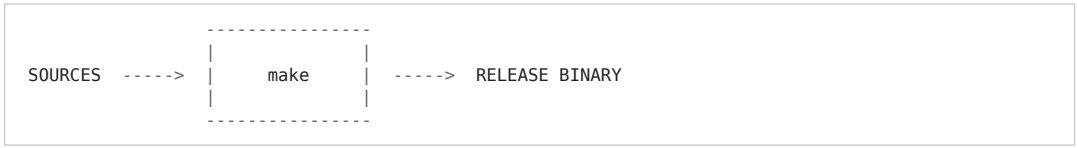
Good place for an intro

What are we doing?

We need a build system that compiles our program, is easy to use, scales as we add more files, and exists everywhere that GNU software does. We're going to use **make**.

What is make?

A build system. Takes source files and makes something releasable out of them.



Isn't that a compiler? That sounds like a compliler. I use `gcc`.

No. Make is a build system. Make will take all of the source files we have, and orchestrate running `gcc` or whatever compiler we need for our target architecture. It will handle determining what needs to be updated if just any source file changes between compilation. As developers, this feature is more for us than end users since it will speed our development with faster compile times.

We can also use make to generate things that have nothing to do with `gcc`. You might have a python script that generates a jpeg based on a text file. Make can help you there too.

That sounds pretty good. What are we doing here?

You tell make what to do by creating a **Makefile**. With some contrived examples of simple c programs, we will start with a poor Makefile and make incremental improvements to it.

I want to use a more modern build system.

That's fair, there are a lot of good options available today. But make has some advantages. First, it is ubiquitous and at hand on all Linux distros. It is also quick to implement for simple to moderately complex projects. Make has become the de facto standard for so many projects over the years that the odds are you will encounter it often. Even if it's not your favorite, being able to understand make is very valuable.

If you were hoping to use **cmake**, it actually generates these Makefiles. You should probably learn this so you know what you're dealing with under the hood.

What do I need to know?

Just be able to edit text files and use some simple shell commands. The class is focused around specifically compiling C code, so it helps to know how `gcc` compiles C programs. If you aren't a C expert, don't worry, you don't need to write any code.

C compilation review

Here's a quick review, C experts skip ahead and don't judge too harshly.

Contents

A C program exists as one or more **.c** files. Here we have **first.c** and **second.c** We will ignore header files (**.h** files) as they won't affect our make examples later. For completeness, just know that if the source file has a `#include "file.h"` in it, the **preprocessor** will just jam the contents of the header file into that source file based on conditional rules. After that, we have something like this:

```
SOURCES
-------
first.c
second.c
```

A compiler will **compile** these into assembly. You can get `gcc` to give you the assembly directly with the **-S** option.

```
$ gcc -S first.c -o first.s
$ gcc -S second.c -o second.s
```

Giving this:

```
SOURCES    ----->   ASSEMBLY
-------             --------
first.c             first.s
second.c            second.s
```

Next, the compiler will call on the assembler to **assemble** the assembly into **object files**. You can use the **-c** option to accomplish this manually.

```
$ gcc -c first.s -o first.o
$ gcc -c second.s -o second.o
```

Now it looks like this:

```
SOURCES    ----->   ASSEMBLY    ----->   OBJECTS
-------             --------             -------
first.c             first.s              first.o
second.c            second.s             second.o
```

Last, to build an execuable we can let the compiler call on the linker to **link** the object files together. This will make the final program we can run

```
$ gcc first.o second.o -o program
```

And finally it looks like this:

```
SOURCES    ----->   ASSEMBLY    ----->   OBJECTS    ----->   EXECUTABLE
-------             --------             -------             ----------
first.c             first.s              first.o    ----->   program
second.c            second.s             second.o   ----/
                                        <libraries> ---/
```

And you can run it

```
$ ./program
```

You can skip any number of these steps. Here's the shortest path that will not create the assembly or object files:

```
gcc first.c second.c -o program
```

Where does *make* fit in again?

Dependency management. You tell make what depends on what and it will generate a computer sciency thingy called a dependency graph.

