : foo.h bar.h main.c
        $(CC) -c main.c

foo.o : foo.h foo.c
        $(CC) -c foo.c

bar.o : bar.h bar.c
        $(CC) -c bar.c

clean :
        rm -f myprog *.o *~
```

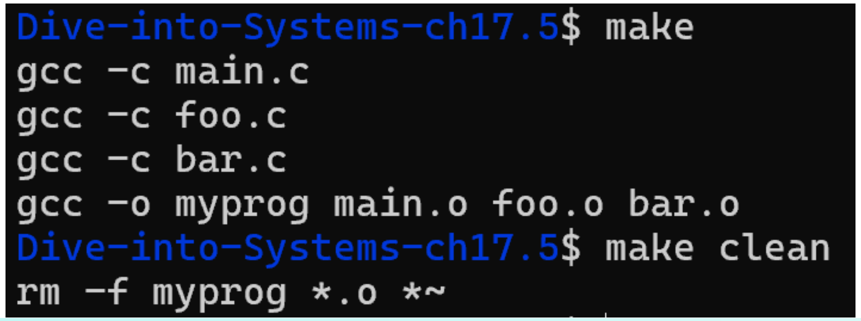
● **Commonly used Variables**

CC = <Compiler>
TARGET = <Build Target>
OBJS = <Object Files>

● Commonly used Flag Variables

CFLAGS : compile option	(`-g -Wall`)
CPPFLAGS : preprocessor flags	(`-MD`)
LDFLAGS : linking directory	(`-L/local/lib`)
LDLIBS : linking library option	(`-lm -lexlib`)

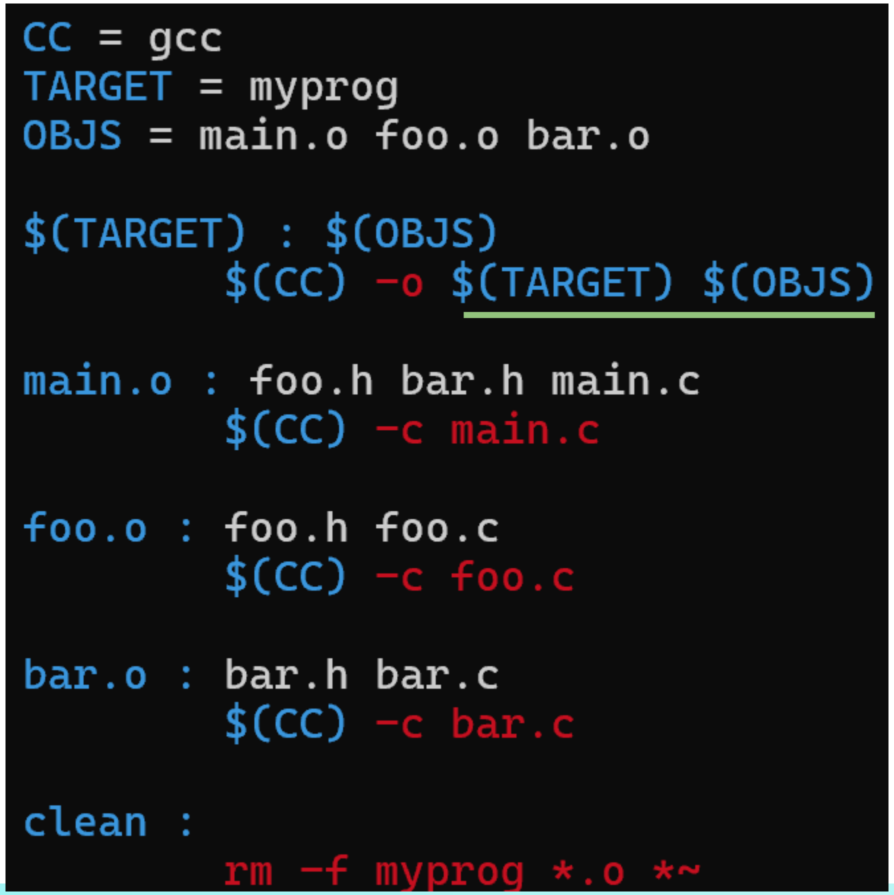
Run



