Makefile Basics

Authors: Tristan Chutka, Victor Delaplaine

Makefiles (and their associated program, make) provide an easy way to automate the compilation and building of programs. They become particularly useful for programs with a large amount of dependencies, where manually compiling and linking each step would be far to tedious.

A makefile could be considered akin to a script, and thus follow some syntactic rules. Targets must be a continuous word (no spaces) and are separated from sources by a colon (:). Commands must be placed on the next line, and be indented by 1 tab. Line breaks are allowed with a '\' character at the end of the line being broken.

The basic command block structure looks like so:

```
target: source

    command
```

The target is a user-specified name that refers to the group of commands (think of it like a function name). To execute the commands, call it using the make application

```
make target
```

Source refers to one or more references to other targets, or files, and represent prerequisites or dependencies for the target they are associated with.

In the following example for instance, target1 will call target2, thus performing the actions of target2, before executing the commands listed in target1. This becomes helpful in such cases as when two targets rely upon a single library which must be compiled first.

```
target1: target2

    target1_command

target2:

    target2_command
```

Command refers to any shell commands which should be executed based on the target. The most common use case is to compile programs; thus a command usually looks like such:

```
    gcc hello_world.c -o hello_world
```

Note that like python, make is whitespace sensitive; commands must have a tab character preceding the line after a target.

Additional Features

Makefiles also support the use of variables, wildcards, and suffix replacement operations. For instance, if we wanted to define an installation directory as /usr/bin, we could add this at the top of the file:

```
INSTALLDIR=/usr/bin
```

Later in the file, we can reference this location by using a selector syntax as follows:

```
$(INSTALLDIR)
```

Variables can also be overridden by command line arguments. To overwrite the `INSTALLDIR` variable with /var/lib, execute make with the flag:

```
make target INSTALLDIR =/var/lib
```

Make provides four default variables that contain information relevant to the current target. These are:

`$?`: This variable contains the list of dependencies for the current target that are more recent than the target. These would be the targets that must be re-done before executing the commands under this target.

`$@`: This variable is the name of the current target. This allows one to reference the file you are trying to make, even though this rule was matched through a pattern.

`$<`: This is the name of the current source argument

`$*`: This file is the name of the current source argument with the matched extension stripped off.

Consider this an intermediate stage between the target and source files.

Paterns

Patterns can be matched using the selector syntax using the term 'wildcard'; for instance, to match all .c files, the selector would look as such:

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

Make selectors can also be used to replace file endings as well. For instance, to change a filename of `hello_world.c` stored in `PROGS` to `hello_world.h` (say, to automatically include a header file of the same name), the selector would look as follows:

```
$(PROGS:.c=)
```

Shell Commands

Shell commands may also be executed by using the selector notation and the command 'shell':

```
SOURCES = $(shell find . -name "*.c")
```

Here, the shell command 'find' is being run, and will return a list of all .c files for the makefile to compile.

Creating Makefiles

Using these elements, we can create an advanced makefile with the following features:

1) Option to set C and C++ compiler (gcc, llvm, etc.)

2) Field to input LDFLAGS to be used by the linker

3) Field to input CFLAGS to be used by the compiler

4) Field to input C source and header files (or automated mechanism for grabbing all source files in current and child directories).

5) Field to set name of output binary

6) Ability to Make with “make” or “make all”, and clean outputs with “make clean”.

A basic makefile, one capable of building any individual file in a folder, would resemble something like this:

```
CC?= gcc
CFLAGS?= -g -Wall
LDFLAGS?=
SOURCES?=
TARGET?=
PROGS=$(wildcard *.c)

all: $(PROGS:.c=)

$(TARGET): $(SOURCES)
	$(CC) $(CFLAGS) $(TARGET) -o $(TARGET) $(LDFLAGS)

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

clean:
	rm -f $(PROGS:.c=)
```

The first 6 lines are viable declarations, using the ?= operator to signify a default value in the event that no input is provided from command line arguments. PROGS is a wildcard selector to grab all .c files in the directory, should you wish to build all. An explanation of the variables is as follows:

CC: This is the c/c++ complier

LDFLAGS: This should be a list of link (or load) directives such as loading the math.h library

CFLAGS: This should be a list of complier directives/options

TARGET: This is the name of the desired binary output (the program name)

SOURCES: This is a list of C files required by the target. This list is dynamically generated and includes all files in the current working directory and subfolders.

The first target, all:, is a default target which can be included or omitted, and is the default target run when calling make with no arguments. This specifies the list of PROGS, created above, as its list of dependencies, and will, for every file in the list, attempt to call a target with the same name as the file. This is where the pattern selectors come into play; a file not matching any of the prior targets will fall through to target 3, and be individually compiled into a file of its base name.

If, however, you want to build a single project with many dependencies, the following makefile would be more suitable:

```
CC = gcc
LDFLAGS?=
CFLAGS?=  -g
SOURCES = $(shell find . -name "*.c")
TARGET = main
OBJS = $(SOURCES:.c=.o)

all: .depend $(TARGET)

# This is to get the depencies of all the c files

.depend: $(SOURCES)
	rm -f ./.depend
	$(CC) $(CFLAGS) -MM $^ > ./.depend;

include .depend


$(TARGET): $(OBJS)
	$(CC) $(LDFLAGS) $^ -o $@


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

The entire makefile is shown for reference, as it relies on more advanced language semantics. In this case, the `.depend` target makes use of the compiler's -MM flag to build a list of dependencies based on the TARGET file. This list is then stored in a temporary file, .depend, which is included when the target file is built.

To build your own program, simply replace the TARGET variable with the name or your program, and run the makefile from your main file's working directory. Thanks to the shell command, sources and includes may be not just in the current directory alongside the main file, but in subfolders.