If you tell make

```
first.c -> first.o
second.c -> second.o
```

and

```
first.o + second.o -> program
```

We can ask make for *program* and it will work backwards to figure out how to get from the source files all the way to the *program* file. If we have hundreds of steps one day, we will rely heavily on this.

No more questions? That's amazing!

Let's *make* up a problem

Be a programmer!

Imagine we are starting a new project in C.

Create the following C file called *hello.c*

```
#include <stdio.h>

int main() {
        printf("Hello World!\n");
        return 0;
}
```

How would we compile it? Let's use *gcc* with some compiler flags we prefer to use. We will compile, assemble and link our simple program all at once because we like fewer steps and gcc does that for us by default. We might want to do something like this in a bash shell:

```
$ gcc -Wall -Werror -Wextra hello.c -o hello
```

Now we can easily run the program

```
$ ./hello
Hello world!
```

Where do we go from here?

How do we make sure this step is repeatable by everyone with access to the project? We could place that gcc line in a bash script called *compile.sh* and be done. But what happens when we add more files? Let's say we continue to work on the project. Soon there are 60 source files. Think about the following:

- How tedious is it to add new files?
- How long will it take to recompile after changing only one source file?
- What does it look like to generate multiple executables?
- What if I want to generate different types of files with more programs than gcc?
- What do we do if we want to clean up all the compiled artifacts?

Capture it in a Makefile

Let's create a Makefile with just our exact steps. When you run the **make** program, it will look for a file called **Makefile** in the current directory. This is the configuration file that will be parsed to build your program. Create the following Makefile called *Makefile*. Note that must be a *tab* and not *spaces* before the *gcc*.

```
program:
        gcc -Wall -Werror -Wextra hello.c -o hello
```

Delete the *hello* file and let **make** compile for us

```
$ rm hello
$ make
gcc -Wall -Werror -Wextra hello.c -o hello
```

You can also try the following

```
$ make program
gcc -Wall -Werror -Wextra hello.c -o hello
```

Go ahead and run either command a few times. Notice it keeps compiling and compiling. Is this really what we want? This is no better than the bash script. Surely we want to only compile what we need when we need it.

A *target*ed approach

Fix up this Makefile

Let's take a look at what we made fix it.

```
program:
	gcc -Wall -Werror -Wextra hello.c -o hello
```

What we have created in this Makefile is our first **rule**. In our rule, we created a **target** named *program* and a **recipe** that is the gcc line. Make will create our target using the recipe we gave it. The recipe can be anything you run in a shell and can be multiple lines long. Here's the thing about targets...they are supposed to be the target filename we want make to build. Oops, we should probably call that *hello*.

Fix up the Makefile:

```
hello:
	gcc -Wall -Werror -Wextra hello.c -o hello
```

Now you can build *hello* using `make` or `make hello`. Running `make hello` tells make to build the *hello* target. By default, make will build the hello target when not given an argument since it is the first rule it found in the file. Let's delete hello and run make a couple times.

```
$ make hello
gcc -Wall -Werror -Wextra hello.c -o hello
$ make hello
make: 'hello' is up to date.
```

Neat! Make knows that *hello* has been made and did not rebuild. But it decided it's up to date?! Does it really know that? Let's edit our source file and try compile

hello.c

```
#include <stdio.h>

int main() {
	printf("Hello Dolly!\n");
	return 0;
}
```

```
$ make hello
make: 'hello' is up to date.
$ ./hello
Hello World!
```

Uh oh, looks like we need to tell make that we depend on the source file. Make the following adjustment and add a **prerequisite** to the rule.

```
hello: hello.c
	gcc -Wall -Werror -Wextra hello.c -o hello
```

Let's go back to bash and try making a couple times to see what happens.

```
$ make hello
gcc -Wall -Werror -Wextra hello.c -o hello
$ make hello
make: 'hello' is up to date.
$ ./hello
Hello Dolly!
```

Now we're in business. Make will now check the last modified time of the target *hello* and compare it to the last modified time of the prerequisite *hello.c*. If *hello.c* is newer than *hello* (or *hello* doesn't exist), the rule will run. Try using *touch* to update the modified time to see this in action:

```
$ make hello
make: 'hello' is up to date.
$ touch hello.c
$ make hello
gcc -Wall -Werror -Wextra hello.c -o hello
```

