Fancier make Options

Recall that the input to make is a Makefile, which contains one or more rules that specify how to produce a target from its prerequisites (the files it depends on). The rule is comprised of the target specification, followed by a colon, and then a list of the prerequisite files. After the list of prerequisites, there is a newline, and then any commands required to rebuild that target from the prerequisites. The commands may appear over multiple lines; however, each line must begin with a TAB character (multiple spaces will not work, and accidentally using them instead of a TAB is often the source of problems with a Makefile).

When you run make, you can specify a particular target to build (if you do not specify one, make uses the first target in the Makefile as the default target). To build the target, make will first check if it is up to date. Checking if a target is up to date first requires ensuring that each prerequisite is up to date by potentially rebuilding it. This process bottoms out when make encounters a file that exists which is not itself the target of any rules. Such a file is up to date.

Once all files which a target depends on are ensured to be up to date, make checks if the target itself needs to be (re)built. First, make check if the target file exists. If not, it must be built. If the target file already exists, make compares its last-modified time (which is tracked by all major filesystems) to the last-modified times of each of the prerequisites specified in the rule. If any dependency is newer than the target file, then the target is out of date, and must be rebuilt. Note that if any of the prerequisites were rebuilt in this process, then that file will have been modified more recently than the target, thus forcing the target to be rebuilt.

An example

To be a bit more concrete, let us look at a specific example of a Makefile:

```
myProgram: oneFile.o anotherFile.o
    gcc -o myProgram oneFile.o anotherFile.o
oneFile.o: oneFile.c oneHeader.h someHeader.h
    gcc -std=gnu99 -pedantic -Wall -c oneFile.c
anotherFile.o: anotherFile.c anotherHeader.h someHeader.h
    gcc -std=gnu99 -pedantic -Wall -c anotherFile.c
```

In this Makefile there are three targets: myProgram, oneFile.o, and anotherFile.o. If we just type make, then make will attempt to (re)build myProgram, as that is the first target in the file, and thus the default. This target depends on oneFile.o and anotherFile.o, so the first thing make will do is make the oneFile.o target (much as if we had typed make oneFile.o).

oneFile.o depends on one .c and two .h files, none of which are targets of other rules. Therefore, make does not need to rebuild them. If they do not already exist, make will give an error like this:

```
make: *** No rule to make target 'oneHeader.h', needed by 'oneFile.o'.  Stop.
```

Assuming that all three of these .c/.h files exist, make will see if oneFile.o does not exist, or if any of those three files are newer than it. If so, then make will rebuild the file by running the specified gcc command. If oneFile.o already exists and is newer than the relevant source files, then nothing needs to be done to build it.

After processing oneFile.o, make does a similar process for anotherFile.o. After that completes, it checks if it needs to build myProgram (that is, if either myProgram does not exist, or either of the object files that it depends on are newer than it). If so, it runs the specified gcc command (which will link the object files and produce the binary called myProgram). If not, it will produce the message:

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

Observe that, because of the way this procedure works, if you were to change code in oneFile.c, then only one of the two object files would be recompiled (oneFile.o), and then the program would be re-linked. The other object file (anotherFile.o) would not need to be recompiled. While this may seem like an insignificant difference for two files, if the project had 50 files, compiled with heavy optimizations, the difference between recompiling all 50, and only recompiling one would be a noticeable amount of time.

Variables

The way we have written our example Makefile has a lot of copying and pasting—something we want to avoid in anything we do. In particular, we might notice that we have the same compiler options in many places. If we wanted to change these options (e.g., to turn on optimizations, or add a debugging flag), we would have to do it in every place. Instead, we would prefer to put the compiler options in a variable, and use that variable in each of the relevant rules. For example, we might change our previous example to the following:

```
CFLAGS=-std=gnu99 -pedantic -Wall
myProgram: oneFile.o anotherFile.o
    gcc -o myProgram oneFile.o anotherFile.o
oneFile.o: oneFile.c oneHeader.h someHeader.h
    gcc $(CFLAGS) -c oneFile.c
anotherFile.o: anotherFile.c anotherHeader.h someHeader.h
    gcc $(CFLAGS) -c anotherFile.c
```

Here, we define a variable CFLAGS, which we set equal to our desired compilation flags. We then use that variable by putting its name inside of $() in the rules. Note that changes to the Makefile do not automatically outdate targets which use them, so they will not necessarily be rebuilt with the new flags if you just type make after making the change (although you could make them all depend on

Makefile).

