

Makefile Guide

This document is a basic introduction to Makefiles. There is a lot to Makefiles and this document does not discuss everything. However, upon reading this document the following will happen:

- You will learn what Makefiles are and why they're used.
- You will learn the basics of Makefiles and will gain the ability to create Makefiles for your own programs. You will also learn the small (but important) differences between C and C++ Makefiles though they are relatively the same when it comes to the basics.
- You will be taught the *why* - instead of just learning how to create Makefiles in the absence of any understanding of what's going on, you will always be taught why you're doing something and what it is doing.
- You will learn some more advanced Makefile features that ultimately culminates in you being provided with some go-to C and C++ Makefiles that you can generically use for any C and C++ program.

Table of Contents

Topics	Page
Why Makefiles Are Used	2
Basic Makefile Setup	3
Makefile Variables	6
C vs. C++ Makefiles	7
Advanced Makefile Features	8
Go-To C and C++ Makefiles	12
Modifying The Go-To Makefiles	13
Resources	14

Why Makefiles Are Used

When a program is running, what is actually running is an **executable file**. The executable file is generated from source code. In C and C++, these are the **.c** and **.cpp** files respectively (along with any header files included with the source code files). Taking source code and turning it into an executable file is called **compiling** a program.

In IDEs like Microsoft Visual Studio or XCode, this process is done automatically. A button (like “Run”) is clicked and then the program just magically starts running. There was a lot that happened behind the scenes that you did not see - the code is compiled, the executable file is created, and then the executable file runs. When working in Linux, this process is not automated via the click of a single button. The program must be manually compiled by entering a series of commands in the command line. It is inefficient to manually type in each one of these commands every time a program is to be compiled, and a **Makefile** helps to automate this process - a single command can be entered into the command line, the Makefile will recognize that command, and then it will automatically run all of the other commands to compile the program and create an executable file. Now that the executable file is made, like *driver* for example, the program can be run by entering *./driver*

An executable file can be directly created from source code files, but the best practice is actually to first create object files from the source code files, and then use the object files to create the executable file.

source code files → object files → executable file

In summary

- IDEs
 - Compile the code, create an executable file, and run the executable file all with the single click of a button.
- Linux
 - Compile the code and create the executable file with a Makefile.
 - Run the executable file by entering *./executable_file_name* in the command line.

Basic Makefile Setup

The general format of a C Makefile is seen below. The logic here can also be applied to C++ programs for which there are only small differences discussed later.

target: ingredients
recipe

target the name of the item being made (like an executable file)
ingredients all of the object files needed to make the item
recipe the commands to make the executable file - note that this must be indented with a tab not spaces

For a program that consisted of only a *main.c* file, the Makefile would look like the following:

```
driver: main.o
    gcc --std=c99 -g -Wall -o driver main.o
main.o: main.c
    gcc --std=c99 -g -Wall -c main.c -o main.o
```

Below is a breakdown of each component shown above:

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

- driver: main.o
 - driver is the target. When in the command line and you enter *make driver*, the Makefile is going to execute all of the commands below the target until the end of the Makefile is reached or another target is encountered. Note that if you just enter *make* in the command line, it will execute the first target that is in the Makefile. So, to create *driver*, you could enter *make driver* or *make* so long as *driver* were the first target in the Makefile.
 - main.o is the object file that the executable file relies on (it can rely on more than one as seen in later examples).
- gcc is the C compiler being used. There are other compilers like clang
- --std=c99 -g -Wall are all **flags** which are optional but always helpful features.
 - --std=c99 specifies that the C99 version of C is being used. There are other versions of C as well.
 - -g enables debugging options for the compiler. This is most commonly used to check for memory leaks with **valgrind**.
 - -Wall will look for some easily fixable issues in your code and display them as warnings during compile time.
- -o driver main.o
 - -o driver means name the executable file *driver*. If this were not here, the executable file would be named *a.out* by default. It is a convention to name the target and the executable file the same in some circumstances.
 - main.o is present since all object files the executable relies on need to be listed