Drowning in a *C* of files

Let's make a larger program

Make a new directory. Create *shout.c shout.h let.c let.h main.c* and *Makefile*

shout.h

```
#ifndef __SHOUT_H__
#define __SHOUT_H___

void shout();

#endif
```

shout.c

```
#include <stdio.h>
#include "shout.h"

void shout() {
    printf("Shout\n");
}
```

let.h

```
#ifndef __LET_H__
#define __LET_H__

void let();

#endif
```

let.c

```
#include <stdio.h>
#include "let.h"

void let() {
    printf("Let it all out\n");
}
```

main.c

```
#include "shout.h"
#include "let.h"

int main() {
    shout();
    shout();
    let();
}
```

Makefile

```
song: main.c shout.c let.c
	gcc -Wall -Werror -Wextra main.c shout.c let.c -o song
```

Note: we could add the header files as prerequisites but we'll keep this simple

Let's try to compile and run:

```
$ make
gcc -Wall -Werror -Wextra main.c shout.c let.c -o song
$ ./song
Shout
Shout
Let it all out
```

Use gcc more to use gcc less

Nice. That works but we've got a lot to improve. A big problem is that if any of the prerequisite source files are modified, we will recompile everything. It looks like trying to do the one simple gcc call will hold us back if this program grows considerably. Let's compile our source files to *object* files and then link them all at the end. That way only the files that have changed will cause a recompilation in the associated binary.

Make more targets! Make the *Makefile* look like this

```
shout.o: shout.c
	gcc -Wall -Werror -Wextra -c shout.c -o shout.o

let.o: let.c
	gcc -Wall -Werror -Wextra -c let.c -o let.o

main.o: main.c
	gcc -Wall -Werror -Wextra -c main.c -o main.o

song: main.o shout.o let.o
	gcc -Wall -Werror -Wextra main.o shout.o let.o -o song
```

Ok, let's make sure it works.

Try making main.o

```
$ make main.o
gcc -Wall -Werror -Wextra -c main.c -o main.o
$ ls main.o
main.o
```

Try let.o

```
$ make let.o
gcc -Wall -Werror -Wextra -c let.c -o let.o
$ ls let.o
let.o
```

Make the complete *song* target

```
$ make song
gcc -Wall -Werror -Wextra -c shout.c -o shout.o
gcc -Wall -Werror -Wextra main.o shout.o let.o -o song
```

Notice that since main.o and let.o were already built, they were not built for song. The missing shout.o did need to be built.

Let's try making before and after updating *let.c*

```
$ make song
make: 'song' is up to date.
$ touch let.c
$ make song
gcc -Wall -Werror -Wextra -c let.c -o let.o
gcc -Wall -Werror -Wextra main.o shout.o let.o -o song
```

Stop and think about why this is so much better than what we started with.

Bonus text targets

Gcc is not the only thing you can use with make. Try adding the following rules to the bottom of *Makefile* and play with making the targets. After making, delete some of the files make has generated and see how the dependencies are resolved and rebuilt.

```
line.txt:
	echo "These are the things I can do without" > line.txt

chorus.txt: song line.txt
	./song > chorus.txt
	cat line.txt >> chorus.txt
```

Go ahead and delete the rules when you're done.

Be a PHONY!

Create helpful targets

You may have noticed on our project, if you run just `make` we don't get our *song* file. The first rule is for the *shout.o* target so that is what get's built. Some projects include an **all** target that builds everything. In this case all we need to build is *song*, so we can add the following rule to the top of the makefile with a *song* prerequisite:

```
all: song
```

Now we can run `make`, `make all`, or `make song` to build our *song* executable.

But we've violated an assumption that make will make about the *all* target. Remember that the target is supposed to refer to an actual filename. We may not have an issue right now, but whether or not a file named *all* somehow gets created in this directory, may affect whether or not the rule is run. Since our intent is not to create an *all* file, we should tell make.

