**Makefile**

**Why makefile?**

When project is bigger, everytime we change a thing that lead to watse a lot of time to compile and rebuild. So we need to split program into many smaller parts and rebuild only that part. That is the purpose of makefile, we can point to a specific part to be compiled and rebuilt to save a lot of time.

Run makefile syntax: make –f ‘nameofmakefile.mk’

**Syntax:** [**https://makefiletutorial.com/**](https://makefiletutorial.com/)[**https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/**](https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/)

targets: prerequisites/dependencies

[tab only]command

command

command

The targets are file names, separated by spaces. Typically, there is only one per rule. Target is only for checking its existance.

The commands are a series of steps typically used to make the target(s). These need to start with a tab character, ~~not spaces~~.

The prerequisites are also file names, separated by spaces. These files need to exist before the commands for the target are run. These are also called dependencies. Dependency is for checking update timestamp of these file.

If targets is not available + no dependencies/has target + dependencies updated, command will run

If dependencies is blank, target will only build 1 time (despite changes later). When we put file into dependencies makefile will check the update of that file and rebuild it if there is changes. When dependency is called and the file is not available -> check below code to make sure that file is compiled, otherwise it will occur an error.

Example:

* Make selects the target blah, because the first target is the default target
* blah requires blah.o, so make searches for the blah.o target
* blah.o requires blah.c, so make searches for the blah.c target
* blah.c has no dependencies, so the echo command is run
* The cc -c command is then run, because all of the blah.o dependencies are finished
* The top cc command is run, because all the blah dependencies are finished
* That's it: blah is a compiled c program

blah: blah.o

cc blah.o -o blah # Runs third

blah.o: blah.c

cc -c blah.c -o blah.o # Runs second

# Typically blah.c would already exist, but I want to limit any additional required files

blah.c:

echo "int main() { return 0; }" > blah.c # Runs first

If we delete blah.c, all three targets will be rerun. If we edit it (and thus change the timestamp to newer than blah.o), the first two targets will run. If we run touch blah.o (and thus change the timestamp to newer than blah), then only the first target will run. If you change nothing, none of the targets will run.

This next example doesn't do anything new, but is nontheless a good additional example. It will always run both targets, because some\_file depends on other\_file, which is never created.

some\_file: other\_file

echo "This will always run, and runs second"

touch some\_file

other\_file:

echo "This will always run, and runs first"

Make clean

clean is often used as a target that removes the output of other targets, but it is not a special word in Make. You can run make and make clean on this to create and delete some\_file.

Note that clean is doing two new things here:

* It's a target that is not first (the default), and not a prerequisite. That means it'll never run unless you explicitly call make clean
* It's not intended to be a filename. If you happen to have a file named clean, this target won't run, which is not what we want. See .PHONY later in this tutorial on how to fix this

some\_file:

touch some\_file

clean:

rm -f some\_file

**Variables (Macro)**

Only be string, single ‘ or double quote “ is still string. Use = or := to assign string into var

Call var using $(var)/${var}

Ex:

files := file1 file2

some\_file: $(files)

**Targets:**

The all target

Making multiple targets and you want all of them to run? Make an all target. Since this is the first rule listed, it will run by default if make is called without specifying a target.

all: one two three

one:

touch one

two:

touch two

three:

touch three

clean:

rm -f one two three

**Automatic variable**:

$@ is the name of the target being generated, and $< the first prerequisite (usually a source file). You can find a list of all these special variables in the [GNU Make manual](https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables).

For example, consider the following declaration:

Test.o fu.o: library.cpp main.cpp

In this case:

* $@ evaluates to test.o fu.o
* $< evaluates to library.cpp
* $^ evaluates to library.cpp main.cpp
* $? is the name of all the prerequisites that are newer than the target, with spaces between them

<https://stackoverflow.com/questions/3220277/what-do-the-makefile-symbols-and-mean>

**Wildcard** \*

Both \* and % are called wildcards in Make, but they mean entirely different things. \* searches your filesystem for matching filenames. I suggest that you always wrap it in the wildcard function, because otherwise you may fall into a common pitfall described below.

# Print out file information about every .c file

print: $(wildcard \*.c)

ls -la $?

\* may be used in the target, prerequisites, or in the wildcard function.

Danger: \* may not be directly used in a variable definitions

Danger: When \* matches no files, it is left as it is (unless run in the wildcard function)

thing\_wrong := \*.o # Don't do this! '\*' will not get expanded

thing\_right := $(wildcard \*.o)

all: one two three four

# Fails, because $(thing\_wrong) is the string "\*.o"

one: $(thing\_wrong)

# Stays as \*.o if there are no files that match this pattern :(

two: \*.o

# Works as you would expect! In this case, it does nothing.

three: $(thing\_right)

# Same as rule three

four: $(wildcard \*.o)

