



Multi-file Development: Writing Makefiles

Shinwoo Kim



Teaching Assistant

shinwookim@pitt.edu

<https://www.pitt.edu/~shk148/>

News

- ▶ Project 4
 - New deadline: Tuesday, April 11th 2023 23:59 ET
- ▶ Final Exam
 - See schedule on PeopleSoft
- ▶ Exam II
 - Grading In progress ...

My Exam Schedule > Spring Term 2022-2023 > University of Pittsburgh					Personalize  
Class	Class Title	Exam Date	Exam Time	Exam Room	Enrolled
CS 0449-1200 (23984)	INTRO TO SYSTEMS SOFTWARE (Lecture)	4/26/2023, Wednesday	8:00AM - 9:50AM	405 Information Sciences Build	50
CS 0449-1300 (29544)	INTRO TO SYSTEMS SOFTWARE (Lecture)	4/26/2023, Wednesday	4:00PM - 5:50PM	5201 Wesley W Posvar Hall	50

Compilation using `gcc`

- ▶ In this course, you've been using the GNU C Compiler (GCC) to compile your C source code into machine code
 - `gcc -Wall hello.c -o hello`
- ▶ This is fine, but what if you have to compile multiple files?
 - `gcc plugin1.c -o plugin1.so -shared`
 - `gcc plugin2.c -o plugin2.so -shared`
 - ...
 - `gcc plugin_manager.c -o plugin_manager.so`
- ▶ But what about even larger projects?
 - The Linux operating system requires you to compile thousands of source files
 - Might take a couple of hours to compile some files... *Will you just sit there and wait until it finishes?*
- ▶ Can we automate compilation?

MAKEFILE 101

Don't read this: <https://www.gnu.org/software/make/manual/make.html>

Unless you really want to :)

Make!

- What is make?
 - It's a program that describes the relationship amongst files in your program!
 - <https://www.gnu.org/software/make/manual/make.html#Overview>
 - If done properly, it can detect which files on your software project were changed. And it can recompile [only!] those files and any other pieces of code that depend on them.
 - Saves time in LARGE projects!
- How does it work?
 - The relationships are described in a file named “Makefile” [by default]
 - You can name it differently, but it's not current practice!
 - <https://www.gnu.org/software/make/manual/make.html#Makefile-Names>
 - Make will look into that file, and follow the rules described therein

Compiling *efficiently* and *effectively*

- ▶ Makefiles allow us to automate and optimize the compiling process
- ▶ Allows us to automatically recompile (or rebuild) only the files that have been modified
 - Compares the last-modified timestamp to the last-compiled timestamp
- ▶ Allows us to compile multiple files quickly and simply with a single command (`make`)
- ▶ Allows us to create custom settings that can be used to build with different options
 - `make x86`
 - `make arm`
 - `make linux`
 - `make force`

You've used Makefiles before...Queue Lab! Malloc Project!

Rules

- Rules specify how to make files
 - How to make a file is specified by a *recipe*
 - Made files are called *targets*
 - Targets have *prerequisite*
 - Dependencies can be made by another rule :s

target: prerequisites
recipe

```
prog: main.o
    gcc -o prog main.o
main.o: main.c
    gcc -c main.c
```

Target main depends on `main.o` (that is created by another rule) and it's made by invoking `gcc -o prog main.o`

Because the Makefile has a rule to generate main.o...
But you still need main.c

Exercising

- Write a Makefile with only the prog rule and try to “make prog”

```
prog: main.o
```



```
gcc -o prog main.o
```

Output:

```
cc -c -o main.o main.c
```

```
gcc -o prog main.o
```



Make sure you use TABS, not SPACES!

What???

It looks like make knows how to generate object files... It has some *implicit rules* But we have no control this way!

Getting some control: Rules with patterns

- We can create implicit rules, for example, how to generate an object file from a C file:

These rules need to have a single %.
The % function is simple! (I promise)
This rule matches any file that ends in “.c” and will create a file with the same name with extension “.o”
For example main.c ⇒ main.o

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

```
gcc -c $< -o $@
```



What about these?

What the???

Well...

- Since we don't know the name of the file being generated or its dependencies, we need to use the make's magical *Automatic Variables*:

```
%.o: %.c  
    gcc -c $< -o $@
```

\$< ⇒ The name of the first prerequisite: whatever.c
\$@ ⇒ The name of the target: whatever.o

- So, when I need main.o, we execute:
 - **gcc -c main.c -o main.o**
- Check this if you find an unknown variable being used:
 - <https://www.gnu.org/software/make/manual/make.html#Automatic-Variables>

Checkpoint: Basic C Makefile

- Give it a try, edit the Makefile:

```
prog: main.o
    gcc -o prog main.o
%.o: %.c
    gcc -c $< -o $@
```

```
$ make
```

```
make: 'prog' is up to date.
```

- It seems there is nothing to do?
 - Well... if you have building up to this point, you already have prog made...
 - Add a rule to clean up the files you generated:

```
clean:
```

```
    rm -f prog main.o
```

Checkpoint: Basic C Makefile

- Give it a try:

```
prog: main.o
    gcc -o prog main.o
%.o: %.c
    gcc -c $< -o $@
clean:
    rm -f prog main.o
```

\$ make

make: 'prog' is up to date.