We will add a special target to our Makefile called **.PHONY**. Make will not check for a file for the prerequisites of this special target. After adding our *.PHONY* target, here is our updated Makefile

```
.PHONY: all

all: song

shout.o: shout.c
	gcc -Wall -Werror -Wextra -c shout.c -o shout.o

let.o: let.c
	gcc -Wall -Werror -Wextra -c let.c -o let.o

main.o: main.c
	gcc -Wall -Werror -Wextra -c main.c -o main.o

song: main.o shout.o let.o
	gcc -Wall -Werror -Wextra main.o shout.o let.o -o song
```

Cleanup, cleanup

We have a lot of non-source files here that it would be handy to clean up. Let's create a common target called **clean** to delete them.

Modify the top of the Makefile

```
.PHONY: all clean

all: song

clean:
	rm -f *.o
	rm -f song
```

And let's try it

```
$ make
make: Nothing to be done for 'all'.
$ make clean
rm -f *.o
rm -f song
$ make
gcc -Wall -Werror -Wextra -c main.c -o main.o
gcc -Wall -Werror -Wextra -c shout.c -o shout.o
gcc -Wall -Werror -Wextra -c let.c -o let.o
gcc -Wall -Werror -Wextra main.o shout.o let.o -o song
```

Testing 1 2 3...

Another common target is **test**. If we have some way to test our program, we can put the commands in this rule. If any recipe that runs during the make execution returns a non-zero exit code, make will close with an error. If you add your test and make exits cleanly, the tests have passed.

Modify the top of the Makefile again

```
.PHONY: all test clean

all: song

test: song
	./song
```

The *test* target will rely on *song* being complete, and it will execute it for us. Try some make targets to see what make does based on the state of the directory.

```
$ make test
./song
Shout
Shout
Let it all out
$ make clean
rm -f *.o
rm -f song
$ make test
gcc -Wall -Werror -Wextra -c main.c -o main.o
gcc -Wall -Werror -Wextra -c shout.c -o shout.o
gcc -Wall -Werror -Wextra -c let.c -o let.o
gcc -Wall -Werror -Wextra main.o shout.o let.o -o song
./song
Shout
Shout
Let it all out
```

You may not want to echo the command in one of your make recipes. In the *test* target we don't need to see `./song`. Try adding @ with this modification:

```
test: song
	@./song
```

```
$ make test
Shout
Shout
Let it all out
```

Think about installation

It is also very common to have a target called **install** to install the program. Think about how you might implement this. Projects like python also allow you to run `make altinstall` so you don't wipe whatever system installation is there already. Think about some other targets that would help the developer or the user.

Stop repeating yourself, stop repeating yourself

Use make syntax to scale better

Look again at our current makefile

```
.PHONY: all test clean

all: song

test: song
	@./song

clean:
	rm -f *.o
	rm -f song

shout.o: shout.c
	gcc -Wall -Werror -Wextra -c shout.c -o shout.o

let.o: let.c
	gcc -Wall -Werror -Wextra -c let.c -o let.o

main.o: main.c
	gcc -Wall -Werror -Wextra -c main.c -o main.o

song: main.o shout.o let.o
	gcc -Wall -Werror -Wextra main.o shout.o let.o -o song
```

You probably noticed that we have violated the software principle of DRY (don't repeat yourself) quite a bit. If we were done with this project, it may not matter. However, if we are going to keep adding to this project there will be a lot of copy and paste and hoping we don't make mistakes.

Let's first set our sights to the objects. We have three lines that are almost identical. How can we make a general rule that covers *shout.o*, *let.o* and *main.o*? Replace those lines with this new rule:

```
%.o: %.c
	gcc -Wall -Werror -Wextra -c $< -o $@
```

Give *make* a try and make sure we can still build everything.

We've got a few new ideas here. You may have guessed what is happening with the target **%.o** and its prerequisite **%.c**. You can attempt to make any target ending in *.o* and make will automatically look for the same filename ending in *.c* as the prerequisite. Down in the recipe, we have used our first make variables. **$@** is an **automatic variable** that gets the value of the target. The other **automatic variable** we used is **$<** which refers to the first prerequisite. We only have one prerequisite in this rule, so in this case it is equivalent to another automatic variable **$^**, which refers to all of the prerequisites.