Now, for every object file that needs to be made (in this case 1 object file), there needs to be a command for it as seen on the next page.

```
main.o: main.c
    gcc --std=c99 -g -Wall -c main.c -o main.o
```

- `main.o: main.c`
 - `main.o` is the object file being created
 - `main.c` is the source code file that will be used to create `main.o`
- `gcc --std=c99 -g -Wall -c main.c -o main.o`
 - `gcc`, `--std=c99`, `-g`, and `-Wall` all mean the same thing as previously described
 - `-c main.c` means compile the `main.c` source code file to create the object code file
 - `-o main.o` means name the object file `main.o`

The Makefile for a program that contains more source code files than just `main.c` can then be built off of that simple example. For every additional source code file, an object file needs to get created - simply write a command for it. For example, let's say that the previous program also included a header and implementation file named `example.h` and `example.c`. Then, the Makefile would look like the following:

```
driver: main.o example.o
    gcc --std=c99 -g -Wall -o driver main.o example.o
main.o: main.c
    gcc --std=c99 -g -Wall -c main.c -o main.o
example.o: example.c
    gcc --std=c99 -g -Wall -c example.c -o example.o
```

In addition to creating a target to create the executable file, a target should be made to “clean up” or remove the executable file and object files made.

```
driver: main.o example.o
    gcc --std=c99 -g -Wall -o driver main.o example.o
main.o: main.c
    gcc --std=c99 -g -Wall -c main.c -o main.o
example.o: example.c
    gcc --std=c99 -g -Wall -c example.c -o example.o

clean:
    rm driver main.o example.o
```

- The `clean` is the target - it can be named anything but `clean` is a convention
- The `rm` stands for remove (delete) and it is followed by the items that should be deleted which are the executable and object files.

Now, to create the executable file, enter `make` or `make driver` in the command line.

To run the executable file, enter `./driver` in the command line or `valgrind ./driver` if you wanted to check for memory leaks.

To remove the executable file and object files, enter `make clean` in the command line.

If the code is modified, enter `make` again to recreate the executable file with the updated code.

A further discussion on the `rm` command - in the above example, `rm` was by itself. Two other things commonly done are to place a dash before it like `-rm` and to place a `-f` after it like `rm -f`. An explanation of the differences are below:

- `rm` (no dash)
 - Any errors that occur are not ignored.
 - The current command will finish executing but after that the Makefile aborts. If there were any commands in the target following that one, they would not get executed.
 - An error message will display.
 - An example of an error would be trying to delete a file that doesn't exist. For example, let's say before entering *make clean* you had manually deleted *main.o*. In that case, when the line `rm driver main.o example.o` executes, *main.o* doesn't exist. What will happen is the command will finish executing so *driver* and *example.o* will be used, but if there were any commands below it, they would not be executed. Thus, all of the commands in the target will not get executed. For example


```
clean:
    rm driver main.o example.o
    rm stuff
```

 The `rm stuff` command would not get executed.
- `-rm` (with a dash)
 - Any errors that occur are ignored.
 - The current command will finish executing but the Makefile does not abort. If there were any commands in the target following that one, they would get executed.
 - An error message will display, but it will also include the word *ignore*. It is informing you that an error happened, but it's ignoring the error.
 - Using the above example, the `rm stuff` command would get executed.
- `rm -f`
 - This is the same as `-rm` in that it ignores errors, but it does not display an error message.

When choosing which option to use for basic Makefiles, an argument could be made for `-rm` since it will not abort the execution of any following commands (unlike `rm`) and it will inform you of any errors (unlike `rm -f`). This way, the execution of the target continues and you are simultaneously informed of any errors that happen. In this document, `-rm` is used in all following examples.