\$ make clean

rm -f prog main.o

\$ make prog

gcc -c main.c -o main.o

gcc -o prog main.o

Variables

- Say I want to compile my code with some compilation flags
`gcc -Wall -g -O2 -c main.c -o prog`

- I could write a Makefile like this:

```
prog: main.o
    gcc -o prog main.o
%.o: %.c
    gcc -Wall -g -O2 -c $< -o $@
```

- But to make is more easily modifiable I could write it like this:

```
CFLAGS := -Wall -g -O2
prog: main.o
    gcc -o prog main.o
%.o: %.c
    gcc $(CFLAGS) -c $< -o $@
```

- Give it a try, and modify your Makefile
 - Then change the optimization flag from -O2 to -O0

Live Demo

Wow that's some big font

Your Turn

Recitation Website: Practice Lab

<http://pitt.edu/~shk148/teaching/CS0449-2234/Makefile-lab.html>

No submission...but you will need to write a Makefile for project 4 (and submit it!)

Reference Slides
Courtesy of TA Jake
Kasper

Sample C Programs

Main.c

```
int main(int argc, char** argv){
    int num = 1;
    num = add_five(num);
    print_num(num);
}
```

file3.c

```
#include <stdio.h>
void print_num(int num){
    printf("Your number is: %d\n", num);
}
```

file2.c

```
int add_five(int num){
    return num + 5;
}
```

The Makefile

The Makefile

The first thing we want to do is specify the 'all' rule, which is executed when someone types 'make' into the shell

Think of it as your `int main()` for a makefile

Also note that the 'all' rule has a prerequisite, which is the executable file called 'exe'

```
all: exe
```

The Makefile

Next, we need to tell the makefile how to construct the exe executable

Thus, we'll add in a rule telling the makefile how we should do this.

```
all: exe  
exe:
```

The Makefile

Next, we need to tell the makefile how to construct the exe executable

Thus, we'll add a rule in telling the makefile how we should do this.

What prerequisites does our executable need to be able to function properly?

```
all: exe
```

```
exe: ?
```

The Makefile

The executable is going to depend on all of the object files we need to link together

Specifically, this means `exe` is going to depend on `main.o`, `file2.o`, and `file3.o`

We'll tell the makefile that these files are the prerequisites for `exe`

```
all: exe
exe: main.o file2.o file3.o
```

The Makefile

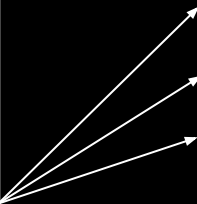
Next, we need to tell the Makefile how to generate each of these 3 object files

We can add rules for main.o, file2.o, and file3.o

Once again, we need to consider the dependencies for each of these files...

What do they depend on?

```
all: exe
exe: main.o file2.o file3.o
main.o:
file2.o:
file3.o:
```



The Makefile

Well, in order to build a main.o, we actually need a main.c file...

However, the main.c file already exists in the directory (we're modifying it), so we don't need to make rules for how to generate our .c files

So, for each object file, they will have their c counterpart as a prerequisite

```
all: exe
exe: main.o file2.o file3.o
main.o: main.c
file2.o: file2.c
file3.o: file3.c
```


The Makefile

We have all of our rules... but we haven't actually told the makefile how to make each of these files...

Let's start with the .o files. In order to turn the c file into a .o file, we need to run the command:

```
gcc -c <fname>.c -o <fname>.o
```

This goes for all .c files and .o files we're trying to generate.

```
all: exe
exe: main.o file2.o file3.o
main.o: main.c
file2.o: file2.c
file3.o: file3.c
```

The Makefile

```
gcc -c <fname>.c -o <fname>.o
```

Now that we have our compile command, let's tell the Makefile to actually run that command for each corresponding .c and .o file

```
all: exe
exe: main.o file2.o file3.o
main.o: main.c
file2.o: file2.c
file3.o: file3.c
```

The Makefile

```
gcc -c <fname>.c -o <fname>.o
```

Now that we have our compile command, let's tell the Makefile to actually run that command for each corresponding .c and .o file

This is what the makefile should look like now that we've filled in the rules for generating our object file.

```
all: exe
exe: main.o file2.o file3.o
main.o: main.c
    gcc -c main.c -o main.o
file2.o: file2.c
    gcc -c file2.c -o file2.o
file3.o: file3.c
    gcc -c file3.c -o file3.o
```

The Makefile

Lastly, we need to tell the Makefile how to generate the executable once all of the object files have been generated.

To do this, we need to link all of the objects together into the executable using the following gcc command:

```
gcc -o exe main.o file2.o file3.o
```

Let's add this to the exe rule

```
all: exe
exe: main.o file2.o file3.o
main.o: main.c
    gcc -c main.c -o main.o
file2.o: file2.c
    gcc -c file2.c -o file2.o
file3.o: file3.c
    gcc -c file3.c -o file3.o
```

The Makefile

Now, our Makefile is complete! We can run our makefile in the shell using the `make` command.

Let's try it out:

```
all: exe
exe: main.o file2.o file3.o
    gcc -o exe main.o file2.o file3.o
main.o: main.c
    gcc -c main.c -o main.o
file2.o: file2.c
    gcc -c file2.c -o file2.o
file3.o: file3.c
    gcc -c file3.c -o file3.o
```

The Makefile - Compilation

```
jbk52@thoth:~/cs449/recitations/recitation10/ex1$ make
gcc -c main.c -o main.o
main.c: In function 'main':
main.c:5:11: warning: implicit declaration of function 'add_five' [-Wimplicit-function-declaration]
   5 |         num = add_five(num);
      |                ^~~~~~
main.c:6:5: warning: implicit declaration of function 'print_num' [-Wimplicit-function-declaration]
   6 |         print_num(num);
      |         ^~~~~~
gcc -c file2.c -o file2.o
gcc -c file3.c -o file3.o
gcc -o exe main.o file2.o file3.o
```

It compiled! But we have some warnings...