Let's update the recipe for *song* with that *$^* variable:

```
song: main.o shout.o let.o
	gcc -Wall -Werror -Wextra $^ -o $@
```

See what happens if you try to compile a source file that doesn't exist.

```
$ make bogus.o
make: *** No rule to make target 'bogus.o'.  Stop.
```

Since *bogus.c* does not exist, *bonus.o* cannot be built.

Add some variation with variables

We still have two different lines with calls to gcc on them. We have consistent flags across them and as we choose to update them, it would be nice for them to be kept in one place. Let's create a variable for our flags called **CFLAGS**.

One way we could do this is by adding

```
CFLAGS = -Wall -Werror -Wextra
```

When a variable is created with the **=** operator, this is called **deferred** because if the contents themselves contain variables, the expansion will not happen during make's first phase of operation, it will be *deferred*. This can lead to many levels of recursion and may cause an infinite loop, although make will detect this for you and halt.

Instead, we will use the **:=** operator to force **immediate** expansion.

```
CFLAGS := -Wall -Werror -Wextra
```

When we want to refer to our flags, we use the variable with `$(CFLAGS)`

Another common variable we might use is to set the compiler to the **CC** variable.

Let's update our Makefile to look like this:

```
.PHONY: all test clean

CFLAGS := -Wall -Werror -Wextra
CC := gcc

all: song

test: song
	@./song

clean:
	rm -f *.o
	rm -f song

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

song: main.o shout.o let.o
	$(CC) $(CFLAGS) $^ -o $@
```

Did we gain anything with the *CC* variable? Think about if we want to cross-compile. It might be handy to change the compiler for different targets.

Group the source and object files

We can clean up more still by refactoring the object files and the *song* binary into variables.

```
.PHONY: all test clean

OBJS := main.o shout.o let.o
EXE := song
CFLAGS := -Wall -Werror -Wextra
CC := gcc

all: $(EXE)

test: $(EXE)
	@./$(EXE)

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

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

$(EXE): $(OBJS)
	$(CC) $(CFLAGS) $^ -o $@
```

Now if we want to change the name of the executable we build or add new objects, we can simply adjust the variables at the top of the file.

But the Makefile somehow loses the intent of what we think of as developers. We are editing source files, not object files. Can we build our list of object files based on a list of source files? Yes.

Let's add this variable declaration above *OBJS*:

```
SRCS := main.c shout.c let.c
```

Now we can use a neat make **function** to generate the *OBJS*. We will use **patsubst** to perform a path substitution. It will look a lot like our improved *%.o* target. Replace our *OBJS* declaration:

```
OBJS := $(patsubst %.c, %.o, $(SRCS))
```

The substitution speaks for itself after looking at the *%.o* target rule. Try using *make* to make sure everything still works the same.

At this point, depending on the project, you may want to stop here and manually control your list of C files. You may have multiple variables that have specific source files to be organized seperately to be linked into different executables. In our project, we just want to grab all the source files, so we do not need to explicitly list them all. We can use the **wildcard** function to do this.

Replace the *SRCS* declaration:

```
SRCS := $(wildcard *.c)
```

This will match all files in the current directory ending in *.c*

Take a second to look at our current Makefile and make sure you understand all the pieces. Also play with all of the targets we have been using to gain confidence that the functionality is identical. What we have gained is we can add new source files and compile our project with no modification to the Makefile.

```
.PHONY: all test clean

SRCS := $(wildcard *.c)
OBJS := $(patsubst %.c, %.o, $(SRCS))
EXE := song
CFLAGS := -Wall -Werror -Wextra
CC := gcc

all: $(EXE)

test: $(EXE)
	@./$(EXE)

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

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

$(EXE): $(OBJS)
	$(CC) $(CFLAGS) $^ -o $@
```

Organization? We don't need no stinkin' organization.

Put everything in its proper place

Some projects like to organize all source files into a directory, object files into another, and release binaries into a third. You may like some variation on this. For this section, we will move all our source files and header files into a directory called **src**. We will then create new empty directories **obj** for the object files and **bin** for the release binary. This sort of organization makes it easy to make clean rules and give version control software an easy place to ignore. Why could this be better? We might create other temporary files that are not **.o* that we also do not want to save in version control. It is also clear to the user where the release binary is.