Clean

A common target to put in a Makefile is a clean target. The clean target is a bit different in that it does not actually create a file called clean (it is therefore called a “phony” target). Instead, it is a target intended to remove the compiled program, all object files, all editor backups (*.c~ *.h~), and any other files that you might consider to be cluttery. This target gets used to either force the entire program to be rebuilt (e.g., after you change various compilation flags in the Makefile), or if you just need to clean up the directory, leaving only the source files (e.g., if you are going to zip or tar up the source files to distribute them to someone).

We might add a clean target to our Makefile as follows:

```
.PHONY: clean
clean:
    rm -f myProgram *.o *.c~ *.h~
```

Note that the .PHONY: clean tells make that clean is a phony target—we do not actually expect it to create a file called “clean”, nor would we want to consider it up to date and skip its commands if a file called “clean” already existed (as there are no prerequisites, it would be considered up to date if it existed). If we wanted other phony targets, we would list them all as if they were prerequisites of the .PHONY target (e.g. .PHONY: clean depend).

In general, you should add a clean target to your Makefiles, as most people will expect one to be present, and it can be quite useful.

Generic rules

Our example Makefile improved slightly when we used a variable to hold the compilation flags. However, our Makefile still suffers from a lot of repetition, and would be a pain to maintain if we had more than a few sources files. If you look at what we wrote, we are doing pretty much the same thing to compile each of our .c source files into an object file. Whenever we find ourselves repeating ourselves, there should be a better way.

In make, we can write generic rules. A generic rule lets us specify that we want to be able to build (something).o from (something).c, where we represent the something with a percent-sign (%). As a first step, we might write (note that `#` indicates a comment to the end of the line):

```
# A good start, but we lost the dependencies on the header files
CFLAGS=-std=gnu99 -pedantic -Wall
myProgram: oneFile.o anotherFile.o
    gcc -o myProgram oneFile.o anotherFile.o
%.o: %.c
    gcc $(CFLAGS) -c $<
.PHONY: clean
clean:
    rm -f myProgram *.o *.c~ *.h~
```

Here, we have replaced the two rules we had for each object file with one generic rule. It specifies how to make a file ending with .o from a file of the same name, except with .c instead of .o. In this rule, we cannot write the literal name of the source file, as it changes for each instance of the rule. Instead, we have to use the special built-in variable `$<`, which make will set to the name of the first prerequisite of the rule (in this case, the name of the .c file).

However, we have introduced a significant problem now. We have made it so that our object files no longer depend on the relevant header files. If we were to change a header file, then make might not rebuild all of the relevant object files. Such a mistake can cause strange and confusing bugs, as one object file may expect data in an old layout but the code will now be passed data in a different layout. We could make every object file depend on every header file (by writing %.o : %.c *.h), however, this approach is overkill—we would definitely rebuild everything that we need to when we change a header file, because we would rebuild every object file, even if we only need to rebuild a few.

We can fix this in a better way by adding the extra dependency information to our Makefile:

```
# This fixes the problem
CFLAGS=-std=gnu99 -pedantic -Wall
myProgram: oneFile.o anotherFile.o
    gcc -o myProgram oneFile.o anotherFile.o
%.o: %.c
    gcc $(CFLAGS) -c $<
.PHONY: clean
clean:
    rm -f myProgram *.o *.c~ *.h~
oneFile.o: oneHeader.h someHeader.h
anotherFile.o: anotherHeader.h someHeader.h
```

Here, we still have the generic rule, but we also have specified the additional dependencies separately. Even though it looks like we have two rules, make understands that we are just providing additional dependence information because we have not specified any commands. If we did specify commands in the, they would supersede the generic rules for those targets.