Since the main.c file doesn't have direct access to these functions, we need to be a little more explicit as to not anger the compiler/linker. Recall one of the special keywords we used when we want to use something declared in another file...

Fixing the C File

Main.c

```
int main(int argc, char** argv){  
    int num = 1;  
    num = add_five(num);  
    print_num(num);  
}
```

This was our original source file... but we know there were some warnings about not knowing what `add_five()` and `print_num()` were

Let's slap in a declaration with our magic keyword

Fixing the C File

New Main.c

```
extern int add_five(int);
extern void print_num(int);

int main(int argc, char** argv){
    int num = 1;
    num = add_five(num);
    print_num(num);
}
```


The Makefile - Compilation

```
jbk52@thoth:~/cs449/recitations/recitation10/ex1$ make
gcc -c main.c -o main.o
gcc -c file2.c -o file2.o
gcc -c file3.c -o file3.o
gcc -o exe main.o file2.o file3.o
jbk52@thoth:~/cs449/recitations/recitation10/ex1$ █
```

Our warnings went away! And better yet, our executable has been generated, and we can now run our program

Running the Make Generated Executable

```
jbk52@thoth:~/cs449/recitations/recitation10/ex1$ ./exe
Your number is: 6
jbk52@thoth:~/cs449/recitations/recitation10/ex1$
```

This output makes sense, given our program

```
int main(int argc, char** argv){
    int num = 1;
    num = add_five(num);
    print_num(num);
}
```

Something Special About Makefiles

- ▶ Makefiles have a very useful feature to them
 - Rules will be run if their prerequisites are “out of date”
 - For example, when making object files, *only modified C files will be recompiled by make*
 - This can be very beneficial... but also tricky
 - It saves *a lot* of compile time by only recompiling new changes
 - But... some changes, like changes to header files, won't allow make to recompile (since we typically don't have rules dependent on header files)

Something Special About Makefiles

- ▶ To use our previous example, if I modify main.c, **only** the main.o rule will run to update main.o (and, of course, the rule to link the files into the executable)
- ▶ The file2.o and file3.o files will be deemed up to date, since file2.c and file3.c haven't changed
- ▶ If no files have been modified since the last make command, it will tell you so by saying:

```
make: Nothing to be done for 'all'.
```

Makefiles - A Less Gentle Introduction

- ▶ While makefiles have useful functionality, we still have some issues...
- ▶ For example, why does our file name start at file2, and not file1?
 - Well, that was a small mistake...
 - No big issue, we can just change it to be file1.c and file2.c
 - ...right?

Makefiles - A Less Gentle Introduction

- ▶ The downside to our Makefile is that all of the file names and object files were hardcoded in
 - Meaning, when we change a file name, or even add a new file, we have to *rewrite* our Makefile

EVERY TIME

Optimizing the Makefile

Before we get started, let's examine the makefile one more time. All of our object rules are separate, so if we add another file, we'd have to add another individual rule with individual commands... which is tedious.

Let's combine our object file rules into **one** rule.

```
all: exe
exe: main.o file2.o file3.o
    gcc -o exe main.o file2.o file3.o
main.o: main.c
    gcc -c main.c -o main.o
file2.o: file2.c
    gcc -c file2.c -o file2.o
file3.o: file3.c
    gcc -c file3.c -o file3.o
```


Optimizing the Makefile

At this point, we've gotten rid of all of the other rules that basically replicate the same command. We still need to modify this new rule a bit, since we're still only using main.c as a prerequisite and only compiling main.c. Let's bring over the other files and commands.

```
all: exe
exe: main.o file2.o file3.o
    gcc -o exe main.o file2.o file3.o
main.o file2.o file3.o: main.c
    gcc -c main.c -o main.o
```

Optimizing the Makefile

Now the makefile should be back to working order.

This makes it a very small bit easier to add new files, but we still run into the problem of changing file names.

All of the file names are still hardcoded into the rule. To fix this issue, we need to introduce **lists** and **wildcards**

```
all: exe
exe: main.o file2.o file3.o
    gcc -o exe main.o file2.o file3.o
main.o file2.o file3.o: main.c file2.c file3.c
    gcc -c main.c -o main.o
    gcc -c file2.c -o file2.o
    gcc -c file3.c -o file3.o
```

Makefiles: Lists and Wildcards

- ▶ Instead of specifying each individual c file, I want the makefile to prune my directory and find all files ending in .c
 - This solves the problem of being able to find all of the files without being too explicit in the makefile
- ▶ Recall a command we've seen before (albeit very briefly)

```
ls *.c
```

- ▶ This is a shell command that prints all files in a directory with the .c extension
 - Which sounds a lot like what we want for our makefile...
 - The `*` is called a **wildcard** - it uses a qualifier to do something with *every* file that has that qualifier (in this case, .c)

The * Wildcard

- ▶ We saw about listing all files with a .c extension

```
ls *.c
```

- ▶ We can also use the wildcard to copy all files in a directory

```
cp my_dir/* my_new_dir
```

- This will copy all files in my_dir and place them in my_new_dir
- If I only wanted to copy .c files from my_dir, I can use:

```
cp my_dir/*.c my_new_dir
```

Makefiles: Lists and Wildcards

- ▶ Using the `*` qualifier, we can find every C file
- ▶ But, we need some kind of variable to store them in
- ▶ Makefiles have some interesting syntax, which is arguably more confusing than C :D
- ▶ That being said, let's dissect how we can declare lists and use wildcards

Makefiles: Lists and Wildcards

Telling the Makefile to interact with the shell and run the `find *.c` command, then storing the result in the SOURCES list

```
SOURCES=$(shell find *.c)
```

Telling the Makefile to interact with the shell and run the `ls *.c` command, then storing the result in the SOURCES list

```
SOURCES=$(shell ls *.c)
```

Telling the Makefile to use its own wildcard function to find files with a `.c` extension and storing them in the SOURCES list

```
SOURCES=$(wildcard *.c)
```

Makefiles: Lists and Wildcards

```
SOURCES=$(shell find *.c)
```

```
SOURCES=$(shell ls *.c)
```

```
SOURCES=$(wildcard *.c)
```

THESE ALL DO THE SAME THING!