Change our directory structure and move files to match the following:

```
.
├── bin (directory)
├── Makefile
├── obj (directory)
└── src (directory)
    ├── let.c
    ├── let.h
    ├── main.c
    ├── shout.c
    └── shout.h
```

Now, in the Makefile, add these variables above *SRCS*:

```
SRC_DIR := src
OBJ_DIR := obj
BIN_DIR := bin
```

This makes it easy to change directory names and locations later if we want. But now we need to change our next three variable declarations:

```
SRCS := $(wildcard $(SRC_DIR)/*.c)
OBJS := $(patsubst $(SRC_DIR)/%.c, $(OBJ_DIR)/%.o, $(SRCS))
EXE := $(BIN_DIR)/song
```

Now the *wildcard* function can find our source files. Also the *OBJS* variable will get the path to where we want our objects to go. Lastly, the *EXE* will be placed in *bin*.

Unfortunately, we just broke our *%.o* rule. Let's fix it:

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

We can also fix up our clean recipe:

```
clean:
	rm -f $(OBJ_DIR)/*
	rm -f $(EXE)
```

Try using all of our make targets to make sure everything does what we expect. Notice if you want to manually build an object file like *main.o* you must now run `make obj/main.o` since that is the new target. Similarly, `make bin/song` is needed to build the actual binary. However, all of our *PHONY* rules work the same way. These provide that syntactic sugar for our users so they do not need to worry about the ins-and-outs of all the rules. Simply communicate to them how to use the *PHONY* rules, and how to find the binary in *bin/*.

Discussion: You may be bothered by the inconsistency between the appearance of these two rules:

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

$(EXE): $(OBJS)
	$(CC) $(CFLAGS) $^ -o $@
```

Should we put the *bin* path inside the *EXE* variable or not? If we don't, we have to place $(BIN_DIR)/$(EXE) at every location we currently have $(EXE). Which is better? You decide.

Use **order-only** to make those dirs automagically

Wouldn't it be better to only save the *src* directory in version control? Absolutely. But what happens if modify our *clean* rule and try to build? Make this change and try it out:

```
clean:
	rm -rf $(OBJ_DIR)
	rm -rf $(BIN_DIR)
```

```
$ make clean
rm -rf obj
rm -rf bin
$ ls
Makefile  src
$ make
gcc -Wall -Werror -Wextra -c src/main.c -o obj/main.o
Assembler messages:
Fatal error: can't create obj/main.o: No such file or directory
Makefile:22: recipe for target 'obj/main.o' failed
make: *** [obj/main.o] Error 1
```

What is the `No such file or directory`? The *obj* directory. We might also predict a similar result when *bin/song* tries to build. We can fix this by creating a new rule for our *OBJ_DIR* and *BIN_DIR*.

```
$(OBJ_DIR) $(BIN_DIR):
	@mkdir -p $@
```

Now we can run `make obj` to create an *obj* directory and `make bin` to make a *bin* directory. Now that we have a rule with multiple targets, that *$@* variable is pretty handly.

How do we get the directory to appear for our objects? Can't we just use our prerequisites like we know? Adding `$(OBJS): $(OBJ_DIR)` should work.

As a recap, this is what we have right now:

```
.PHONY: all clean test

SRC_DIR := src
OBJ_DIR := obj
BIN_DIR := bin
SRCS := $(wildcard $(SRC_DIR)/*.c)
OBJS := $(patsubst $(SRC_DIR)/%.c, $(OBJ_DIR)/%.o, $(SRCS))
EXE := $(BIN_DIR)/song
CFLAGS := -Wall -Werror -Wextra
CC := gcc

all: $(EXE)

test: $(EXE)
	@./$(EXE)

clean:
	rm -rf $(OBJ_DIR)
	rm -rf $(BIN_DIR)

$(OBJ_DIR) $(BIN_DIR):
	@mkdir -p $@

$(OBJS): $(OBJ_DIR)

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

$(EXE): $(OBJS)
	$(CC) $(CFLAGS) $^ -o $@
```

Let's see if we get what we expect:

```
$ make clean
rm -rf obj
rm -rf bin
$ make obj/let.o
gcc -Wall -Werror -Wextra -c src/let.c -o obj/let.o
$ make obj/let.o
make: 'obj/let.o' is up to date.
$ make obj/shout.o
gcc -Wall -Werror -Wextra -c src/shout.c -o obj/shout.o
```

So far so good

