A Guide to Makefiles

OVERVIEW

A Makefile is a list of instructions to compile your code into an executable file.

You should make a **rule** to compile each of your source files,

Rules have the following format:

Example 1

```
target: ingredients
    recipe
```

this must be a TAB

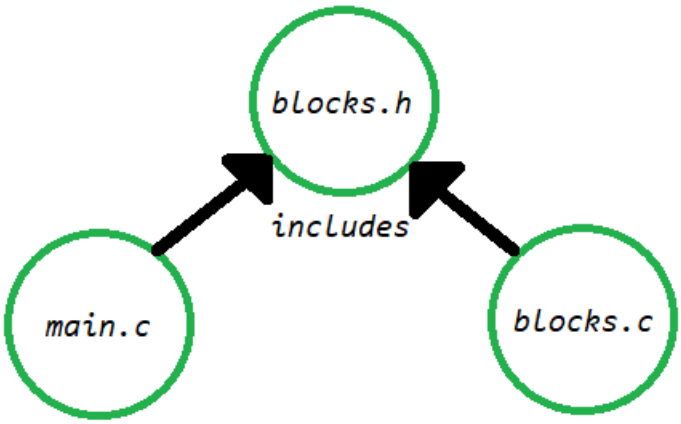
target:
the name of the item being made

ingredients:
the files needed to make the item

recipe:
the commands ran to make the item

As an example, let's compile the following three files. . .

Example 2



- main.c contains the **driver** program
- blocks.c contains the **implementation** of many useful functions
- blocks.h is a **header** file containing the declarations of these functions

We can compile these files into an executable called “driver” using the following three rules. . .

Example 3

```
a  driver: main.o blocks.o
       gcc -Wall -std=c99 -o driver main.o blocks.o

b  main.o: main.c
       gcc -Wall -std=c99 -c main.c -o main.o

c  blocks.o: blocks.c
       gcc -Wall -std=c99 -c blocks.c -o blocks.o
```

The first rule (a) is used to make the **executable.**

The second and third rule (b & c) are used to make the intermediate **object files.**

The executable can be ran using the terminal command. . .

```
./driver
```

The object files are incomplete binaries and should not be ran. . .

You may also add a “make clean” rule in the makefile.

```
clean:
    rm -f driver main.o blocks.o
```

This removes the object files and executable form the source directory.
(the “-f ” prevents the command from complaining)

Type “make” to compile the code.

Type “make clean” to remove the compiled code.

MACROS

These are shorthand definitions for Makefile syntax.

```
CC = gcc
CFLAGS = -Wall -std=c99
OBJECTS = main.o blocks.o
```

Example 4

With these defined at the top of the Makefile, we can rewrite our rules.

Here is an example using the executable rule from example 3a:

```
driver: $(OBJECTS)
    $(CC) $(CFLAGS) -o driver $(OBJECTS)
```

The order of these commands and macros matter!

- gcc is used for C files, and g++ is used for C++. These should appear first since they specify the compiler. gcc stands for ***GNU Compiler Collection***.

- There are many CFLAGS that you could use; these are options for the compiler to use. See the following table:

-Wall	**-Werror**	**-pedantic**
displays all warnings for syntax and semantic faults. This should always be used.	treats all warnings as errors.	displays even the smallest of errors. This tag may be annoying, but it helps you make clean code.
-c [*filename*]	**-o [*name*] [*object_list*]**	**-std=c99 *or* -std=c++14**
generate linkable (.o) object code from source filename.	redirect the object code and link other objects together into an executable *name*.	specifies which language standard to use during compilation.
-O0	**-O1 *or* -O2 *or* -O3**	**-OS**
optimization for compile time.	optimization for execution speed. Level 1 is the safest, 3 is the highest	optimization for file size.

OBJECTS & EXECUTABLES

Each source file should have a rule to compile it into an object file.

Example **3b** and **3c** can be rewritten in the following generic format:

Example 5

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

%.o is a placeholder for the name of an object file

%.c is a placeholder for the name of a C source file (you can also use **%.cpp**)

$@ is a wildcard meaning “for all”

$< is a wildcard that pulls-in the name of the source file

Effectively, this example accomplishes the following:

For all .c files, compile them into .o files with their respective names.

Once all of the objects files are made,

they can then be linked together into an executable.

Rules for executables are much simpler . . .

Example 6

```
target: $(OBJECTS)
	$(CC) $(CFLAGS) -o name_of_executable $(OBJECTS)
```

- Usually, the name of the target and the name of the executable should match.
- If you are making more than one executable, they may require different object files from different sources.
- The first argument after the “-o” is the executable name, all others are the files that you are linking to the executable.