Makefile Variables

Variables are often used in Makefiles which are shorthand definitions for Makefile syntax. They can be thought of just like the macros in C and are essentially text replacement. For example:

```
SOME_VAR = something1 something2
```

Now, instead of having to write out *something1 something2* every time you can just write `$(SOME_VAR)`

The previous Makefile example is rewritten below with some common use cases of variables - one for the compiler, one for the flags, one for the object files, and one for the executable file and target. Again, the executable file and target don't have to be named the same but it's a convention in this case - one reason would be (as seen below) that a single variable can be used to refer to both of them.

```
CC = gcc
CFLAGS = --std=c99 -g -Wall
OBJ = main.o example.o
EXE = driver

$(EXE): $(OBJ)
    $(CC) $(CFLAGS) -o $(EXE) $(OBJ)
main.o: main.c
    $(CC) $(CFLAGS) -c main.c -o main.o
example.o: example.c
    $(CC) $(CFLAGS) -c example.c -o example.o

clean:
    -rm $(EXE) $(OBJ)
```

The benefits of variables can be seen in the above example

- Less has to be written - you can just say `$(CFLAGS)` instead of `--std=c99 -g -Wall`
- If anything is changed, it will be automatically adjusted in the Makefile. Let's say you decided to add in another flag like *Wpedantic*. Notice how `$(CFLAGS)` is written in 3 places. With a variable, you simply have to add it to the `CFLAGS` variable

```
CFLAGS = --std=c99 -g -Wall -Wpedantic
```

and it automatically gets added everywhere `$(CFLAGS)` appears. Without a variable, you would have to manually add *Wpedantic* to all of those 3 places.

C vs. C++ Makefiles

The keyword **gcc** stands for GNU Compiler Collection most commonly referred to as GCC.

For C

- Use **gcc** which invokes the GCC compiler for C programs.
- Specify which version of C you're using like **--std=c99**. Note the two dashes.
- File extensions are **.c** for source code files and **.h** for header files.
- *Versions (as of the time of this document's creation)*: K&R (the original version), C89 (also referred to as ANSI C), C90, C95, C99, C11, and C17.
- Use the variables **CC** and **CFLAGS**.

For C++

- Use **g++** which invokes the GCC compiler for C++ programs.
- Specify which version of C++ you're using like **-std=c++11**. Note the one dash.
- File extensions are **.cpp** for source code files and **.h** or **.hpp** for header files. Note how **.h** is for C or C++ header files whereas **.hpp** is exclusively for C++ header files.
- *Versions (as of the time of this document's creation)*: C++98, C++03, C++11, C++14, C++17, and C++20.
- Use the variables **CXX** and **CXXFLAGS**

Commonly Used Flags

-Wall: Enables all warnings that are considered questionable and easy to avoid.

-g: Enable debugging options - use this if you want to use valgrind to check for memory leaks. If you manually manage resources on the heap in a C or C++ program, you should always use valgrind to ensure there are no memory leaks. Memory leaks are a serious bug in a program.

-Wextra: Enables some extra warning flags that are not enabled by **-Wall**.

-Werror: Make all warnings into errors - a warning will now cause compilation failure.

-Wpedantic: Issue all the warnings demanded by the strict ISO C and ISO C++. Reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++.

-O0: Default mode - reduce compilation time and make debugging produce the expected results.

-O1: Optimize - the compiler tries to reduce code size and execution time without performing any optimizations that take a great deal of compilation time.

-O2: More optimization - the compiler performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to O1, this option increases compilation time and the performance of the generated code.

-O3: Optimize even more

-OS: Optimize for file size

-Og: Optimize for debugging

-fsanitize=undefined: The code detects some undefined behavior that valgrind won't catch. Makes the code slower but is okay for a debug build.

-lm: Needed if using the C **<math.h>** library. Add to the first command

driver: **main.o**

gcc --std=c99 -g -Wall -o driver main.o -lm

The Linux Man Pages provide a comprehensive overview of all flags.

Advanced Makefile Features

There are some advanced Makefile features that can be added to a Makefile to help it be more concise and generic. Instead of having to write a command to create every single object file, the features listed below can be used to write one generic command to create all of the object files.

- `%o` is a placeholder for the name of an object file
- `%.c` is a placeholder for the name of a C source code file
- `%.cpp` is a placeholder for the name of a C++ source code file
- `$(@)` is the name of the target being generated (used in place of object file name)
- `$<` is the name of the first prerequisite (used in place of the source file named)

Note that `$(@)` and `$<` are called **automatic variables**.

Below is the finalized Makefile from page 4 both before and after the changes.