```
Dive-into-Systems-ch17.5$ make
gcc -c main.c
gcc -c foo.c
gcc -c bar.c
gcc -o myprog main.o foo.o bar.o
Dive-into-Systems-ch17.5$ make clean
rm -f myprog *.o *~
```

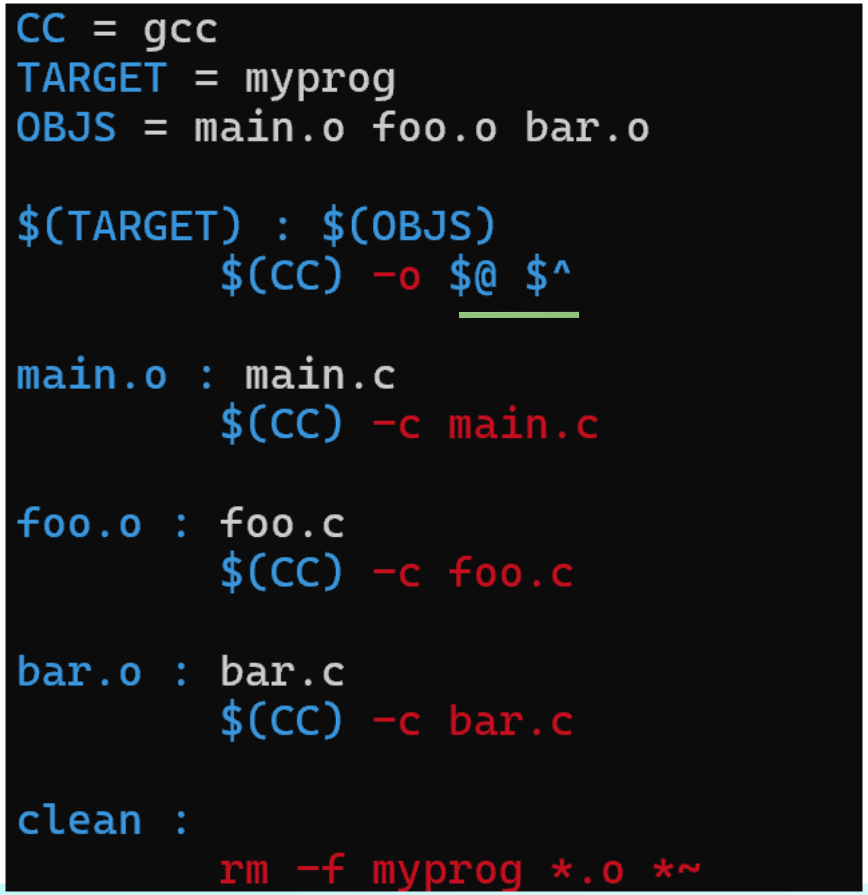
Makefile Example #2

- A Simple Generic Makefile
  : simplify the variables using **Automatic Variables ($^)**



```
CC = gcc
TARGET = myprog
OBJS = main.o foo.o bar.o

$(TARGET) : $(OBJS)
        $(CC) -o $(TARGET) $(OBJS)

main.o : foo.h bar.h main.c
        $(CC) -c main.c

foo.o : foo.h foo.c
        $(CC) -c foo.c

bar.o : bar.h bar.c
        $(CC) -c bar.c

clean :
        rm -f myprog *.o *~
```

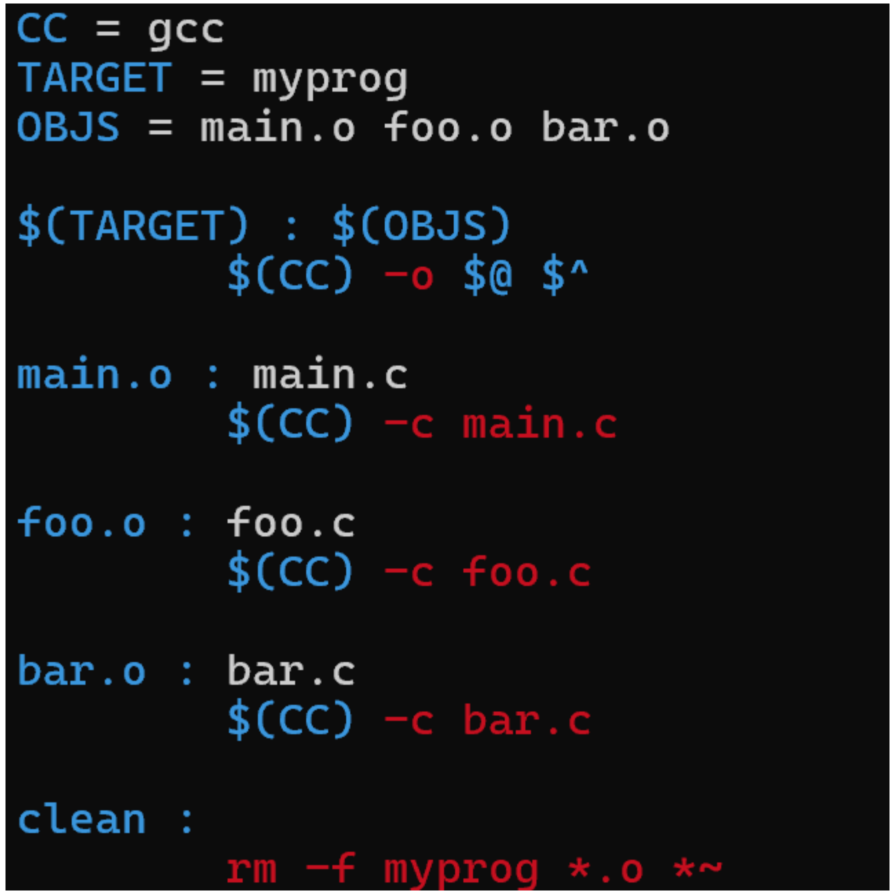
Simplify