The important thing is that we can store all of the source files (.c files) in our SOURCES list

Makefiles: Lists and Wildcards

- ▶ Now that we have a list of our sources, we can generate a list of object files we want to create from the source files
- ▶ We can do this by doing a similar process, just replacing the `.c` extension with a `.o` extension
- ▶ Let's see how we can do that in a Makefile

Makefiles: Lists and Wildcards

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

- ▶ There's a lot to unpack here.
- ▶ Let's go left to right
 - First, we're declaring a list called `OBJS`, which will store our object files
 - Second, we're using a command that Makefiles have called `patsubst`, which takes in 3 arguments
 - The first pattern we want to overwrite (we want to replace the `.c`)
 - The second pattern we want to overwrite with (replace extension as `.o`)
 - The values we're actually trying to overwrite (our `.c` source file names)

Makefiles: Lists and Wildcards

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

- ▶ The function also introduces a new wildcard, the % symbol
 - This wildcard will essentially find the name of any .c file and replace all instances of % with that file name
- ▶ This function will also execute for each file in the SOURCES list
 - We use \$(SOURCES) to access every value in our list at once

Makefiles: Lists and Wildcards

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

- ▶ Consider this scenario:
 - SOURCES is a list containing main.c, file2.c, and file3.c
 - \$(SOURCES) = main.c file2.c file3.c
- ▶ Based on this, the above line will go through each file in SOURCES, and perform the pattern substitution on the string
- ▶ Thus, \$(OBJS) = main.o file2.o file3.o

Makefiles: Lists Variables and Wildcards

- ▶ So far, we've been saying that SOURCES and OBJS are lists
 - Which is true
 - However, they are more accurately described as just variables
 - We can also make variables set to be only one value
- ▶ **All** variables in Makefiles can be accessed using the `$(var_name)` syntax
 - In the case of lists, this evaluates to a concatenation of every element in the list
 - Think of it as printing the contents of an entire array instead of accessing each element one by one

Using Variables and Wildcards in the Makefile

Revisiting our old makefile, let's try to use what we know about variables to create a more general and responsive Makefile.

To do this, we'll utilize the SOURCES and OBJS variables we've talked about so far.

```
all: exe
exe: main.o file2.o file3.o
    gcc -o exe main.o file2.o file3.o
main.o file2.o file3.o: main.c file2.c file3.c
    gcc -c main.c -o main.o
    gcc -c file2.c -o file2.o
    gcc -c file3.c -o file3.o
```

Using Variables and Wildcards in the Makefile

Now that the variables are in, we can replace some of the rules with our variable names (since the prerequisites of the exe rule are the contents of the OBJS list and the following rule are the contents of the OBJS and SOURCES list)

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
all: exe
exe: main.o file2.o file3.o
    gcc -o exe main.o file2.o file3.o
main.o file2.o file3.o: main.c file2.c file3.c
    gcc -c main.c -o main.o
    gcc -c file2.c -o file2.o
    gcc -c file3.c -o file3.o
```

Using Variables and Wildcards in the Makefile

There is still one more problem we need to resolve.

For the `exe` rule, we can replace the object files with `$(OBJJS)` to make the `exe` rule dynamic.

The main issue, however, is the `$(OBJJS)` rule. All of the files are still being compiled manually. To fix this, we'll need some more wildcards.

```
SOURCES=$(shell find *.c)
OBJJS=$(patsubst %.c, %.o, $(SOURCES))
all: exe
exe: $(OBJJS)
    gcc -o exe $(OBJJS)
$(OBJJS): $(SOURCES)
    gcc -c main.c -o main.o
    gcc -c file2.c -o file2.o
    gcc -c file3.c -o file3.o
```

Examining Makefile Wildcards

- ▶ Let's look at some wildcards to see how they'll be useful to us in our rules
 - @ → filename of the target, or if multiple targets, name of the target which caused the rule to execute
 - < → name of the **first** prerequisite
 - ^ → names of **all** prerequisites

Using Variables and Wildcards in the Makefile

Looking at these wildcards, trying to write the instructions for the OBJS rule seems tricky...

They won't let us iterate over each of the prerequisite and target pairs like we'd want (for example, matching main.c and main.o)

So... we'll need to revisit something else.

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
all: exe
exe: $(OBJS)
    gcc -o exe $(OBJS)
$(OBJS): $(SOURCES)
    gcc -c main.c -o main.o
    gcc -c file2.c -o file2.o
    gcc -c file3.c -o file3.o
```

Using Variables and Wildcards in the Makefile

Let's revisit the % wildcard.

This wildcard performs a string matching with the given suffix, then swaps that matching string into place.

For example, if we have %.c and the wildcard finds a main.c file, it replaces instances of the wildcard with 'main'.

So, what if we made our rule with the % wildcard?