Before changes

```
CC = gcc
CFLAGS = --std=c99 -g -Wall
OBJ = main.o example.o
EXE = driver

$(EXE): $(OBJ)
    $(CC) $(CFLAGS) -o $(EXE) $(OBJ)
main.o: main.c
    $(CC) $(CFLAGS) -c main.c -o main.o
example.o: example.c
    $(CC) $(CFLAGS) -c example.c -o example.o

clean:
    -rm $(EXE) $(OBJ)
```

After changes

```
CC = gcc
CFLAGS = --std=c99 -g -Wall
OBJ = main.o example.o
EXE = driver

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

clean:
    -rm $(EXE) $(OBJ)
```

Note how one single generic command is used to create *main.o* and *example.o*

These features are even more helpful if you have multiple targets you want to execute. For example, consider adding in a unit test for the *example.h/example.c* interface. Now, *example.h/example.c/main.c* are used to create the *driver* executable file to run the program, and *example.h/example.c/unit_test.c* are used to create *unit_test* which is the executable file for doing some unit testing. In this case, *unit_test.c* contains a main function for the unit testing. Without the more advanced features, the Makefile would look like the one below. Note how for each executable file, there is a target to create it and a clean to remove the executable files and its associated object files.

```
CC = gcc
CFLAGS = --std=c99 -g -Wall
DRIVER_OBJ = main.o example.o
UNIT_TEST_OBJ = unit_test.o example.o

driver: $(DRIVER_OBJ)
    $(CC) $(CFLAGS) -o driver $(DRIVER_OBJ)
main.o: main.c
    $(CC) $(CFLAGS) -c main.c -o main.o
example.o: example.c
    $(CC) $(CFLAGS) -c example.c -o example.o

unit_test: $(UNIT_TEST_OBJ)
    $(CC) $(CFLAGS) -o unit_test $(UNIT_TEST_OBJ)
unit_test.o: unit_test.c
    $(CC) $(CFLAGS) -c unit_test.c -o unit_test.o
example.o: example.c
    $(CC) $(CFLAGS) -c example.c -o example.o

clean_driver:
    -rm driver $(DRIVER_OBJ)

clean_unit_test:
    -rm unit_test $(UNIT_TEST_OBJ)
```

<i>make</i> or <i>make driver</i>	Create the <i>driver</i> executable file
<i>make clean_driver</i>	Delete <i>driver</i> and its associated object files

<i>make unit_test</i>	Create the <i>unit_test</i> executable file
<i>make clean_unit_test</i>	Delete <i>unit_test</i> and its associated object files

Though the above example is a completely valid and fine Makefile, it could be rewritten much more concisely using some of the advanced features. This is seen on the next page. In the example, some other features are being introduced:

- *.o The * is a wildcard which says to search the file system for matching file names, in this case files with the .o file extension.
- all This is a target which specifies that other targets should be made

```
CC = gcc
CFLAGS = --std=c99 -g -Wall
DRIVER_OBJ = main.o example.o
UNIT_TEST_OBJ = unit_test.o example.o
EXE = driver unit_test
```

```
all: $(EXE)
```

```
driver: $(DRIVER_OBJ)
```

```
$(CC) $(CFLAGS) -o driver $(DRIVER_OBJ)
```

```
unit_test: $(UNIT_TEST_OBJ)
```

```
$(CC) $(CFLAGS) -o unit_test $(UNIT_TEST_OBJ)
```

```
%.o: %.c
```

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

```
clean:
```

```
-rm $(EXE) *.o
```

- The EXE variable holds the name of every target that should get executed in the Makefile to create the associated executable files (note how the names of the targets are also the same as the names of the executable files - again, this is not required but helpful. In this case, because the names are the same, then \$(EXE) can be used in the **clean** target to remove the executable files. If the names were different, then this couldn't be done).
- The **all** is a target that tells the Makefile to execute the targets listed in EXE.
- The target for every associated executable file is listed (**driver** and **unit_test**).
- A generic rule is used to create all of the object files from the source code files.
- One single **clean** target can be used to remove all the executable files and all of the associated object files. Note how the command associated with clean uses *.o instead of \$(DRIVER_OBJ) \$(UNIT_TEST_OBJ). This is because DRIVER_OBJ and UNIT_TEST_OBJ both contain the *example.o* object file. So, if they were used, it would be telling the Makefile to remove the same file twice like

```
-rm main.o example.o unit_test.o example.o
```