```
CC = gcc
TARGET = myprog
OBJS = main.o foo.o bar.o

$(TARGET) : $(OBJS)
        $(CC) -o $@ $^

main.o : main.c
        $(CC) -c main.c

foo.o : foo.c
        $(CC) -c foo.c

bar.o : bar.c
        $(CC) -c bar.c

clean :
        rm -f myprog *.o *~
```

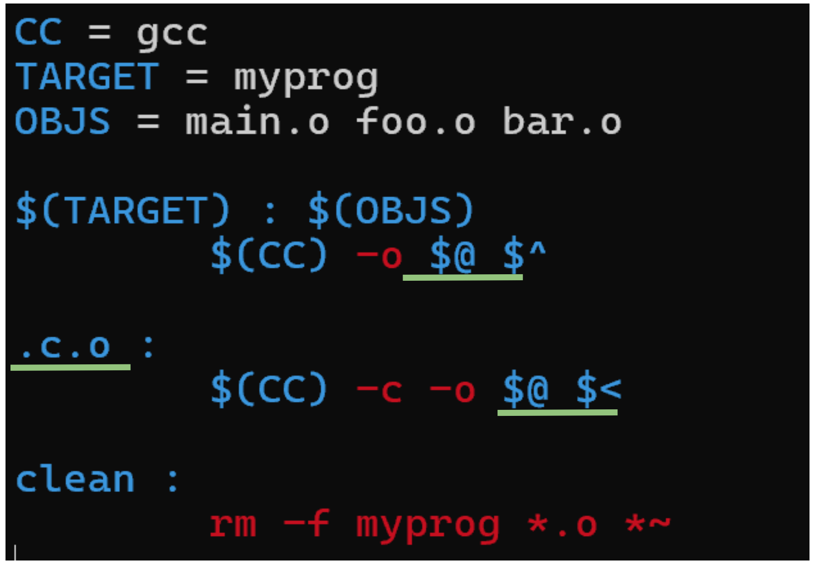
Makefile Example #2

- A Simpler Generic Makefile using the Makefile **Automatic variables**



