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 - Single Source Code File	3
Basic Makefile Setup - Multiple Source Code Files and a Header File	6
Makefile Variables	9
Advanced Makefile Features	10
C vs. C++ Makefiles	16
Commonly Used Flags	17
Go-To C and C++ Makefiles	18
Modifying The Go-To Makefiles	22
Computing IV Dependencies	23
Resources	24

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. Taking source code and turning it into an executable file is called **compiling** a program. Note that header files are not source code files and do not get directly compiled themselves. Header files are the **.h** and **.hpp** files in C and C++ respectively (though **.h** can be used for C++ as well). When a header file is included in a source code file, it's like the code is being imported into the source code file.

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 - Single Source Code File

Makefile Setup

The general format of a C Makefile for a single main.c source code file 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
ingredients	all of the things the target relies on, called **dependencies**
recipe	the commands to execute the target - note that this must be indented with a 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 -g -std=c99 -Wall -o driver main.o
main.o: main.c
    gcc -g -std=c99 -Wall -c main.c -o main.o

clean:
    rm driver main.o
```

Below is a breakdown of each component shown above:

```
driver: main.o
    gcc -g -std=c99 -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 target's dependency. It 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
- -g -std=c99 -Wall are all **flags** which are optional but always helpful features.
  - -g enables debugging options for the compiler. This is most commonly used to check for memory leaks with **valgrind**.
  - -std=c99 specifies that the C99 version of C is being used. There are other versions of C as well. It is also acceptable to use two dashes instead of one like --std=c99
  - -Wall enables all warnings that are considered questionable and easy to avoid.

- -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.
  - Common point of confusion - driver is the name of the target, and driver in the -o driver is the name of the executable file. The name of a target doesn't matter but it is a convention to name the target and executable file the same. The utility of this will be seen in examples later on.
  - 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 like below

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

- main.o: main.c
  - main.o is the object file being created
  - main.c is main.o's dependency, the source code file used to create main.o
- gcc -g -std=c99 -Wall -c main.c -o main.o
  - gcc, -g, -std=c99, 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