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
all: exe
exe: $(OBJS)
    gcc -o exe $(OBJS)
?: ?
    gcc -c main.c -o main.o
    gcc -c file2.c -o file2.o
    gcc -c file3.c -o file3.o
```

Using Variables and Wildcards in the Makefile

Now, the rule can be interpreted as:

For all files ending with `.c`, make a corresponding `.o`, one file at a time

By using the wildcard instead of a list, we can compile each individual file instead of trying to use the whole list at once. This is the behavior we want.

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
all: exe
exe: $(OBJS)
    gcc -o exe $(OBJS)
%.o: %.c
    gcc -c main.c -o main.o
    gcc -c file2.c -o file2.o
    gcc -c file3.c -o file3.o
```

Using Variables and Wildcards in the Makefile

The final step is to use the other wildcards we learned about: ^ and @.

Recall that @ is the name of the target (in this case, the .o file) and ^ is the list of all prerequisites (which, due to the % wildcard, is only 1 .c file).

So, let's replace the hardcoded file names with these wildcards.

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
all: exe
exe: $(OBJS)
    gcc -o exe $(OBJS)
%.o: %.c
    gcc -c main.c -o main.o
    gcc -c file2.c -o file2.o
    gcc -c file3.c -o file3.o
```

Using Variables and Wildcards in the Makefile

Finally, this is our shiny new makefile. No matter what changes we make to filenames, or however files we add or remove from our directory, the simple make command will be able to compile our program.

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
all: exe
exe: $(OBJS)
    gcc -o exe $(OBJS)
%.o: %.c
    gcc -c $^ -o $@
```

More Functionality

- ▶ While this Makefile is dynamic, there are still some more things we can add
- ▶ Specifically, we need things like C compiler flags
 - `-g`, `-std=c99`, `-Wall`, `-Werror`, etc.
- ▶ We might also want an easy way to clean up all of the generated object files
 - Using a `clean` rule

More Functionality - Variables

Let's add in the compiler flags that we want to compile our program with. We'll make a variable called CFLAGS and store them there. Then, when we go to compile our source files, we can add these CFLAGS to the compile command.

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
all: exe
exe: $(OBJS)
    gcc -o exe $(OBJS)
%.o: %.c
    gcc -c $^ -o $@
```

More Functionality - Variables

Now, our Makefile has a variable where we can set and change the compiler flags as needed, without having to edit the rest of the contents of the makefile.

Let's also extend this idea to the name of the executable. It might be better practice to store it in a variable so we don't have to change multiple instances if we want to alter its name.

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
CLFLAGS=-g -Wall -std=c99
all: exe
exe: $(OBJS)
    gcc -o exe $(OBJS)
%.o: %.c
    gcc -c $^ -o $@ $(CLFLAGS)
```


More Functionality - Variables

Once again, we've made it easier to be able to alter the name of the executable without having to change more than one line in our Makefile.

The last thing we might want to consider is the `clean` rule, which will remove the executable and the object files to keep the environment clean.

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
CFLAGS=-g -Wall -std=c99
TARGET=exe
all: $(TARGET)
$(TARGET): $(OBJS)
    gcc -o $(TARGET) $(OBJS)
%.o: %.c
    gcc -c $^ -o $@ $(CFLAGS)
```

More Functionality - Clean

Let's design our clean function.

Does the clean function have any dependencies?

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
CFLAGS=-g -Wall -std=c99
TARGET=exe
all: $(TARGET)
$(TARGET): $(OBJS)
    gcc -o $(TARGET) $(OBJS)
%.o: %.c
    gcc -c $^ -o $@ $(CFLAGS)
clean:
```

More Functionality - Clean

Let's design our clean function.

Does the clean function have any dependencies?

Not really, we only want to delete files if they exist, and do nothing if they don't. We don't need to build any files for clean to work.

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
CFLAGS=-g -Wall -std=c99
TARGET=exe
all: $(TARGET)
$(TARGET): $(OBJS)
    gcc -o $(TARGET) $(OBJS)
%.o: %.c
    gcc -c $^ -o $@ $(CFLAGS)
clean:
```

More Functionality - Clean

Let's design our clean function.

Does the clean function have any dependencies?

Not really, we only want to delete files if they exist, and do nothing if they don't. We don't need to build any files for clean to work.

What commands should clean run?

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
CFLAGS=-g -Wall -std=c99
TARGET=exe
all: $(TARGET)
$(TARGET): $(OBJS)
    gcc -o $(TARGET) $(OBJS)
%.o: %.c
    gcc -c $^ -o $@ $(CFLAGS)
clean:
```

More Functionality - Clean

Since we're only trying to delete object files and the executable, we can simply run `rm` to remove the object list and the target.

In order to clean our environment, we can type `make clean` into the command line.

This just about sums up our basic Makefile!

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
CLFAGS=-g -Wall -std=c99
TARGET=exe

all: $(TARGET)

$(TARGET): $(OBJS)
    gcc -o $(TARGET) $(OBJS)

%.o: %.c
    gcc -c $^ -o $@ $(CFLAGS)

clean:
    rm -f $(TARGET)
    rm -f $(OBJS)
```

Our Basic Makefile

- ▶ This Makefile is a great start
- ▶ It allows us to have a variable number of C source files with variable names, and will still allow us to compile to the executable with a simple command
- ▶ But... there's even more we can do
- ▶ Before we talk about it, let's review the C preprocessor

The C Preprocessor

- ▶ The C preprocessor comes in before the compiler and alters the file, basically performing a search and replace
- ▶ There's *a ton* of stuff we can do with the C preprocessor, but let's start with some simple things
 - Let's start with a simple integer constant

The C Preprocessor