Managing all of this dependency information by hand would, of course, be tedious and error-prone. The programmer would have to figure out every file which is transitively included by each source file, and keep the information up to date as the code changes. Instead, there is a tool called makedepend which will edit the Makefile to put all of this information at the end. In its simplest usage, makedepend takes as arguments all of the source files (i.e., all of the .c and/or .cpp files), and edits the Makefile. It can also be given a variety of options, such as -I path to tell it to look for include files in the specified path. See man makedepend for more details.

Built-in generic rules

Some generic rules are so common that they are built into make, and we do not even have to write them. As you may suspect, building a .o file from a similarly named .c file is quite common, as it is what C programmers do most often. Accordingly, we do not even need to explicitly write our %.o: %.c rule if we are happy with the built-in generic rule for this pattern.

We can see the all of the rules (including both those that are built-in and those that are specified by the Makefile) by using make -p. Doing so also builds the default target as usual—if we want to avoid building anything, we can do make -p -f/dev/null to use the special file /dev/null as our Makefile (reading from /dev/null results in end-of-file right away, so the result will be a Makefile with no rules, thus it will not do anything).

If we use make -p to explore the built-in rules for building .o files from .c files, we will find:

```
%.o: %.c
# commands to execute (built-in):
    $(COMPILE.c) $(OUTPUT_OPTION) $<
```

Understanding this rule requires us to look at the definitions of COMPILE.c and OUTPUT_OPTION, which are also included in the output of make -p:

```
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
OUTPUT_OPTION = -o $@
```

By default, CFLAGS (flags for the C-compiler) and CPPFLAGS (flags for the C preprocessor143), as well as TARGET_ARCH (flags to specify what architecture to target) are empty. By default CC (the C-compiler command) is cc (which may or may not be gcc depending on how our system is configured). The defaults for any of these variables (or any other variables) can be overridden by specifying their values in our Makefile. Note that @inOUTPUTOPTIONisaspecialvariablewhichisthenameofthecurrenttarget(muchlike@inOUTPUTOPTIONisaspecialvariablewhichisthename
is the name of the first prerequisite).

All of that may sound a bit complex but it basically boils down to the fact that the default rule is: cc -c -o something.o something.c, and we can override the specifics to get the behavior we want, while still using the default rule. That is, we might use the following Makefile:

```
CC = gcc
CFLAGS = -std=gnu99 -pedantic -Wall
myProgram: oneFile.o anotherFile.o
    gcc -o myProgram oneFile.o anotherFile.o
.PHONY: clean depend
clean:
    rm -f myProgram *.o *.c~ *.h~
depend:
    makedepend anotherFile.c oneFile.c
# DO NOT DELETE
anotherFile.o: anotherHeader.h someHeader.h
oneFile.o: oneHeader.h someHeader.h
```

Here, we have specified that we want to use gcc as the C-compiler (CC), and specified the CFLAGS that we want. Now, when we try to compile an object file from a C file, the default rule will result in

gcc -std=gnu99 -pedantic -Wall -c -o something.o something.c

You should also note that we have added another phony target, depend which runs makedepend with the two C source files that we are working with. the # DO NOT DELETE line and everything below it are what makedepend added to our Makefile when I ran make depend. Note that if we re-run makedepend (preferably via the make depend), it will look for this line to tell where to delete the old dependency information and add its new information.

Built-in functions

Our Makefile is looking more like something we could use in a large project, but we have still manually listed our source and object files in a couple places. If we were to add a new source file, but forget to update the makedepend command line, we would not end up with the right dependencies for that file when we run make depend. Likewise, we might forget to add object files in the correct places (e.g., if we add it to the compilation command line, but not the dependencies for the entire program, we may not rebuild that object file when needed).