Lastly, there is a clean target to "clean up" the built solution - delete the executable/object files.

```
clean:
    rm driver main.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.

Compilation Process

Now that the Makefile is setup, you can use the following commands:

make or *make driver*	compile the code and create the executable file
./driver	run the executable file (run the program)
valgrind ./driver	run the executable with valgrind to check for memory leaks
make clean	clean up the solution (delete the executable file/object files)

But how does the code actually get compiled when you enter *make* or *make driver*? Since an executable file is built from dependencies, the dependencies need to get built before the executable file is made. Only after all dependencies have been created can the executable file get created. In this example, the first command is:

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


This says an executable file named *driver* needs to get created from its dependency *main.o*. The Makefile looks to see if *main.o* exists yet and if it doesn't, it looks for a command to build *main.o* and finds it as the next command:

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

It executes the command to create *main.o*, and since all dependencies have now been created for the *driver* executable file, it executes the command to create *driver*. So, the order in which the commands execute is as follows:

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

So, to reiterate, the Makefile sees that an executable file needs to get created. It also sees that the executable file is built from certain dependencies (the object files). So it first creates the dependencies and then after creates the executable file.

A further discussion on executable files and their dependencies - when you enter *make* in the command line, it will create the executable file if it doesn't exist yet, or it will recreate the executable file if any of the dependencies have been updated. Consider the following scenarios:

- You enter *make* for the first time. This creates *main.o* and *driver*.
- You enter *make* again. In this case, nothing happens since all of the files are up to date.
- You enter *make clean*. This deletes *main.o* and *driver*. You enter *make* again which creates *main.o* and *driver* just like you did the first time.
- You enter *make* again, but this time before entering *make* you had modified *main.c*. In this case, *main.c* gets recompiled creating an updated version of *main.o*, and since *main.o* is a dependency of *driver*, *driver* gets recreated with the updated version of *main.o*.

The takeaway from this is that anytime you update a source code file, you don't have to enter *make clean* to delete everything and then enter *make* to rebuild everything with the changes. You can just enter *make* and the Makefile will look to see if any dependencies need to get updated with code changes. If so, it will rebuild them and then ultimately rebuild the executable file. This feature is especially helpful when working with multiple files.

Basic Makefile Setup - Multiple Source Code Files and a Header File

Makefile Setup

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 and you 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*, and *main.c* used the *example* interface which means both *main.c* and *example.c* have *example.h* included in them with an include directive. Then, the Makefile would look like the following:

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

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

Compilation Process

The compilation process is very similar to the example with a single *main.c* source code file. In this example, the first command is:

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

This says an executable file named *driver* needs to get created from its dependencies *main.o* and *example.o*. The Makefile looks to see if *main.o* and *example.o* exist yet and if they don't, it looks for a command to build *main.o* and *example.o* and finds them in the following commands.

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

It executes the command to create *main.o*, executes the command to create *example.o*, and then since all dependencies have now been created for the *driver* executable file, it executes the command to create *driver*. So, the order in which the commands execute is as follows:

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

Lastly, since *driver* now has multiple dependencies, one thing to note would be what happens if one of the source code files (*main.c* or *example.c*) gets updated but not the other? Let's say *main.c* were updated. In this case, it would recreate *main.o* but it would not recreate *example.o* since *main.c* is not a dependency of *example.o*. So, *driver* would get recreated using the updated *main.o* and the existing *example.o*. This is an example of how using object files is helpful because a change in one source code file doesn't require recompiling every single file. Without

using object files in compilation, both *main.c* and *example.c* would have to get recompiled to recreate *driver* if one of them were updated. But by using object files, only *main.c* has to get recompiled to recreate *driver*.

Header Files As Object File Dependencies

The first and simplest change to notice is how *example.o* has been added to the clean command since it needs to get deleted as well when entering *make clean*. The second change to notice is that in this example, header files are being added as dependencies for the object files. Since *main.c* and *example.c* both have *example.h* included in them with an include directive, it is optional but recommended to then include *example.h* as a dependency. The pitfall of not doing this is described below (on a side note, notice how *example.h* is not included as a dependency of *driver*, that would be a mistake).

Recall from the example on the previous page how if *main.c* were updated, this would update *main.o* which in turn updates *driver*. This all happened because *main.c* was listed as a dependency of *main.o*. In this example, if the header file were not listed as a dependency for *main.o* and *example.o* like below:

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

then *example.h* could get modified and entering *make* would not cause *driver* to get recreated if it already existed since no dependencies have been modified. This is not good because it means that a code file has been changed, and the executable file *driver* hasn't been updated to include that change. But, if *example.h* is listed as a dependency for *main.o* and *example.o*, then if *example.h* is modified it would in turn cause recompilation and *main.o* and *example.o* would get recreated with the updated code changes which in turn would cause the executable file *driver* to get recreated with the updated code changes. This is good and what you should want the Makefile to do.

If you're not going to include the header file as a dependency, then if you update the header file you would want to enter *make clean* to delete everything and *make* to rebuild everything so that the executable file is up to date with the changes in the header file. This is obviously not the preferred method since it requires you to enter both *make clean* and *make* instead of just *make* to create the updated executable file if the header file is modified.

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.

- 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 deleted, 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. You should always use variables. They can be somewhat be thought of like 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 = -g -std=c99 -Wall
OBJ = main.o example.o
EXE = driver

$(EXE): $(OBJ)
	$(CC) $(CFLAGS) -o $(EXE) $(OBJ)
main.o: main.c example.h
	$(CC) $(CFLAGS) -c main.c -o main.o
example.o: example.c example.h
	$(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 -g -std=c99 -Wall
- If anything is changed, it will be automatically adjusted in the Makefile. Let's say you decided to add in another flag like *Wextra*. Notice how $(CFLAGS) is written in 3 places. With a variable, you simply have to add it to the CFLAGS variable

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

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

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 object files
- %.c or %.cpp are placeholders for C or C++ source code files
- %.h or %.hpp are placeholders for C or C++ header files
- $@ is the name of the thing to the left of the colon (used for target or object file)
- $< is the name of the first thing to the right of the colon (used for source code file)
- $^ is the name of everything to the right of the colon (used for target dependencies)

Note that $@, $<, and $^ are called **automatic variables**.

Below is the finalized Makefile from page 6 both before and after the changes. Note that the header file being used as a dependency is gone for main.o - this is explained later (page 12)

Before changes