```
#define TRUE 1
#define FALSE 0
```

- ▶ In general, we dislike “magic numbers” in programming
 - Basically, numbers that come from seemingly nowhere
- ▶ We can fix this idea by labeling these magic numbers with a name
 - In this case, replace the magic number 1 with TRUE and the magic number 0 with FALSE
- ▶ The C preprocessor will come in before compilation and replace all instances of TRUE with 1 and all instances of FALSE with 0

The C Preprocessor

- ▶ We can also extend the use of these “macros”
- ▶ What if I want to write a min and max function?
 - We can use the C preprocessor to do this without unnecessary function calls

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))  
#define MIN(x, y) ((x) < (y) ? (x) : (y))
```

- ▶ It uses the preprocessor and ternary operators to implement a min and max function

The C Preprocessor

- ▶ Let's apply the preprocessor to debugging
- ▶ We can create macros that control whether or not we want to specify debugging prints
- ▶ If we do want the debugging control, insert desired print statements into our program
- ▶ If not, ignore the printf statements

The C Preprocessor - Debugging

```
#define DEBUG
#ifdef DEBUG
#define PDEBUG(...) printf(__VA_ARGS__)
#else
#define PDEBUG(...)
```

The C Preprocessor

Define the DEBUG macro



```
#define DEBUG
#ifdef DEBUG
#define PDEBUG(...) printf(__VA_ARGS__)
#else
#define PDEBUG(...)
#endif
```

The C Preprocessor

If the debug macro has been defined, do the next line (which it is defined)

```
#define DEBUG
#ifdef DEBUG
#define PDEBUG(...) printf(__VA_ARGS__)
#else
#define PDEBUG(...)
#endif
```

The C Preprocessor

Define the PDEBUG macro to be a replacement for printf.

This syntax says that all arguments passed to PDEBUG will be copy-pasted into printf

```
#define DEBUG
#ifdef DEBUG
#define PDEBUG(...) printf(__VA_ARGS__)
#else
#define PDEBUG(...)
#endif
```

The C Preprocessor

If DEBUG was not defined, then define PDEBUG to be replaced with empty text

Basically, PDEBUG will be replaced with "", which won't print anything

```
#define DEBUG
#ifdef DEBUG
#define PDEBUG(...) printf(__VA_ARGS__)
#else
#define PDEBUG(...)
#endif
```

The C Preprocessor - Debugging

Debug is defined, so print statements are enabled

```
#define DEBUG
#ifdef DEBUG
#define PDEBUG(...) printf(__VA_ARGS__)
#else
#define PDEBUG(...)
#endif
```

Debug is not defined, so print statements are disabled

```
// #define DEBUG
#ifdef DEBUG
#define PDEBUG(...) printf(__VA_ARGS__)
#else
#define PDEBUG(...)
#endif
```


The C Preprocessor

- ▶ This can obviously be very useful for debugging
- ▶ We don't need to constantly add/delete these statements, we can just use the PDEBUG macro in place of printf in our code, and define DEBUG to enable print statements
- ▶ But, we have to alter and recompile the C file every time...
- ▶ Enter, once again, the Makefile

The C Preprocessor and the Makefile

- ▶ We can actually tell the Makefile to define macros for a C source file
- ▶ We can use a gcc compiler flag, `-D`, to do this
- ▶ For example, for the `DEBUG` macro in the previous example, we can tell make to compile with the flag `-DDEBUG` to define the debug macro
- ▶ We can also tell the compiler what the value of the macro should be
 - If I wanted the `DEBUG` macro to have a value of 5, I can tell gcc `-DDEBUG=5`

The C Preprocessor and the Makefile

- ▶ This means that in our C code, we can remove the initial `#define DEBUG`
- ▶ Instead of commenting it out every time, we can situationally tell the compiler to initialize it using a rule in the Makefile
- ▶ Let's think about how we can do this

More Functionality - Debugging

This is the Makefile we had previously. This time, let's modify it so that we can situationally allow debugging statements.

First, we'll need a new variable `DEBUG` which will contain the debug flags we want to use, and a debug rule that we can call with `make debug`

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
CFLAGS=-g -Wall -std=c99
TARGET=exe

all: $(TARGET)

$(TARGET): $(OBJS)
    gcc -o $(TARGET) $(OBJS)

%.o: %.c
    gcc -c $^ -o $@ $(CFLAGS)

clean:
    rm -f $(TARGET)
    rm -f $(OBJS)
```

More Functionality - Debugging

We'll also need to append this debug variable onto our object file compilation rule, so the debug command will also get passed in when we want.

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
CLFLAGS=-g -Wall -std=c99
TARGET=exe
DEBUG=
all: $(TARGET)
debug:
$(TARGET): $(OBJS)
    gcc -o $(TARGET) $(OBJS)
%.o: %.c
    gcc -c $^ -o $@ $(CFLAGS)
clean:
    rm -f $(TARGET)
    rm -f $(OBJS)
```

More Functionality - Debugging

Originally, the DEBUG variable is empty, so that when the user calls a regular make, it will never get initialized, and therefore never initialize our macro, which won't swap PDEBUG with printf's. We need to update its value in `debug`

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
CLFLAGS=-g -Wall -std=c99
TARGET=exe
DEBUG=
all: $(TARGET)
debug:
$(TARGET): $(OBJS)
    gcc -o $(TARGET) $(OBJS)
%.o: %.c
    gcc -c $^ -o $@ $(CFLAGS) $(DEBUG)
clean:
    rm -f $(TARGET)
    rm -f $(OBJS)
```

More Functionality - Debugging

Since we're not running shell commands in the debug rule, we can just tell debug to change the value of one of our previously declared variables.

Now that DEBUG has been properly set, however, we need to rebuild the executable. Thus, we can add a second stage to the debug rule telling it to build the executable, or TARGET

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
CLFLAGS=-g -Wall -std=c99
TARGET=exe
DEBUG=
all: $(TARGET)
debug: DEBUG=-DDEBUG
$(TARGET): $(OBJS)
    gcc -o $(TARGET) $(OBJS)
%.o: %.c
    gcc -c $^ -o $@ $(CLFLAGS) $(DEBUG)
clean:
    rm -f $(TARGET)
    rm -f $(OBJS)
```

More Functionality - Debugging

Now, whenever we run `make debug`, the `DEBUG` macro will be defined, and print statements will be enabled.