```
CC = gcc
TARGET = myprog
OBJS = main.o foo.o bar.o

$(TARGET) : $(OBJS)
        $(CC) -o $@ $^

main.o : main.c
        $(CC) -c main.c

foo.o : foo.c
        $(CC) -c foo.c

bar.o : bar.c
        $(CC) -c bar.c

clean :
        rm -f myprog *.o *~
```

Simplify



```
CC = gcc
TARGET = myprog
OBJS = main.o foo.o bar.o

$(TARGET) : $(OBJS)
        $(CC) -o $@ $^

.c.o :
        $(CC) -c -o $@ $<

clean :
        rm -f myprog *.o *~
```

● **Automatic Variables**

$@ : Current target name (**main.o**)

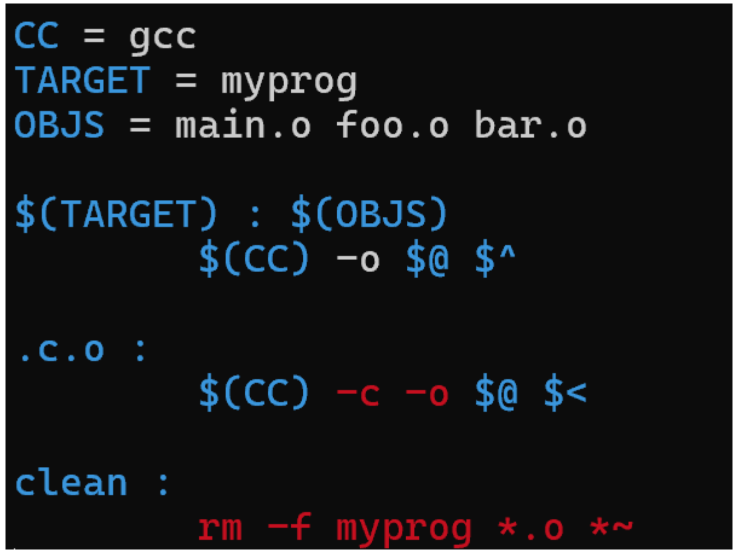
$^ : All dependencies (**main.c foo.h bar.h**)

$< : The first dependency file (**main.c**)

$* : Current target name without extension (**main**)

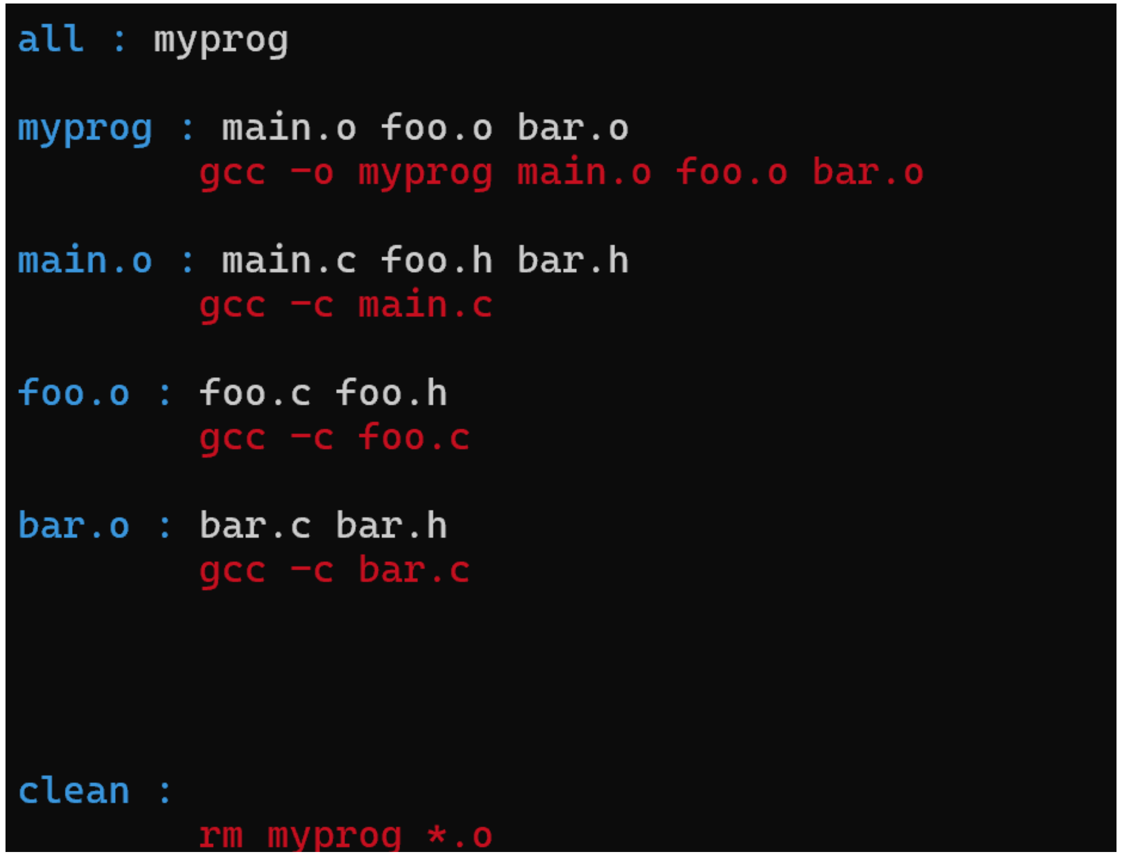
Compare Two Makefiles

Generic version may look like complex and hard to understand