```
CC = gcc
CFLAGS = -g -std=c99 -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 example.h
	$(CC) $(CFLAGS) -c example.c -o example.o

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

After changes

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

$(EXE): $(OBJ)                          # target 1
	$(CC) $(CFLAGS) -o $@ $^

%.o: %.c %.h                            # generic rule 1
	$(CC) $(CFLAGS) -c $< -o $@
%.o: %.c                                # generic rule 2
	$(CC) $(CFLAGS) -c $< -o $@

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


Explanation of the changes

The target

```
$(EXE): $(OBJ)                          # target 1
    $(CC) $(CFLAGS) -o $@ $^
```

$@ refers to the thing on the left of the colon which in this case is $(EXE), the name of the target and executable file.
$^ refers to everything to the right of the colon which in this case is $(OBJ), all of the target's dependencies (object files).
So, breaking it down

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

is equivalent to

```
$(CC) $(CFLAGS) -o $(EXE) $(OBJ)
```

which is equivalent to

```
$(CC) $(CFLAGS) -o driver main.o example.o
```

The generic rules

```
%.o: %.c %.h                            # generic rule 1
    $(CC) $(CFLAGS) -c $< -o $@
%.o: %.c                                # generic rule 2
    $(CC) $(CFLAGS) -c $< -o $@
```

To build each object file, the Makefile implements pattern matching with the generic rules.

- For *generic rule 1*, the %.o: %.c %.h means for a given object file *example.o*, execute this command if *example.c* and *example.h* exist in the same file directory.
- For *generic rule 2*, the %.o: %.c means if for a given object file *example.o*, execute this command if *example.c* exists in the same file directory.
- The use of -c $< -o $@ is new here as well. The $< refers to the first thing in the dependencies aka the first thing on the right side of the colon. In this case, that is %.c. The $@ refers to the thing on the left side of the colon, in this case %.o.
- The Makefile checks the rules in order (so it looks for a match in *generic rule 1* before *generic rule 2*).
    - To create *main.o*, it looks for a pattern match with *generic rule 1* but fails to find a match. There is a *main.c* but there is no *main.h*. So, it moves onto *generic rule 2* where it finds a match between %.o: %.c and main.o: main.c So *main.o* gets created with the command associated with *generic rule 2*. Putting it all together…

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

      which is the same as

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

      which is the same as

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


- To create *example.o*, the same thing happens. It looks for a pattern match with *generic rule 1* and finds a match since there is both an *example.c* and *example.h* file. Here, %.o: %.c %.h matches to example.o: example.c example.h So, *example.o* gets created with the command associated with *generic rule 1*. Putting it all together…

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

which is the same as

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

which is the same as

```
example.o: example.c example.h
	$(CC) $(CFLAGS) -c example.c -o example.o
```

A few important things to note about the explanations above

- The explanation should illustrate why it is important that source code files and their corresponding header files have the same name (with the exception of the file extension). The pattern matching only works if this is the case. Using *example.o* as an example, the generic rule %.o: %.c %.h means create a file named *example.o* only if there is a corresponding *.c* and *.h* file also with the same name. If the files were off even by 1 character, like *example.c* and *example2.h* it wouldn't work because *example* and *example2* are not the same. So, source code files and their corresponding header files are named the same not just because it's a naming convention but because it actually has some utility in some circumstances, such as in Makefiles.
- The order of the generic rules matters - %.o: %.c %.h should be before %.o: %.c. If the order were reversed, then the %.o: %.c %.h rule would never get executed. Take *example.o* as an example. The pattern matching %.o: %.c means execute the command to create *example.o* if *example.c* exists. Since *example.c* does in fact exist, the rule will execute and the generic rule %.o: %.c %.h will never have a chance to get checked for a pattern match.