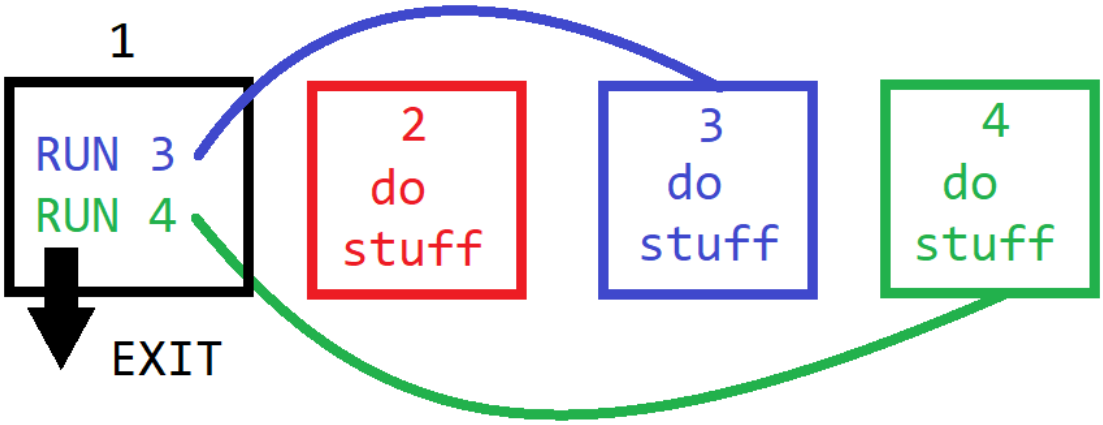
MAKE ALL

If you want to make multiple executables —such as a driver, some unit tests, and helper programs— there are a few things to note.

Makefiles do not execute in sequence. They follow a rule-based paradigm.

For example:

Example 7



Rule-1 calls for rule-3 & rule-4 before exiting, so rule-2 will never run.

To alleviate this problem, we can use a “make all” rule.

```
all: rule1 rule2 rule3 rule4
```

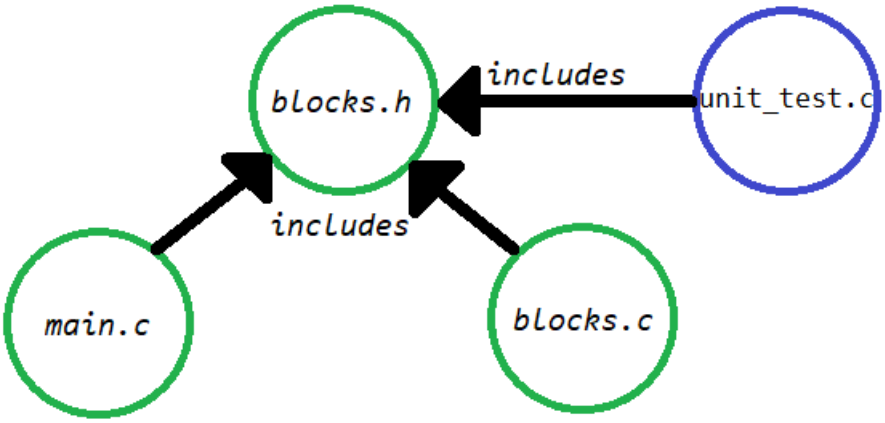
This will make sure all the executables are compiled.

Notice that the executables are written in the ingredients list, not the recipe. This is because makefile rules will “skip” forward in order to build the necessary prerequisite ingredients before making the recipe.

> Before a baker makes bread, they first need to collect their ingredients.

CUMULATIVE EXAMPLE

This example implements a unit test along with the main driver




```
# macros
CC = gcc
CFLAGS = -Wall -Werror -pedantic -std=c99
T_OBJECTS = unit_test.o blocks.o
D_OBJECTS = main.o blocks.o

# make all rule
all: driver unit_test

# executables
driver: $(D_OBJECTS)
     $(CC) $(CFLAGS) -o driver $(D_OBJECTS)
unit_test: $(T_OBJECTS)
     $(CC) $(CFLAGS) -o driver $(T_OBJECTS)

# generic rule for object files
%.o: %.c
     $(CC) $(CFLAGS) -o $@ $<

# make clean rule
clean:
     rm -f main.o blocks.o unit_test.o driver unit_test
```

What's happening?

main.c	→ main.o	→ driver
unit_test.c	→ unit_test.o	→ unit_test
blocks.c	→ blocks.o	→ driver & unit_test

TROUBLESHOOTING

- You can only have one Makefile *per folder*, and it must be named "**M**akefile" with a capital 'M'.

- The *terminal command* must be typed "**m**ake" with a lowercase 'm'.

- You must use a TAB before the recipe. Some text editors automatically insert spaces in place of a TAB; be aware of this.

- The pound sign, '#', is used for comments.

- A function can be used in the recipe to output text. $(info [your_text_here])

```
    rm -f driver $(info files have been removed)
```

- Use a forward slash '/' to continue a long command onto the next line.

```
    $(CC) $(CFLAGS) -o driver file1.o file2. file3.o /
        file4.o file5.o file 6.o
```

- If you are taking COMP.2040, you may need to include libraries such as -lsfml-graphics or -lboost_regex. These should be appended to the tail of the executable rule; this tells the compiler to link them to your objects.

```
    $(CC) $(CFLAGS) -o driver $(OBJ) -lsfml-graphics
```

Further study materials

Web tutorial
https://www.tutorialspoint.com/makefile/index.htm

Article
https://makefiletutorial.com/

Book
https://www.gnu.org/software/make/manual/make.html

Video
https://www.youtube.com/watch?v=_r7i5X0rXJk