```
CC = gcc
TARGET = myprog
OBJS = main.o foo.o bar.o

$(TARGET) : $(OBJS)
        $(CC) -o $@ $^

.c.o :
        $(CC) -c -o $@ $<

clean :
        rm -f myprog *.o *~
```

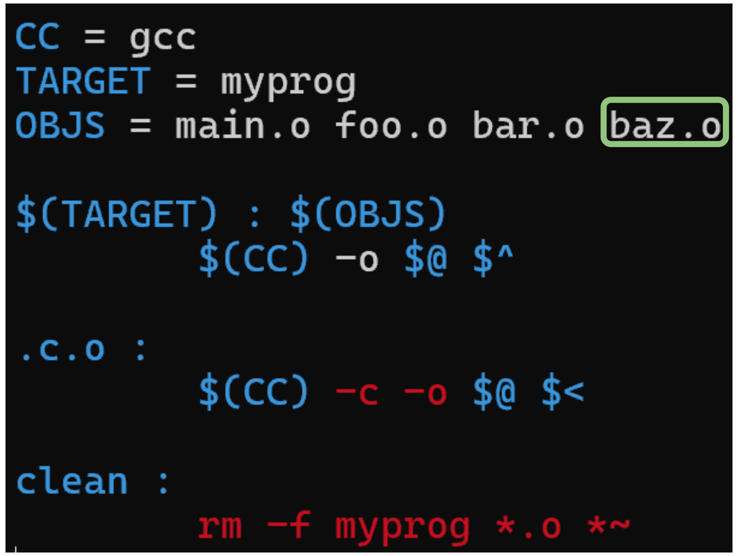
VS



```
all : myprog

myprog : main.o foo.o bar.o
        gcc -o myprog main.o foo.o bar.o

main.o : main.c foo.h bar.h
        gcc -c main.c

foo.o : foo.c foo.h
        gcc -c foo.c

bar.o : bar.c bar.h
        gcc -c bar.c




clean :
        rm myprog *.o
```

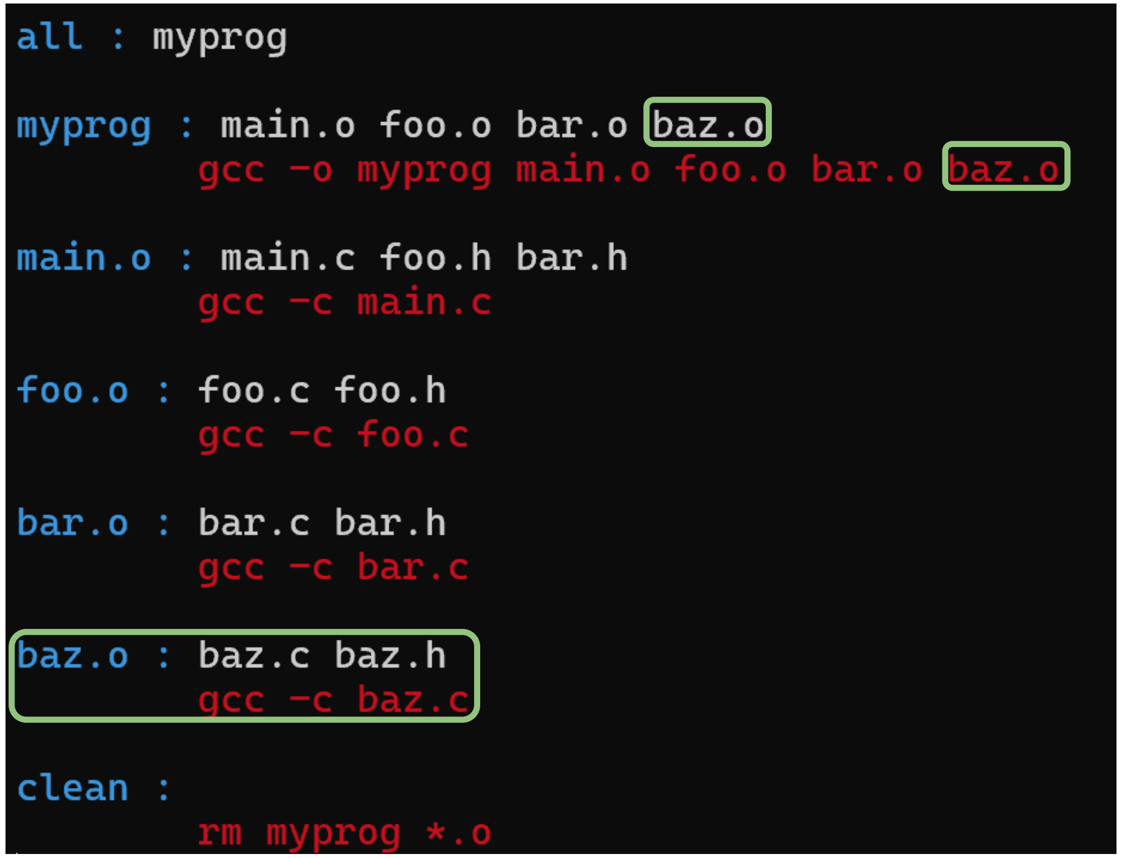
Compare Two Makefiles

Generic version may look like complex and hard to understand, but it has versatility and expandability (baz.c baz.h)



