1. What is a makefile? A makefile is a file containing a set of directives used by make on how to compile and link a program. Make is a build automation tool that automatically builds executable programs and libraries from source code by reading files and makefiles, which specify how to derive the target program. A makefile works upon the principle that files only need recreating if their dependencies are newer than the file being created/recreated. The makefile is recursively carried out (with dependency prepared before each target depending upon them) until everything has been updated (that requires updating) and the primary/ultimate target is complete. These instructions with their dependencies are specified in a makefile. If none of the files that are prerequisities have been changed since the last time the program was compiled, no actions take place.

(Source: https://en.wikipedia.org/wiki/Makefile)

A Simple Makefile Tutorial

(Source: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/)

Makefiles are a simple way to organize code compilation. This tutorial 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.

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

```
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

```
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

```
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

```
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

```
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)/*~
```

So now you have a perfectly good makefile that you can modify to manage small and medium-sized software projects. You can add multiple rules to a makefile; you can even create rules that call other rules. For more information on makefiles and the make function, check out the GNU Make Manual, which will tell you more than you ever wanted to know (really).

Additional Guides

- Writing Makefiles: https://www.cs.bu.edu/teaching/cpp/writing-makefiles/
- What is a Makefile and how does it work? https://opensource.com/article/18/8/what-how-makefile

2. What is CMake? CMake is a platform free and open-source software application for managing the build process of software using a compiler-independent method. It supports hierarchies and applications that depend on multiple libraries. It is used in conjuction with native build environments. It has minimal dependencies, requiring only a C++ compiler on its own build system. Its build process consists of two stages. First, standard build files are created from configuration files. Then the platform's native build tools are used for the actual building. Each build project contains a CMakeLists.txt file in very directory that controls the build process. The CMakeLists.txt file has one or more commands in the form COMMAND (args…), with COMMAND representing the name of each command and "args" the list of arguments, each separated by white space. While there are many built-in rules for compiling the software libraries and executables, there are also provisions for custom build rules. Some build dependencies can be determined automatically. Advanced users can also create and incorporate additional makefile generators to support their specific compiler and OS needs.

(Source: https://en.wikipedia.org/wiki/CMake)

