Makefile concepts

Here is a straightforward makefile that describes the way an executable file called edit depends on eight object files which, in turn, depend on eight C source and three header files.

In this example, all the C files include defs.h, but only those defining editing commands include command.h, and only low level files that change the editor buffer include buffer.h.

edit : main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

cc -o edit main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

main.o : main.c defs.h

cc -c main.c

kbd.o : kbd.c defs.h command.h

cc -c kbd.c

command.o : command.c defs.h command.h

cc -c command.c

display.o : display.c defs.h buffer.h

cc -c display.c

insert.o : insert.c defs.h buffer.h

cc -c insert.c

search.o : search.c defs.h buffer.h

cc -c search.c

files.o : files.c defs.h buffer.h command.h

cc -c files.c

utils.o : utils.c defs.h

cc -c utils.c

clean :

rm edit main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

We split each long line into two lines using backslash/newline; this is like using one long line, but is easier to read. See [Splitting Long Lines](https://www.gnu.org/software/make/manual/html_node/Splitting-Lines.html#Splitting-Lines).

To use this makefile to create the executable file called edit, type:

make

To use this makefile to delete the executable file and all the object files from the directory, type:

make clean

Here is how the complete simple makefile looks when you use a variable for the object files:

objects = main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

edit : $(objects)

cc -o edit $(objects)

main.o : main.c defs.h

cc -c main.c

kbd.o : kbd.c defs.h command.h

cc -c kbd.c

command.o : command.c defs.h command.h

cc -c command.c

display.o : display.c defs.h buffer.h

cc -c display.c

insert.o : insert.c defs.h buffer.h

cc -c insert.c

search.o : search.c defs.h buffer.h

cc -c search.c

files.o : files.c defs.h buffer.h command.h

cc -c files.c

utils.o : utils.c defs.h

cc -c utils.c

clean :

rm edit $(objects)

Here is how we could write a make rule for cleaning our example editor:

clean:

rm edit $(objects)

In practice, we might want to write the rule in a somewhat more complicated manner to handle unanticipated situations. We would do this:

.PHONY : clean

clean :

-rm edit $(objects)

This prevents make from getting confused by an actual file called clean and causes it to continue in spite of errors from rm.

### Rule Syntax

In general, a rule looks like this:

targets : prerequisites

recipe

…

or like this:

targets : prerequisites ; recipe

recipe

…

### Rule Example

Here is an example of a rule:

foo.o : foo.c defs.h # module for twiddling the frobs

cc -c -g foo.c

Its target is foo.o and its prerequisites are foo.c and defs.h. It has one command in the recipe: ‘cc -c -g foo.c’. The recipe starts with a tab to identify it as a recipe.

This rule says two things:

* How to decide whether foo.o is out of date: it is out of date if it does not exist, or if either foo.c or defs.h is more recent than it.
* How to update the file foo.o: by running cc as stated. The recipe does not explicitly mention defs.h, but we presume that foo.c includes it, and that that is why defs.h was added to the prerequisites.

**Types of Prerequisites**

targets : normal-prerequisites | order-only-prerequisites

Wildcard expansion does not happen when you define a variable. Thus, if you write this:

objects = \*.o

then the value of the variable objects is the actual string ‘\*.o’. However, if you use the value of objects in a target or prerequisite, wildcard expansion will take place there. If you use the value of objects in a recipe, the shell may perform wildcard expansion when the recipe runs. To set objects to the expansion, instead use:

objects := $(wildcard \*.o)

#### **The Function wildcard**

One use of the wildcard function is to get a list of all the C source files in a directory, like this:

$(wildcard \*.c)

We can change the list of C source files into a list of object files by replacing the ‘.c’ suffix with ‘.o’ in the result, like this:

$(patsubst %.c,%.o,$(wildcard \*.c))

(Here we have used another function, patsubst. See [Functions for String Substitution and Analysis](https://www.gnu.org/software/make/manual/html_node/Text-Functions.html#Text-Functions).)

Thus, a makefile to compile all C source files in the directory and then link them together could be written as follows:

objects := $(patsubst %.c,%.o,$(wildcard \*.c))

foo : $(objects)

cc -o foo $(objects)