```
CC = gcc
TARGET = myprog
OBJS = main.o foo.o bar.o baz.o

$(TARGET) : $(OBJS)
        $(CC) -o $@ $^

.c.o :
        $(CC) -c -o $@ $<

clean :
        rm -f myprog *.o *~
```

vs



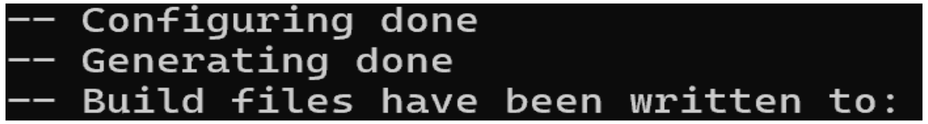
```
all : myprog

myprog : main.o foo.o bar.o baz.o
        gcc -o myprog main.o foo.o bar.o baz.o

main.o : main.c foo.h bar.h
        gcc -c main.c

foo.o : foo.c foo.h
        gcc -c foo.c

bar.o : bar.c bar.h
        gcc -c bar.c

baz.o : baz.c baz.h
        gcc -c baz.c

clean :
        rm myprog *.o
```

Makefile Generators - CMake

CMakeLists.txt

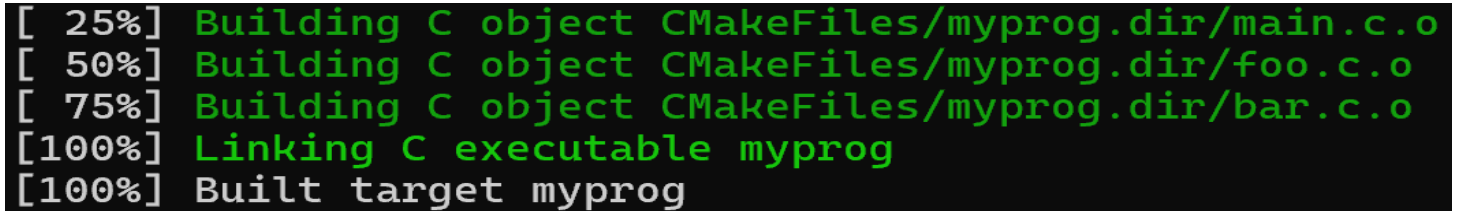
```
cmake_minimum_required(VERSION 3.10)

project(MyProg)

set(CMAKE_C_COMPILER gcc)

set(TARGET myprog)

set(SOURCES main.c foo.c bar.c)

add_executable(${TARGET} ${SOURCES})
```

```
$ cmake .
```



```
-- Configuring done
-- Generating done
-- Build files have been written to:
```