We can fix these problems by using some of make’s built-in functions to automatically compute the set of .c files in the current directory, and then to generate the list of target object files from that list. The syntax of function calls in make is (functionNamearg1,arg2,arg3).Wecanusethe(functionNamearg1,arg2,arg3).Wecanusethe(wildcard pattern) function to generate the list of

.c files in the current directory: SRCS = (wildcard∗.c).Thenwecanusethe(wildcard∗.c).Thenwecanusethe(patsubst pattern, replacement, text) function to replace the .c endings with .o endings: OBJS = (patsubst(patsubst(SRCS)). Once we have done this, we can use (SRCS)and(SRCS)and(OBJS) in our Makefile:

```
CC = gcc
CFLAGS = -std=gnu99 -pedantic -Wall
SRCS=$(wildcard *.c)
OBJS=$(patsubst %.c,%.o,$(SRCS))
myProgram: $(OBJS)
    gcc -o $@ $(OBJS)
.PHONY: clean depend
clean:
    rm -f myProgram *.o *.c~ *.h~
depend:
    makedepend $(SRCS)
# DO NOT DELETE
anotherFile.o: anotherHeader.h someHeader.h
oneFile.o: oneHeader.h someHeader.h
```

At this point, we have a Makefile that we could use on a large-scale project. The only thing we need to do when we add source files or include new header files in existing source files is run make depend to update the dependency information. Other than that, we can build our project with make, which will only recompile the required files.

We could, however, be a little bit fancier. In a real project, we likely want to build a debug version of our code (with no optimizations, and -ggdb3 to turn on debugging information—see the next module for more info about debugging), and an optimized version of our code that will run faster (where the compiler works hard to produce improve the instructions that it generates, but those transformations generally make debugging quite difficulty). We could change our CFLAGS back and forth between flags for debugging and flags for optimization, and remember to make clean each time we switch. However, we can also just set our Makefile up to build both debug and optimized object files and binaries with different names:

```
CC = gcc
CFLAGS = -std=gnu99 -pedantic -Wall -O3
DBGFLAGS = -std=gnu99 -pedantic -Wall -ggdb3 -DDEBUG
SRCS=$(wildcard *.c)
OBJS=$(patsubst %.c,%.o,$(SRCS))
DBGOBJS=$(patsubst %.c,%.dbg.o,$(SRCS))
.PHONY: clean depend all
all: myProgram myProgram-debug
myProgram: $(OBJS)
    gcc -o $@ -O3 $(OBJS)
myProgram-debug: $(DBGOBJS)
    gcc -o $@ -ggdb3 $(DBGOBJS)
%.dbg.o: %.c
    gcc $(DBGFLAGS) -c -o $@ $<
clean:
    rm -f myProgram myProgram-debug *.o *.c~ *.h~
depend:
    makedepend $(SRCS)
    makedepend -a -o .dbg.o  $(SRCS)
# DO NOT DELETE
anotherFile.o: anotherHeader.h someHeader.h
oneFile.o: oneHeader.h someHeader.h
```

Now, if we make, we get both myProgram (the optimized version), and myProgram-debug (which is compiled for debugging).

Parallelizing computation

One useful feature of make, especially on modern multi-core systems is the ability to have it run independent tasks in parallel. If you give make the -j option, it requests that it run as many tasks in parallel as it can. You may wish to ask it to limit the number of parallel tasks to a particular number at any given time, which you can do by specifying that number as an argument to the -j option (, make -j8 runs up to 8 tasks in parallel). On large projects, this may make a significant difference in how long a build takes.

... And much more

You can use make for much more beyond just compiling C programs. In fact, you can use make for pretty much any task that you can describe in terms of creating targets from the prerequisites that they depend on. For most such tasks, you can put the parallelization capabilities of make to good use to speed up the task.

We have given you enough of an introduction to make to write a basic Makefile. However, as with many topics, we have barely scratched the surface. You can find out a lot more about make by reading the online manual: https://www.gnu.org/software/make/manual/.