```
$ make obj/let.o
gcc -Wall -Werror -Wextra -c src/let.c -o obj/let.o
```

Drat. We built *let.o* again! What happened? Think about linux directory modification times. When you modify a file in a directory, that directory modification time is updated. Since *obj/let.o* has a prerequisite of the *obj* directory, when the directory is updated, make will think *obj/let.o* is out of date. We need make to know the order we want rules to be invoked *without* forcing the target to be updated if a rule runs.

Good new for us, make has a special prerequisite called **order-only** that does exactly what we want. We will add the **|** operator to tell make this is what we want. change this

```
$(OBJS): $(OBJ_DIR)
```

to this

```
$(OBJS): | $(OBJ_DIR)
```

Now do a test:

```
$ make clean
rm -rf obj
rm -rf bin
$ make obj/let.o
gcc -Wall -Werror -Wextra -c src/let.c -o obj/let.o
$ make obj/shout.o
gcc -Wall -Werror -Wextra -c src/shout.c -o obj/shout.o
$ make obj/let.o
make: 'obj/let.o' is up to date.
```

Perfect. Now we can add another order-only-prerequisite for the *bin* directory. change this

```
$(EXE): $(OBJS)
        $(CC) $(CFLAGS) $^ -o $@
```

to this

```
$(EXE): $(OBJS) | $(BIN_DIR)
        $(CC) $(CFLAGS) $^ -o $@
```

We get another bonus out of this order-only business. It turns out the *$^* variable only refers to the *normal* prerequisites and not the order-only ones, so we do not have to worry about accidently adding the *bin* directory to the gcc command.

As we near the end of our journey through Makefiles, marvel at what we have now:

```
.PHONY: all clean test

SRC_DIR := src
OBJ_DIR := obj
BIN_DIR := bin
SRCS := $(wildcard $(SRC_DIR)/*.c)
OBJS := $(patsubst $(SRC_DIR)/%.c, $(OBJ_DIR)/%.o, $(SRCS))
EXE := $(BIN_DIR)/song
CFLAGS := -Wall -Werror -Wextra
CC := gcc

all: $(EXE)

test: $(EXE)
        @./$(EXE)

clean:
        rm -rf $(OBJ_DIR)
        rm -rf $(BIN_DIR)

$(OBJ_DIR) $(BIN_DIR):
        @mkdir -p $@

$(OBJS): | $(OBJ_DIR)

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

$(EXE): $(OBJS) | $(BIN_DIR)
        $(CC) $(CFLAGS) $^ -o $@
```

After `make clean` you should have this:

```
.
├── Makefile
└── src
    ├── let.c
    ├── let.h
    ├── main.c
    ├── shout.c
    └── shout.h
```

and after `make all`:

```
.
├── bin
│   └── song
├── Makefile
├── obj
│   ├── let.o
│   ├── main.o
│   └── shout.o
└── src
    ├── let.c
    ├── let.h
    ├── main.c
    ├── shout.c
    └── shout.h
```

Make yourself great

Some more things

We have finished with our contrived example. It is not the greatest Makefile ever made. That task is left to you.

Before we part ways, here are a few more simple ideas that may be useful.

Concurrency is nice

Make does a great job of building a dependency graph between all of your targets. So, shouldn't make have some insight into what tasks can be run at the same time? If you have multiple processors, you might hope so. The answer is yes we can!

Find a project that uses a Makefile and try the **-j** option:

```
$ make --help | grep -e "-j "
  -j [N], --jobs[=N]          Allow N jobs at once; infinite jobs with no arg.
```

If you have created your dependencies correctly, you build all of your object files concurrently! The more threads you can run, the faster you will build. The time we spent splitting up our compilation process bears fruit once again.

Example:

```
$ make -j4
```

Split long lines for readability

In our example, our lines were pretty short but we might want longer ones. Make's parser will decide your line is over at the newline. If you want to split it out, you can use \ before the newline. Make will turn `\<newline>` into " " and replace all whitespace with a single " "

Example:

```
SRCS := one.c two.c three.c \
        four.c
```

becomes

```
SRCS := one.c two.c three.c four.c
```

What if we want to split on a long filename without adding the space since

```
SRC := supercalifragilistic\
        expialidocious.c
```

becomes

```
SRC := supercalifragilistic expialidocious.c
```

?

There's a trick we can use here. Look at what this does:

```
SRC := supercalifragilistic$\
       expialidocious.c
```

becomes

```
SRC := supercalifragilisticexpialidocious.c
```

Why? **$\<newline>** becomes the variable `$ `, the variable with a name of a space " ". That does not exist, therefore it expands to nothing.

Do some loops in the shell

When else do we want to split lines? In the recipe, each line is independantly sent to the shell (bash). If you want **if** or **for** statements, you might choose to use bash. We need to actually do this on one line so it works properly. How about:

```
LIST: a b c d
all:
        for i in $(LIST); do\
            echo $i; \
        done
```

But wait, there's a problem here. What is the difference between $(LIST) and $i? How does make know what is a shell variable and what is a make variable? We need to add another **$** to send the **$i** through. This is correct:

```
LIST: a b c d
all:
        for i in $(LIST); do\
            echo $$i; \
        done
```

it becomes:

```
for i in a b c d; do echo $i; done
```

You may also want to try make's **foreach** function https://www.gnu.org/software/make/manual/make.html#Foreach-Function

I'm smelling what you're stepping in

Make can do all sorts of things, and sometimes magic. Although you do not need to build c projects with it, if you do then make may be able to guess what you want.

In an empty directory, create a new file *hello.c*

```
#include <stdio.h>

int main() {
        printf("Hello World!\n");
        return 0;
}
```

Try a very minimal *Makefile*

```
hello.o: hello.c
```

and make it

```
$ make
cc    -c -o hello.o hello.c
```

Whoa. Make guessed from the filenames that you wanted to build an object file *hello.o* from *hello.c*. Remember that *CC* variable? By default it is equal to *cc* and *CFLAGS* is empty, but still used. Delete the *hello.o* file and try this:

```
hello: hello.c
```

```
$ rm hello.o
$ make
cc     hello.c   -o hello
```

Okay, do we even need the prerequisite? Edit the Makefile

```
hello:
```

```
$ rm hello
$ make
cc     hello.c   -o hello
$ make
make: 'hello' is up to date.
```

Make inferred what the source file was. We don't even need the Makefile if we give the target.

```
$ rm hello Makefile
$ ls
hello.c
$ make hello
cc     hello.c   -o hello
```

Should we use these implied rules? Up to you. Explicit tends to be better than implicit for readability in the future, but to each their own. If we started this way, would you know how make works?

Make yourself even better

When I think about recursion, it makes me think about recursion which...

You can have multiple Makefiles in different directories all for the same project. This can add some modularity and scale, but also a lot of complexity. If you are so daring, you can include one or more Makefiles into your Makefile before the dependency graph is generated. Read more about it here https://www.gnu.org/software/make/manual/make.html#Include

Do some more magic for me, I hate typing directory names

In our example, we specified where the source, object, and release directories are in each rule. In some projects, you may want make to find sources regardless of the path. There is a feature called **vpath** that gives make directories to search for files in. Give it a read at https://www.gnu.org/software/make/manual/make.html#Directory-Search

Find my prerequisites too!

Make has a feature for generating prerequisites automatically. Think about those header files. We did not add the header files in our example and we probably want to update our object files if a header changes. It would be nice if we did not have to manually put in our header files since it doesn't fit with what we have built. It turns out `gcc` has a special tool we can use.

Try this with our project:

```
$ gcc -M src/main.c
main.o: src/main.c /usr/include/stdc-predef.h src/shout.h src/let.h
```

That looks like a make target with prerequisites! There is an example of how to use this in the make manual at https://www.gnu.org/software/make/manual/make.html#Automatic-Prerequisites

More and more

We have barely scratched the surface on Makefiles. There are more functions, special targets, ways to use variables, and more obscure syntax. If you need anything more complicated or want to do more research, the make manual has got you covered.

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

I hope you learned something! Go *make* something awesome.

By The Makers