```
$ make
```



```
[ 25%] Building C object CMakeFiles/myprog.dir/main.c.o
[ 50%] Building C object CMakeFiles/myprog.dir/foo.c.o
[ 75%] Building C object CMakeFiles/myprog.dir/bar.c.o
[100%] Linking C executable myprog
[100%] Built target myprog
```

Practice for Make

tutorial2-arith

- There are **5 C source files** in src directory and **4 header files** in include directory.
- Manage the dependencies to rebuild the updated file only.
- Object files should be stored in **build** directory.

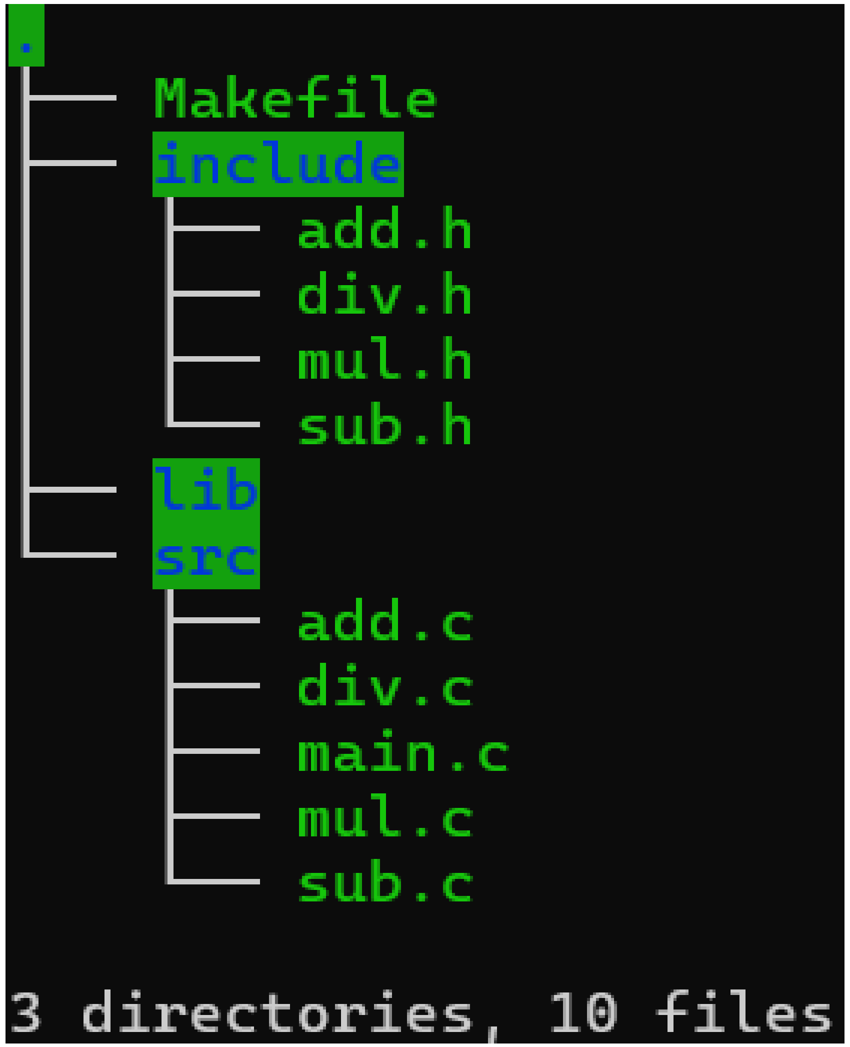
  (If build directory doesn't exist, create it.)

- You should make static and shared library.

  - Static library **libarith_static.a** includess **add** and **sub** function

  - Shared library **libarith_shared.so** includes **mul** and **div** functions.

✓ Build the program $ **`make`**
✓ Test the execution $ **`make test`**
✓ Remove the created files $ **`make clean`**
✓ Rebuild the libraries $ **`make (static|shared)_lib`**



```
.
├── Makefile
├── include
│   ├── add.h
│   ├── div.h
│   ├── mul.h
│   └── sub.h
├── lib
└── src
    ├── add.c
    ├── div.c
    ├── main.c
    ├── mul.c
    └── sub.c

3 directories, 10 files
```

Practice Answer(1)

Makefile

```
CC = gcc
AR = ar
CFLAGS = -Wall -O2 -fPIC -Iinclude
TARGET = myprog
STATIC_LIB = lib/libarith_static.a
SHARED_LIB = lib/libarith_shared.so
STATIC_OBJS = add.o sub.o
SHARED_OBJS = mul.o div.o
MAIN_OBJS = main.o

all: build $(TARGET)

build:
	mkdir -p build

$(TARGET): $(addprefix build/,$(MAIN_OBJS)) $(STATIC_LIB) $(SHARED_LIB)
	$(CC) $(CFLAGS) -o $@ $(addprefix build/,$(MAIN_OBJS)) $(STATIC_LIB) -Llib -
larith_shared -Wl,-rpath,lib
```

Continue..

Practice Answer(2)

Makefile

```
$(STATIC_LIB): $(addprefix build/,$(STATIC_OBJS))
	$(AR) rcs $@ $(addprefix build/,$(STATIC_OBJS))

$(SHARED_LIB): $(addprefix build/,$(SHARED_OBJS))
	$(CC) -shared -o $@ $(addprefix build/,$(SHARED_OBJS))

build/%.o: src/%.c | build
	$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm -f $(TARGET) $(STATIC_LIB) $(SHARED_LIB) $(addprefix
build/,$(STATIC_OBJS)) $(addprefix build/,$(SHARED_OBJS))
	rm -rf build

test: $(TARGET)
	./$(TARGET)