Note that this solution is not perfect. We wanted to have *example.h* be a dependency of *main.c* like in the example on page 6. However, due to using the pattern matching, if *example.h* changes only *example.o* will get recompiled and not *main.o* as well since *example.h* will not be a dependency of *main.o*. The only header file that could be a dependency of *main.o* using this basic pattern matching would be *main.h* which doesn't exist. As discussed previously in the document, this means that if *example.h* gets modified, you should rebuild the solution from scratch by entering *make clean* before *make* instead of just entering *make*.

More Advanced Features - Multiple Executable Files

The features just discussed are even more helpful if you have multiple executables that you want to create. 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 = -g -std=c99 -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 example.h
	$(CC) $(CFLAGS) -c main.c -o main.o
example.o: example.c example.h
	$(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 example.h
	$(CC) $(CFLAGS) -c unit_test.c -o unit_test.o
example.o: example.c example.h
	$(CC) $(CFLAGS) -c example.c -o example.o

clean_driver:
	-rm driver $(DRIVER_OBJ)

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

make or *make driver*	Create the *driver* executable file
make clean_driver	Delete *driver* and its associated object files
make unit_test	Create the *unit_test* executable file
make clean_unit_test	Delete *unit_test* and its associated object files

Though the previous example is a completely valid and fine Makefile, it could be rewritten much more concisely using the advanced features discussed (again, the only pitfall being *example.h* is no longer a dependency of *main.o* and *unit_test.o*). Note two new features:

*.o The * is a wildcard which says to search the current file directory (folder) for matching file names, in this case files with the *.o* file extension which are object files.

all This is a target which specifies that other targets should be made

```
CC = gcc
CFLAGS = -g -std=c99 -Wall
DRIVER = driver
DRIVER_OBJ = main.o example.o
UNIT_TEST = unit_test
UNIT_TEST_OBJ = unit_test.o example.o
EXES = $(DRIVER) $(UNIT_TEST)

all: $(EXES)

$(DRIVER): $(DRIVER_OBJ)
	$(CC) $(CFLAGS) -o $@ $^
$(UNIT_TEST): $(UNIT_TEST_OBJ)
	$(CC) $(CFLAGS) -o $@ $^

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

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

- The EXES 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 $(EXES) can be used in the clean target to remove the executable files. If the names were different, this couldn't be done).
- The all is a target that tells the Makefile to execute the targets listed in EXES.
- Each target is listed - driver and unit_test but represented in variables.
- 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 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.

Now, you can use the following commands to build, run, and clean the programs.

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

Now that you have an understanding of Makefiles, the following pages show the differences between C and C++ Makefiles, some of the most commonly used Makefile flags, and the basic go-to Makefiles that you can use for C and C++.

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**.
- 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, C17, and C2x (2023 expected).
    - Note that C89 and C90 refer to essentially the same language. C90 was the same standard as C89 but just with some formatting changes.
- 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++17**.
- 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, C++20, and experimental support for C++23.
- Use the variables CXX and CXXFLAGS.

Versions of C (view the *Resources* section for more information)

- C89 **-std=c89** or **-std=gnu89** to enable GNU extensions
- C90 **-std=c90** or **-std=gnu90** to enable GNU extensions
- C99 **-std=c99** or **-std=gnu99** to enable GNU extensions
- C11 **-std=c11** or **-std=gnu11** to enable GNU extensions
- C17 **-std=c17** or **-std=gnu17** to enable GNU extensions. The **gnu17** version if the current default mode
- C2x **-std=c2x** or **-std=gnu2x** to enable GNU extensions. This is C23 expected to come out in 2023. Support for this is currently experimental and incomplete.

Versions of C++ (view the *Resources* section for more information)

- C++98 **-std=c++98** or **-std=gnu++98** to enable GNU extensions. Default mode prior to GCC 6.1
- C++03 **-std=c++03** or **-std=gnu++03** to enable GNU extensions
- C++11 **-std=c++11** or **-std=gnu++11** to enable GNU extensions
- C++14 **-std=c++14** or **-std=gnu++14** to enable GNU extensions. Default mode from GCC 6.1 up to GCC 10 (including).
- C++17 **-std=c++17** or **-std=gnu++17** to enable GNU extensions. Default mode for GCC 11.
- C++20 **-std=c++20** or **-std=c++2a** if using GCC 9 and earlier. Use **-std=gnu++20** to enable GNU extensions. GCC's support is experimental for C++20 as of the time of this document's creation since C++20 is the most recent version.
- C++23 **-std=c++2b** or **-std=gnu++2b** to enable GNU extensions. GCC's support is experimental for C++23 as of the time of this document's creation since C++23 has not been officially released yet.

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.

Most of the flags used with gcc can be used with g++ as well.

View the *Resources* section for more information on flags.

Go-To C and C++ Makefiles

The following pages contain some “go-to” Makefiles for C and C++. They are not the end-all-be-all Makefiles but they serve the following purposes:

- Provide a generic Makefile template that is reusable for any basic C and C++ project.
- Provide a way to easily modify the Makefile to create additional executable files.
- Provides a way to easily enable/disable helpful debugging features.

Go-To C Makefile