**Fancy rule**

* Implicit rule
* Compiling a C program: n.o is made automatically from n.c with a command of the form $(CC) -c $(CPPFLAGS) $(CFLAGS) $^ -o $@
* Compiling a C++ program: n.o is made automatically from n.cc or n.cpp with a command of the form $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $^ -o $@
* Linking a single object file: n is made automatically from n.o by running the command $(CC) $(LDFLAGS) $^ $(LOADLIBES) $(LDLIBS) -o $@

The important variables used by implicit rules are:

* CC: Program for compiling C programs; default cc
* CXX: Program for compiling C++ programs; default g++
* CFLAGS: Extra flags to give to the C compiler
* CXXFLAGS: Extra flags to give to the C++ compiler
* CPPFLAGS: Extra flags to give to the C preprocessor
* LDFLAGS: Extra flags to give to compilers when they are supposed to invoke the linker
* Static pattern rule

objects = foo.o bar.o all.o

all: $(objects)

$(objects): %.o: %.c: targets \*.o with dependencies \*.c (only with name is included in (objects))

! Note: %.o : %.c is still a valid syntax that will excecute all \*.c file in that folder

$(filter %.o,$(obj\_files)): %.o: %.c : same with ($(objects): %.o: %.c)

**A Simple Makefile Tutorial**

Makefiles are a simple way to organize code compilation. This tutorial does not even scratch the surface of what is possible using *make*, but is intended as a starters guide so that you can quickly and easily create your own makefiles for small to medium-sized projects.

**A Simple Example**

Let's start off with the following three files, hellomake.c, hellofunc.c, and hellomake.h, which would represent a typical main program, some functional code in a separate file, and an include file, respectively.

|  |  |  |
| --- | --- | --- |
| **hellomake.c** | **hellofunc.c** | **hellomake.h** |
| #include <hellomake.h>  int main() {  // call a function in another file  myPrintHelloMake();  return(0);  } | #include <stdio.h>  #include <hellomake.h>  void myPrintHelloMake(void) {  printf("Hello makefiles!\n");  return;  } | /\*  example include file  \*/  void myPrintHelloMake(void); |

Normally, you would compile this collection of code by executing the following command:

gcc -o hellomake hellomake.c hellofunc.c -I.

This compiles the two .c files and names the executable hellomake. The -I. is included so that gcc will look in the current directory (.) for the include file hellomake.h. Without a makefile, the typical approach to the test/modify/debug cycle is to use the up arrow in a terminal to go back to your last compile command so you don't have to type it each time, especially once you've added a few more .c files to the mix.

Unfortunately, this approach to compilation has two downfalls. First, if you lose the compile command or switch computers you have to retype it from scratch, which is inefficient at best. Second, if you are only making changes to one .c file, recompiling all of them every time is also time-consuming and inefficient. So, it's time to see what we can do with a makefile.

The simplest makefile you could create would look something like:

[Makefile 1](https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/makefile.1)

hellomake: hellomake.c hellofunc.c

gcc -o hellomake hellomake.c hellofunc.c -I.

If you put this rule into a file called Makefile or makefile and then type make on the command line it will execute the compile command as you have written it in the makefile. Note that make with no arguments executes the first rule in the file. Furthermore, by putting the list of files on which the command depends on the first line after the :, make knows that the rule hellomake needs to be executed if any of those files change. Immediately, you have solved problem #1 and can avoid using the up arrow repeatedly, looking for your last compile command. However, the system is still not being efficient in terms of compiling only the latest changes.

One very important thing to note is that there is a tab before the gcc command in the makefile. There must be a tab at the beginning of any command, and make will not be happy if it's not there.

In order to be a bit more efficient, let's try the following:

[Makefile 2](https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/makefile.2)

CC=gcc

CFLAGS=-I.

hellomake: hellomake.o hellofunc.o

$(CC) -o hellomake hellomake.o hellofunc.o

So now we've defined some constants CC and CFLAGS. It turns out these are special constants that communicate to make how we want to compile the files hellomake.c and hellofunc.c. In particular, the macro CC is the C compiler to use, and CFLAGS is the list of flags to pass to the compilation command. By putting the object files--hellomake.o and hellofunc.o--in the dependency list and in the rule, make knows it must first compile the .c versions individually, and then build the executable hellomake.

Using this form of makefile is sufficient for most small scale projects. However, there is one thing missing: dependency on the include files. If you were to make a change to hellomake.h, for example, make would not recompile the .c files, even though they needed to be. In order to fix this, we need to tell make that all .c files depend on certain .h files. We can do this by writing a simple rule and adding it to the makefile.

[Makefile 3](https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/makefile.3)

CC=gcc

CFLAGS=-I.

DEPS = hellomake.h

%.o: %.c $(DEPS)

$(CC) -c -o $@ $< $(CFLAGS)

hellomake: hellomake.o hellofunc.o

$(CC) -o hellomake hellomake.o hellofunc.o

This addition first creates the macro DEPS, which is the set of .h files on which the .c files depend. Then we define a rule that applies to all files ending in the .o suffix. The rule says that the .o file depends upon the .c version of the file and the .h files included in the DEPS macro. The rule then says that to generate the .o file, make needs to compile the .c file using the compiler defined in the CC macro. The -c flag says to generate the object file, the -o $@ says to put the output of the compilation in the file named on the left side of the :, the $< is the first item in the dependencies list, and the CFLAGS macro is defined as above.