Whenever we run `make`, however, the macro will not be defined and the print statements won't appear.

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.o, $(SOURCES))
CLFAGS=-g -Wall -std=c99
TARGET=exe
DEBUG=
all: $(TARGET)
debug: DEBUG=-DDEBUG
debug: $(TARGET)
$(TARGET): $(OBJS)
    gcc -o $(TARGET) $(OBJS)
%.o: %.c
    gcc -c $^ -o $@ $(CLFAGS) $(DEBUG)
clean:
    rm -f $(TARGET)
    rm -f $(OBJS)
```


Drawbacks

- ▶ As we mentioned before, the makefile will only execute rules when the source has been modified
- ▶ This method of debugging enabling, however, doesn't modify the source file...
- ▶ So when we switch between make and make debug, it's not going to recompile the program, meaning it won't update the value of the DEBUG variable
- ▶ In order to actually push our changes through, we'll need to first `make clean` to get rid of the old executable, then run the desired make command

Drawbacks

```
jbk52@thoth:~/cs449/recitations/recitation10/ex5$ make
gcc -c main.c -o main.o -g -Wall -std=c99
gcc -o exe main.o
jbk52@thoth:~/cs449/recitations/recitation10/ex5$ make debug
make: Nothing to be done for 'debug'.
jbk52@thoth:~/cs449/recitations/recitation10/ex5$ ./exe
jbk52@thoth:~/cs449/recitations/recitation10/ex5$ █
```

```
jbk52@thoth:~/cs449/recitations/recitation10/ex5$ make debug
gcc -c main.c -o main.o -g -Wall -std=c99 -DDEBUG=1
gcc -o exe main.o
jbk52@thoth:~/cs449/recitations/recitation10/ex5$ make
make: Nothing to be done for 'all'.
jbk52@thoth:~/cs449/recitations/recitation10/ex5$ ./exe
Debugging has been enabled.
jbk52@thoth:~/cs449/recitations/recitation10/ex5$ █
```

Notice that in the first example, we first ran `make` to generate the executable. After, we ran `make debug` to try to add our print statements, but the Makefile told us there was nothing to be done, and the print statements weren't executed.

In the second example, however, the opposite was true.

Drawbacks

```
jbk52@thoth:~/cs449/recitations/recitation10/ex5$ make debug
gcc -c main.c -o main.o -g -Wall -std=c99 -DDEBUG=1
gcc -o exe main.o
jbk52@thoth:~/cs449/recitations/recitation10/ex5$ ./exe
Debugging has been enabled.
jbk52@thoth:~/cs449/recitations/recitation10/ex5$ make clean
rm -f exe
rm -f main.o
jbk52@thoth:~/cs449/recitations/recitation10/ex5$ make
gcc -c main.c -o main.o -g -Wall -std=c99
gcc -o exe main.o
jbk52@thoth:~/cs449/recitations/recitation10/ex5$ ./exe
jbk52@thoth:~/cs449/recitations/recitation10/ex5$ █
```

Using make clean will allow the makefile to generate the executable with the proper definition of the DEBUG macro.

If we want it to work every time without using make clean, we can alter the rules to execute clean before trying to make the target. This does, however, sacrifice some of the benefits of Makefiles.

Last Example - Compiling Shared Objects with a Makefile

- ▶ Let's use our knowledge of makefiles to do something beneficial to our upcoming project: let's write some simple rules that will allow us to compile shared object files
- ▶ To do this, let's add a filesystem that will hold the .c source files for the shared objects as well as the resulting .so files

Last Example - Compiling Shared Objects with a Makefile

- ▶ Let's use 2 folders
 - `so_sources`
 - `so_objects`
- ▶ We'll compile each individual source file in `so_sources` to a resulting `.so` file in `so_objects`
- ▶ For simplicity, we'll also assume that each shared object only uses 1 `.c` file

Makefile For Shared Objects

To start, we're going to find our source files and make object files with a .so

Our makefile is going to change in the sense that we're not making a target anymore, just the individual shared object files. So, we'll use the wildcard rule to generate them.

Also, all is only going to depend on our objects now.

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.so, $(SOURCES))
```

Makefile For Shared Objects

Now we need to think about the commands we want to run to generate the shared object files. If we recall from Lab 5, we want:

```
gcc -c <fname>.c -o <fname>.so -shared
```

So, we'll use a similar setup to our old object file rule, just modifying the command slightly to match the one above.

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.so, $(SOURCES))
all: $(OBJS)
%.so: %.c
```

Makefile For Shared Objects

That should just about do it!
Well, almost...

The meat of the Makefile is finished, this should hopefully compile all of the c source files to the .so object files.

However, we didn't tell the Makefile about our filesystem. We'll need to add these paths to the Makefile.