```
CC = gcc
CFLAGS = -std=c99 -Wall -Wextra -Wpedantic
LDLIBS = # any additional dependencies go here, like -lm for <math.h>
DEBUG = #-Og -g -fsanitize=undefined # uncomment line for debugging while developing code
EXE1 = driver
EXE1_OBJ = main.o example.o
EXES = $(EXE1)

.PHONY: all clean

all: $(EXES)

$(EXE1): $(EXE1_OBJ)
	$(CC) $(CFLAGS) $(DEBUG) -o $@ $^ $(LDLIBS)

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

clean:
	-rm $(EXES) $(wildcard *.o)
```

Copy this Makefile exactly as it is since order matters for some things.
Explanation of additional features

- LDLIBS: All extra dependencies will now be listed in this variable. It is important this is at the end of the line since some dependencies, like -lm, may have to come at the end.
- DEBUG: During the code development process, if you want to debug you can uncomment this. The -Og, -g, and -fsanitize=undefined are all helpful debugging tools discussed on page 17. At a minimum, it is recommended to have -g and -fsanitize=undefined. The -Og optimizes compiling for debugging. It is less necessary.
- For every executable file number "n" you want to do the following
  - Have a EXEn to store the name of the executable file
    EXEn = executable_file_name
  - Have a EXEn_OBJ for all of the object files
    EXEn_OBJ = obj1.o obj2.o ….
  - Add EXEn to the EXES variable (after any preceding EXEns).
    EXES = …. EXEn
  - .PHONY: This lists the targets, *all* and *clean* in this case, that are not actually files in the directory the Makefile is in.

- The $(wildcard *.o) in the clean target
  - $(wildcard *.o) is used instead of just regular *.o which stops the Makefile from trying to delete nonexistent object files. Instead of

    clean:

        -rm $(PRODS) *.o

    it is now

    clean:

        -rm $(PRODS) $(wildcard *.o)

Go-To C++ Makefile

```
CXX = g++
CXXFLAGS = -std=c++17 -Wall -Wextra -Wpedantic
LDLIBS = # any additional dependencies go here
DEBUG = #-Og -g -fsanitize=undefined # uncomment line for debugging while developing code
EXE1 = driver
EXE1_OBJ = main.o example.o
EXES = $(EXE1)

.PHONY: all clean

all: $(EXES)

$(EXE1): $(EXE1_OBJ)
	$(CXX) $(CXXFLAGS) $(DEBUG) -o $@ $^ $(LDLIBS)