As a final simplification, let's use the special macros $@ and $^, which are the left and right sides of the :, respectively, to make the overall compilation rule more general. In the example below, all of the include files should be listed as part of the macro DEPS, and all of the object files should be listed as part of the macro OBJ.

[Makefile 4](https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/makefile.4)

CC=gcc

CFLAGS=-I.

DEPS = hellomake.h

OBJ = hellomake.o hellofunc.o

%.o: %.c $(DEPS)

$(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)

$(CC) -o $@ $^ $(CFLAGS)

So what if we want to start putting our .h files in an include directory, our source code in a src directory, and some local libraries in a lib directory? Also, can we somehow hide those annoying .o files that hang around all over the place? The answer, of course, is yes. The following makefile defines paths to the include and lib directories, and places the object files in an obj subdirectory within the src directory. It also has a macro defined for any libraries you want to include, such as the math library -lm. This makefile should be located in the src directory. Note that it also includes a rule for cleaning up your source and object directories if you type make clean. The .PHONY rule keeps make from doing something with a file named clean.

[Makefile 5](https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/makefile.5)

IDIR =../include

CC=gcc

CFLAGS=-I$(IDIR)

ODIR=obj

LDIR =../lib

LIBS=-lm

\_DEPS = hellomake.h

DEPS = $(patsubst %,$(IDIR)/%,$(\_DEPS))

\_OBJ = hellomake.o hellofunc.o

OBJ = $(patsubst %,$(ODIR)/%,$(\_OBJ))

$(ODIR)/%.o: %.c $(DEPS)

$(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)

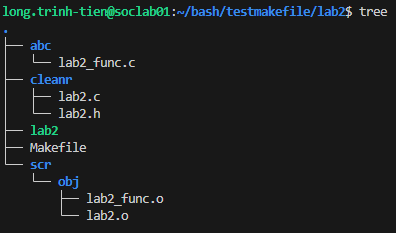
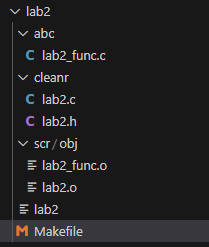
$(CC) -o $@ $^ $(CFLAGS) $(LIBS)

.PHONY: clean

clean:

rm -f $(ODIR)/\*.o \*~ core $(INCDIR)/\*~

Example: compile file with structure below (use ‘tree’ command to see file structure) (this is not ‘lab2’ ex)



CC = gcc

DEPDIR = ./cleanr

SRCDIR = ./abc

SRCDIR\_2 = ./cleanr

OBJDIR = ./scr/obj

OBJ = $(OBJDIR)/lab2.o $(OBJDIR)/lab2\_func.o

DEPS = $(wildcard $(DEPDIR)/\*.h)

TARGET = lab2

.PHONY: all clean

all: $(TARGET)

$(TARGET): $(OBJ)

    $(CC) -o $@ $^

$(OBJDIR)/%.o: $(SRCDIR)/%.c $(DEPS)

    $(CC) -c -o $@ $<

$(OBJDIR)/%.o: $(SRCDIR\_2)/%.c $(DEPS)

    $(CC) -c -o $@ $<

clean:

    rm -rf $(OBJDIR)/\*.o $(TARGET)

!note:

Nếu không có cờ **-Iinclude** trong Makefile, trình biên dịch sẽ không biết cách tìm kiếm các file header trong thư mục **include** khi biên dịch các file mã nguồn. Kết quả là, khi trình biên dịch gặp các lệnh sử dụng các hàm hoặc biến được khai báo trong các file header mà không thể tìm thấy chúng, sẽ phát sinh lỗi "file not found" hoặc "undefined reference".

Để giải quyết vấn đề này, có thể thêm đường dẫn đầy đủ đến thư mục **include** trong các lệnh biên dịch hoặc sửa lại Makefile để bao gồm cờ **-Iinclude**.

Nếu các file header nằm trong các thư mục con như **/include/include2/include3** và **/dinclude/include21**, bạn cũng có thể sử dụng cờ **-I** để chỉ định các đường dẫn đầy đủ đến các thư mục này. Ví dụ:

bashCopy code

CFLAGS := -I/include -I/dinclude/include21

Khi đó, trình biên dịch sẽ tìm kiếm các file header trong các thư mục **/include**, **/include/include2/include3** và **/dinclude/include21**.

Top of Form

**How to check if a directory doesn't exist in make and create it**

The only correct way to do this is with **order-only-prerequisites**, see: <https://www.gnu.org/software/make/manual/html_node/Prerequisite-Types.html>. Note the **|** in the snippet.

If you don't use order-only-prerequisites each modification (e.g. coping or creating a file) in that directory will trigger the rule that **depends** on the directory-creation target again!

object/%.o: code/%.cc | object

compile $< somehow...

object:

mkdir -p $@