```
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.so, $(SOURCES))
all: $(OBJS)
%.so: %.c
    gcc -c $^ -o $@ -shared
```


Makefile For Shared Objects

We also need to update the definition of sources to include the new directories.

Specifically, it's still going to look for our source files in the current directory, but they'll be in the `so_sources` directory, which is where we want to pull the sources from. We'll also need to do something similar to the objects.

```
SOURCE_DIR=so_sources
OBJECT_DIR=so_objects
SOURCES=$(shell find *.c)
OBJS=$(patsubst %.c, %.so, $(SOURCES))
all: $(OBJS)
%.so: %.c
    gcc -c $^ -o $@ -shared
```

Makefile For Shared Objects

Now, we need to edit the OBJS definition.

Currently, it replaces all instances of .c with .so. We want to do something similar to change the destination directory.

Instead of going to so_sources/.so, we need to pattern substitute so_sources with so_objects.

```
SOURCE_DIR=so_sources
OBJECT_DIR=so_objects
SOURCES=$(shell find $(SOURCE_DIR)/*.c)
OBJ=$(patsubst %.c, %.so, $(SOURCES))
all: $(OBJ)
%.so: %.c
    gcc -c $^ -o $@ -shared
```

Makefile For Shared Objects

In this case, we changed the initial OBJS definition to be TMP_OBJJS to avoid a self referential definition using patsubst. Next, we'll tell patsubst to replace every instance of the source directory with the object directory.

The last thing we have to do is update our rule to also be looking in the right places for these files.

```
SOURCE_DIR=so_sources
OBJECT_DIR=so_objects
SOURCES=$(shell find $(SOURCE_DIR)/*.c)
TMP_OBJJS=$(patsubst %.c, %.so, $(SOURCES))
OBJJS=$(patsubst $(SOURCE_DIR)/%, $(OBJECT_DIR)/%, $(TMP_OBJJS))
all: $(OBJJS)
%.so: %.c
    gcc -c $^ -o $@ -shared
```

Makefile For Shared Objects

There's just one problem with this makefile. It assumes that the object folder already exists, which it might not if we want to write a clean rule.

So, let's tell the Makefile that `all` will also depend on the existence of the object directory. Then, we'll make a rule to generate the object directory if it doesn't exist already.

```
SOURCE_DIR=so_sources
OBJECT_DIR=so_objects
SOURCES=$(shell find $(SOURCE_DIR)/*.c)
TMP_OBJS=$(patsubst %.c, %.so, $(SOURCES))
OBJS=$(patsubst $(SOURCE_DIR)/%, $(OBJECT_DIR)/%, $(TMP_OBJS))
all: $(OBJS)
$(OBJECT_DIR)/%.so: $(SOURCE_DIR)/%.c
    gcc -c $^ -o $@ -shared
```

Makefile For Shared Objects

Notice that we put the directory as a prerequisite first. We can't build the objects until we've made the directory, which is why we've chosen this specific order.

Now, if the object directory rule is executed, it's because the object directory doesn't exist. So, we just need to have that rule make the directory.

```
SOURCE_DIR=so_sources
OBJECT_DIR=so_objects
SOURCES=$(shell find $(SOURCE_DIR)/*.c)
TMP_OBJS=$(patsubst %.c, %.so, $(SOURCES))
OBJS=$(patsubst $(SOURCE_DIR)/%, $(OBJECT_DIR)/%, $(TMP_OBJS))
all: $(OBJECT_DIR) $(OBJS)
$(OBJECT_DIR)/%.so: $(SOURCE_DIR)/%.c
    gcc -c $^ -o $@ -shared
$(OBJECT_DIR):
```

Makefile For Shared Objects

Now our makefile can read in all source files in the source directory and place them in the object directory, even if that directory doesn't yet exist.

Since we've also stored all of our objects in their own directory, our `clean` rule becomes much more simple to write.

```
SOURCE_DIR=so_sources
OBJECT_DIR=so_objects
SOURCES=$(shell find $(SOURCE_DIR)/*.c)
TMP_OBJS=$(patsubst %.c, %.so, $(SOURCES))
OBJS=$(patsubst $(SOURCE_DIR)/%, $(OBJECT_DIR)/%, $(TMP_OBJS))
all: $(OBJECT_DIR) $(OBJS)
$(OBJECT_DIR)/%.so: $(SOURCE_DIR)/%.c
    gcc -c $^ -o $@ -shared
$(OBJECT_DIR):
    mkdir -p $@
```

Makefile For Shared Objects

This is our final makefile for generating shared objects.

The `clean` function will remove the directory and any files that exist within it.

```
SOURCE_DIR=so_sources
OBJECT_DIR=so_objects
SOURCES=$(shell find $(SOURCE_DIR)/*.c)
TMP_OBJS=$(patsubst %.c, %.so, $(SOURCES))
OBJS=$(patsubst $(SOURCE_DIR)/%, $(OBJECT_DIR)/%, $(TMP_OBJS))
all: $(OBJECT_DIR) $(OBJS)
$(OBJECT_DIR)/%.so: $(SOURCE_DIR)/%.c
    gcc -c $^ -o $@ -shared
$(OBJECT_DIR):
    mkdir -p $@
clean:
    rm -rf $(OBJECT_DIR)
```

Takeaways

- ▶ Makefiles can be very useful for larger projects with multiple files
- ▶ They can also extend to languages other than C!
- ▶ There are also many ways to write Makefiles, as we've witnessed
- ▶ Makefile design can be centered around functionality or efficiency
 - Like in our `make debug` example
- ▶ It's up to you how to write and store your code, and how to write Makefiles in response to your own style