static_lib: $(STATIC_LIB)
shared_lib: $(SHARED_LIB)
```

addprefix

- **addprefix** is GNU Make function that is used to add a specified prefix to each word in a list of words
  - particularly useful when adding directory paths to filenames
- Syntax: $(**addprefix** *prefix, names)*
  - *prefix: the string or path you want to add*
  - *names: list of words (filenames or variables)*
- Example:

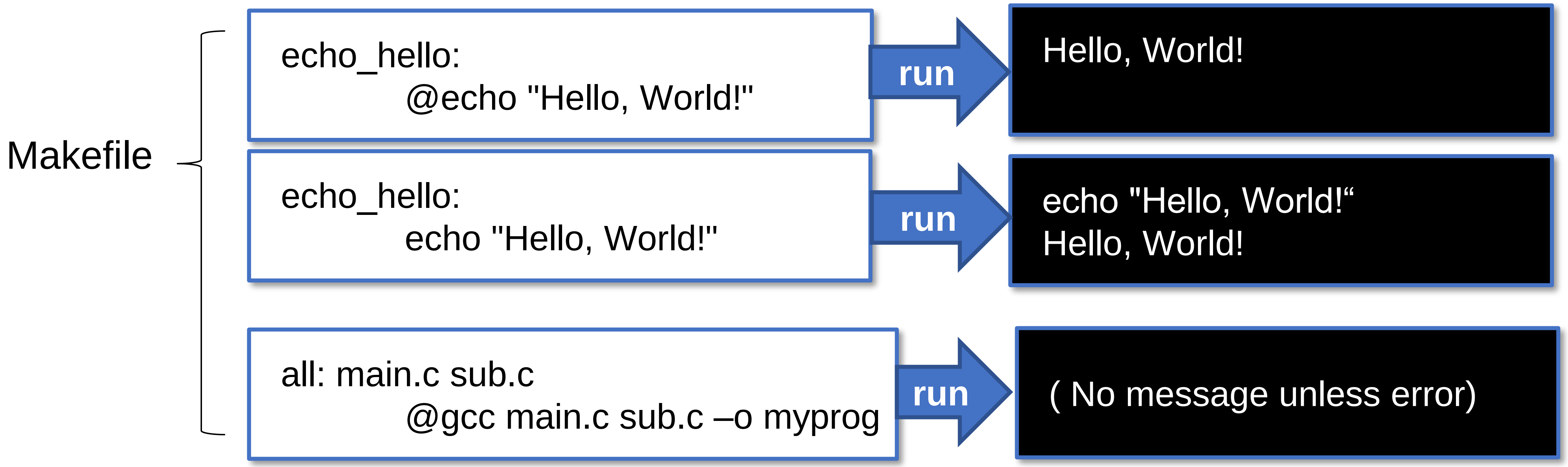
```
SRC_FILES := main.c util.c foo.c
OBJ_FILES := $(addprefix build/, $(SRC_FILES:.c=.o))
```

expand

```
SRC_FILES := main.c util.c foo.c
OBJ_FILES := build/main.o build/util.o build/foo.o
```

Makefile automatic variables : '@' sign

- The @ sign in a Makefile is used to **suppress the command echo**. By default, when you run make, each command that is executed is printed to the terminal before it is executed. If you don't want the command to be echoed, you can prefix it with @.
- Example:



```
echo_hello:
	@echo "Hello, World!"
```

```
Hello, World!
```

```
echo_hello:
	echo "Hello, World!"
```

```
echo "Hello, World!“
Hello, World!
```

```
all: main.c sub.c
	@gcc main.c sub.c –o myprog
```

```
( No message unless error)
```

Makefile automatic variables : '$@'

- $@ represents the full name of the target of the rule. The target is the file that the Makefile is currently trying to build.
- Example:

```
main.o: main.c
        gcc –c main.c –o $@
```

expand

```
gcc –c main.c –o main.o
```

```
SRC= main.c sub.c foo.c
OBJ= $(SRC:%.c=.o)
myprog: $(OBJ)
        gcc  -o $<  $(OBJ)
```

expand

```
gcc –c main.c –o main.o
```

Makefile automatic variables : ‘$<’

- $< represents the first prerequisite of the rule. The prerequisite is a file or set of files that the target depends on.
- Example:

```
%.o: %.c
        gcc  -c $< –o $@
```

expand

```
gcc –c main.c –o main.o
```

  - %.o is the target pattern (e.g., main.o).
  - %.c is the prerequisite pattern (e.g., main.c)
  - $< will expand to the first prerequisite, which is the source file (main.c)
  - $@ will expand to the target, which is the object file (main.o)

References

Dive into Systems

17.5 make and Makefiles (https://diveintosystems.org/book/Appendix2/makefiles.html)

Makefile official manual (https://www.gnu.org/software/make/manual/make.html)

Understanding and Using Makefile Flags (https://earthly.dev/blog/make-flags)

Cmake (https://cmake.org) (https://github.com/ttroy50/cmake-examples)