An Example CMakeLists.txt with comments

(Source: https://learnxinyminutes.com/docs/cmake/)

```
# In CMake, this is a comment

# To run our code, we will use these steps:
#  - mkdir build && cd build
#  - cmake ..
#  - make
#
# With those steps, we will follow the best practice to compile into a subdir
# and the second line will request to CMake to generate a new OS-dependent
# Makefile. Finally, run the native Make command.

#------------------------------------------------------------------------------
# Basic
#------------------------------------------------------------------------------
#
# The CMake file MUST be named as "CMakeLists.txt".

# Setup the minimum version required of CMake to generate the Makefile
cmake_minimum_required (VERSION 2.8)

# Raises a FATAL_ERROR if version < 2.8
cmake_minimum_required (VERSION 2.8 FATAL_ERROR)

# We setup the name for our project. After we do that, this will change some
# directories naming convention generated by CMake. We can send the LANG of
# code as second param
project (learncmake C)

# Set the project source dir (just convention)
set( LEARN_CMAKE_SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR} )
set( LEARN_CMAKE_BINARY_DIR ${CMAKE_CURRENT_BINARY_DIR} )

# It's useful to setup the current version of our code in the build system
# using a `semver` style
set (LEARN_CMAKE_VERSION_MAJOR 1)
set (LEARN_CMAKE_VERSION_MINOR 0)
set (LEARN_CMAKE_VERSION_PATCH 0)

# Send the variables (version number) to source code header
configure_file (
```

```
  "${PROJECT_SOURCE_DIR}/TutorialConfig.h.in"
  "${PROJECT_BINARY_DIR}/TutorialConfig.h"
)

# Include Directories
# In GCC, this will invoke the "-I" command
include_directories( include )

# Where are the additional libraries installed? Note: provide includes
# path here, subsequent checks will resolve everything else
set( CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/CMake/modules/" )

# Conditions
if ( CONDITION )
  # Output!

  # Incidental information
  message(STATUS "My message")

  # CMake Warning, continue processing
  message(WARNING "My message")

  # CMake Warning (dev), continue processing
  message(AUTHOR_WARNING "My message")

  # CMake Error, continue processing, but skip generation
  message(SEND_ERROR "My message")

  # CMake Error, stop processing and generation
  message(FATAL_ERROR "My message")
endif()

if( CONDITION )

elseif( CONDITION )

else( CONDITION )

endif( CONDITION )

# Loops
foreach(loop_var arg1 arg2 ...)
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
endforeach(loop_var)

foreach(loop_var RANGE total)
foreach(loop_var RANGE start stop [step])

foreach(loop_var IN [LISTS [list1 [...]]]
                    [ITEMS [item1 [...]]])

while(condition)
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
endwhile(condition)


# Logic Operations
if(FALSE AND (FALSE OR TRUE))
  message("Don't display!")
```

```
endif()

# Set a normal, cache, or environment variable to a given value.
# If the PARENT_SCOPE option is given the variable will be set in the scope
# above the current scope.
# `set(<variable> <value>... [PARENT_SCOPE])`

# How to reference variables inside quoted and unquoted arguments
# A variable reference is replaced by the value of the variable, or by the
# empty string if the variable is not set
${variable_name}

# Lists
# Setup the list of source files
set( LEARN_CMAKE_SOURCES
  src/main.c
  src/imagem.c
  src/pather.c
)

# Calls the compiler
#
# ${PROJECT_NAME} refers to Learn_CMake
add_executable( ${PROJECT_NAME} ${LEARN_CMAKE_SOURCES} )

# Link the libraries
target_link_libraries( ${PROJECT_NAME} ${LIBS} m )

# Where are the additional libraries installed? Note: provide includes
# path here, subsequent checks will resolve everything else
set( CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/CMake/modules/" )

# Compiler Condition (gcc ; g++)
if ( "${CMAKE_C_COMPILER_ID}" STREQUAL "GNU" )
  message( STATUS "Setting the flags for ${CMAKE_C_COMPILER_ID} compiler" )
  add_definitions( --std=c99 )
endif()

# Check for OS
if( UNIX )
    set( LEARN_CMAKE_DEFINITIONS
        "${LEARN_CMAKE_DEFINITIONS} -Wall -Wextra -Werror -Wno-deprecated-declarations -
Wno-unused-parameter -Wno-comment" )
endif()
```

CMake CheatSheet

(Source: http://blog.mbedded.ninja/programming/build-systems-and-package-managers/cmake/cmake-cheat-sheet)

Function	Explanation
`set(srcs file1.c file2.c ...)`	Creates a variable (e.g. src), and assigns something to it (e.g. the list file1.c file2.c). To clear a variable, do not provide second argument, e.g. set(srcs).
`include_directories(dir1 dir2 ...)`	Adds the provided directory paths to the compilers list of directories that it will search for include files in, for any following targets.
`add_library(name STATIC/SHARED/MODULE file1.c file2.c ...)`	Adds a library target that will be build from the provided source files. DO NOT APPEND lib_ to the name (this is done automatically by cmake depending on architecture).

`add_executable(name file1.c file2.c ...)`	Adds an executable target (as opposed to a library target).
`link_libraries(lib_1 lib_2 ...)`	Links the provided libraries to all following targets in the CMakeLists.txt file. This is deprecated. It is recommended you use target_link_libraries() instead.
`target_link_libraries(target_lib other_lib_1 other_lib_2 ...)`	Links the provided libraries to the specific target library. link_libraries() can be used to apply to libraries to all following targets (i.e. no specific target is provided), however, it is deprecated.
`install(TARGETS targets...)`	Used to place build output into certain directories on the user's system (as well as do things like assign privileges to these files).

Additional Guides

- CMake Tutorial: https://cmake.org/cmake-tutorial/
- CMake Reference Documentation: https://cmake.org/documentation/