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

clean:
	-rm $(EXES) $(wildcard *.o)
```

The C++ go-to Makefile is the same with small changes

- CXX and g++ are used instead of CC and gcc
- CXXFLAGS and the version of C++ are used instead of CFLAGS and the version of C.
- All occurrences of *.h* have changed to *.hpp*.
- All occurrences of *.c* have changed to *.cpp*.

-

Modifying The Go-To Makefiles

Using an actual example below, you can see how easy it is to modify the go-to Makefiles so they can create a whole new executable file. The original Makefile is for a program with an executable file named *driver* which relies on *main.c*, *example.c*, and *example.h*. Then you decide to add in a unit test program which has an executable file named *unit_test* which relies on *unit_test.c*, *example.c*, and *example.h*.

```
1  CC = gcc
2  CFLAGS = --std=c99 -Wall
3  LDLIBS = -lm
4  # DEBUG = -Og-g -fsanitize=undefined
5  EXE1 = driver
6  EXE1_OBJ = main.o example.o
7  EXES = $(EXE1)
8
9  .PHONY: all clean
10
11 all: $(EXES)
12
13 $(EXE1): $(EXE1_OBJ)
14         $(CC) $(CFLAGS) $(DEBUG) -o $@ $^ $(LDLIBS)
15
16 %.o: %.c %.h
17         $(CC) $(CFLAGS) $(DEBUG) -c $< -o $@
18 %.o: %.c
19         $(CC) $(CFLAGS) $(DEBUG) -c $< -o $@
20
21 clean:
22         -rm $(EXES) $(wildcard *.o)
```

```
1  CC = gcc
2  CFLAGS = --std=c99 -Wall
3  LDLIBS = -lm
4  # DEBUG = -Og-g -fsanitize=undefined
5  EXE1 = driver
6  EXE1_OBJ = main.o example.o
7  EXE2 = unit_test
8  EXE2_OBJ = unit_test.o example.o
9  EXES = $(EXE1) $(EXE2)
10
11 .PHONY: all clean
12
13 all: $(EXES)
14
15 $(EXE1): $(EXE1_OBJ)
16         $(CC) $(CFLAGS) $(DEBUG) -o $@ $^ $(LDLIBS)
17 $(EXE2): $(EXE2_OBJ)
18         $(CC) $(CFLAGS) $(DEBUG) -o $@ $^ $(LDLIBS)
19
20 %.o: %.c %.h
21         $(CC) $(CFLAGS) $(DEBUG) -c $< -o $@
22 %.o: %.c
23         $(CC) $(CFLAGS) $(DEBUG) -c $< -o $@
24
25 clean:
26         -rm $(EXES) $(wildcard *.o)
```

Notice how few steps had to be done to give the Makefile the ability to create a whole new executable file.

- The name of the new executable file and its object file dependencies were stored in variables (lines 7 and 8)
- The name of the new executable file was added to the EXES variable (line 9)
- A new target was added to create the new executable file (lines 17 - 18).

That's it.

Computing IV Dependencies

The following are some common dependencies needed in the Computing IV class at the University of Massachusetts Lowell. The one needed for the math.h library in C is included as well since that's such a commonly used C library.

-lm	If using the C <math.h> library
-lboost_unit_test_framework	If using the C++ Boost unit testing library
-lsfml-graphics	If using the C++ SFML graphics library
-lsfml-window	If using the C++ SFML window library
-lsfml-system	If using the C++ SFML system library
-lsfml-network	If using the C++ SFML network library
-lsfml-audio	If using the C++ SFML audio library

Resources

GNU Compiler Collection (GCC)

- https://gcc.gnu.org/

In-depth descriptions of all available compiler options

- 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/

How to specify a version of C or C++

- https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html

More information on C++ versions

- https://gcc.gnu.org/projects/cxx-status.html

Note that the Linux man pages (man7.org, link above) also provide information about how to specify versions of C and C++

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.