By using *.o, it tells the Makefile to just look for and remove all object files like

```
-rm main.o example.o unit_test.o
```

- In summary, entering *make* in the command line will execute the target **all** which will in turn execute the targets **driver** and **unit_test** to create the *driver* and *unit_test* executable files. The object files *main.o*, *unit_test.o*, and *example.o* are created, and then *main.o* and *example.o* will be used to create *driver*, and *unit_test.o* and *example.o* will be used to create *unit_test*. Lastly, entering *make clean* in the command line will execute the **clean** target which executes the command to remove the executable and object files.

<i>make</i>	create the <i>driver</i> and <i>unit_test</i> executable files
<i>./driver</i>	run the <i>driver</i> program (or <i>valgrind ./driver</i> to check for memory leaks)
<i>./unit_test</i>	run the <i>unit_test</i> program (or <i>valgrind ./unit_test</i> to check for memory leaks)
<i>make clean</i>	remove the executable files and object files

Now that you have an understanding of Makefiles, the next two pages show the basic go-to Makefiles that you can use for C and C++. Using the advanced features just discussed, they can be easily modified to add in extra source code files and executable files.

Go-To C Makefile

```
CC = gcc
CFLAGS = --std=c99 -g -Wall          # note the two dashes on --std
DRIVER_OBJ = main.o example.o
EXE = driver

all: $(EXE)

driver: $(DRIVER_OBJ)
    $(CC) $(CFLAGS) -o driver $(DRIVER_OBJ)

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

clean:
    -rm $(EXE) *.o
```

Go-To C++ Makefile

```
CXX = g++
CXXFLAGS = -std=c++11 -g -Wall       # note the one dash on -std
DRIVER_OBJ = main.o example.o
EXE = driver

all: $(EXE)

driver: $(DRIVER_OBJ)
    $(CXX) $(CXXFLAGS) -o driver $(DRIVER_OBJ)

%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $< -o $@

clean:
    -rm $(EXE) *.o
```

-

Modifying The Go-To Makefiles

If you decide to add another source code file into your program that is run with the *driver* executable file, such as *example2.c* for C or *example2.cpp* for C++, then add *example2.o* to the DRIVER_OBJ variable.

```
DRIVER_OBJ = main.o example.o example2.o
```

If you decide to create another program entirely that requires another executable file, for example *driver2*, then take the following steps (using the C Makefile as an example).

- Below the DRIVER_OBJ variable, add another variable named DRIVER2_OBJ that holds the object files needed to create the *driver2* executable file. Then, add *driver2* to the EXE variable.

```
DRIVER_OBJ = main.o example.o
DRIVER2_OBJ = # list object files needed here
EXE = driver driver2
```
- Below the **driver** target to create the *driver* executable file, add the **driver2** target to create the *driver2* executable file.

```
driver: $(DRIVER_OBJ)
    $(CC) $(CFLAGS) -o driver $(DRIVER_OBJ)
driver2: $(DRIVER2_OBJ)
    $(CC) $(CFLAGS) -o driver2 $(DRIVER2_OBJ)
```

That's it - notice how by using the more advanced features of Makefiles, a single generic Makefile can be moderately adjusted to account for all programs.

Resources

GNU Compiler Collection (GCC)

- <https://gcc.gnu.org/>

In-depth descriptions of all available compiler flags

- <https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html> (official reference)
- <https://man7.org/linux/man-pages/man1/gcc.1.html> (other good reference)

GNU Make - comprehensive overview on all Makefile features (click on “entirely on one web page” option or “entirely one web page” option).

- <https://www.gnu.org/software/make/manual/>

Other

- A simple “Makefile Guide” or “Makefile Tutorial” search on Google will bring up many great Makefile resources in the search results. They’re all more or less versions of the same thing, and you can probably learn new things from all of them.