COMP1917: Computing 1

11. Writing a Makefile

Introduction

`make` is a utility that is used by programmers to manage the process of recompiling their code.

The actions of the `make` utility is governed by a `Makefile`, which describes:

- the source files that are needed to generate the "target"; and
- how a "target" is to be generated;

When invoked, `make` automatically determines which files need to be recompiled and issues the commands needed to recompile them.

The `make` utility may be used to describe any task where some files must be updated automatically from others whenever the others change.

Building Your First Makefile

Let's assume that you have three C files and two Header files:

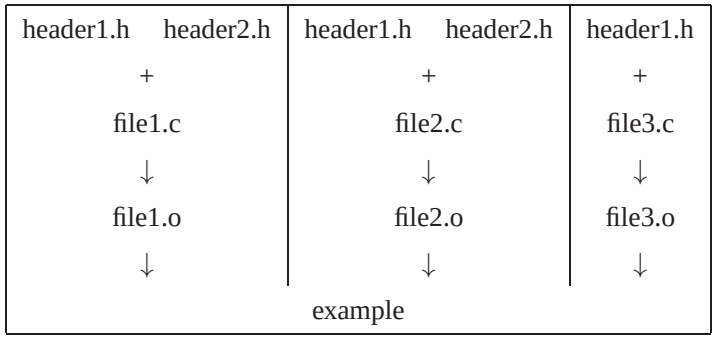
- `file1.c`, `file2.c` and `file3.c`; and
- `header1.h` and `header2.h`

Let's assume that all three C files depend on information that's in `header1.h`, while only `file1.c` and `file2.c` require information that's in `header2.h`

Let's assume that the executable will be called `example`.

Building Your First Makefile *cont.*

Your source file dependency now looks like this:



Building Your First Makefile *cont.*

Create a file called `Makefile` and let's define the goal/target:

```
# our target
example: file1.o file2.o file3.o
```

Things to note:

- Comments start with the "#" character.
- A target (in this case `example`) is on a newline followed by a ":" and followed by the files that the target is dependent on.

Building Your First Makefile *cont.*

Create a file called `Makefile` and let's define the goal/target:

```
# our target
example: file1.o file2.o file3.o
    gcc -o example file1.o file2.o file3.o   # <-- action
```

Things to note:

- The action is always on a newline and must be indented with a TAB.
- The action is executed if there are no additional rules to satisfy for `file1.o`, `file2.o` and `file3.o`

Building Your First Makefile *cont.*

At this point it does not know how to create the ".o" files so let's tell it how:

```
# our target
example: file1.o file2.o file3.o
    gcc -o example file1.o file2.o file3.o   # <-- action

file1.o: file1.c header1.h header2.h
    gcc -Wall -c file1.c

file2.o: file2.c header1.h header2.h
    gcc -Wall -c file2.c

file3.o: file3.c header1.h
    gcc -Wall -c file3.c
```


Building Your First Makefile *cont.*

Things to note:

- The syntax is the same as for the rule "example".
- The action is executed if there are no additional rules to satisfy (i.e. things to the right of the ":").
- The dependencies for each ".o" file is as per table on the previous slide.

Building Your First Makefile *cont.*

With what's defined so far, you've got all the dependencies necessary to build `example`.

`make` is smart enough to work out that `example` is the default target so when you type the command `make` you'll get

```
% make
  gcc -Wall -c file1.c
  gcc -Wall -c file2.c
  gcc -Wall -c file3.c
  gcc -o example file1.o file2.o file3.o
```


Using Variables

You'll notice that there's a lot of repetition in the `Makefile`. Where there is lots of repetition, there's a greater chance of being inconsistent.

It is therefore useful to be able to define a value in one place but reference the definition in lots of places.

You do this by introducing a "variable" and defining its value, for example:

```
CC = gcc
CFLAGS = -Wall
```


Using Variables

You'll see that we've replaced all actual references to `gcc` and the standard "`-Wall`" compilation flag with the variable `$(CC)` and `$(CFLAGS)`

Benefits: now if we decide to change the compiler or compilation flags, we only have to change it in one place, i.e. at the top of the `Makefile`.

The expression:

```
CSRC = file1.c file2.c file3.c
OBJ = $(CSRC:.c=.o)
```

means that `OBJ = file1.o file2.o file3.o` is built by replacing the ".c" suffixes of the files in $(CSRC) with the suffix ".o"

Pattern Rules

To make life even easier, `make` has a number of "Pattern Rules", like this:

```
%o:%c
        $(CC) $(CFLAGS) -c $<
```

This rule simply says that a file `X.o` that's dependent on the file `X.c` will have the stated action applied.

The variable $< references the first prerequisite in the rule, i.e. `X.c`

Pattern Rules *cont.*

With this general pattern rule, it is now possible to remove the individual compile commands from `file1.o`, `file2.o` and `file3.o`, replacing them with this:

```
HSRC = header1.h hearder2.h

%o:%c $(HSRC)
        $(CC) $(CFLAGS) -c $<
```


Adding More Targets

Sometimes you need more than one default target in a project.
For example, you might want to add to the `Makefile` a target that cleans up all the temporary files that a compilation generates.

```
# Additional targets
.PHONY: clean
clean:
        rm -f $(OBJ)
```

The keyword `.PHONY` reminds `make` that `clean` is not the name of a file to be generated by compilation; it is simply a command to instigate some other action. To specify this new target, you say:

```
% make clean
```


Pattern Rules *cont.*

When you have multiple targets in one `Makefile`, you need to make explicit which rule you wish to apply when `make` is run without any rules/targets being specified.

This is done by stating what the “default” rule should be, i.e.:

```
default: example
```


Additional Reading

What I have shown so far is just an overview of `make`, it has far more features than I have mentioned here.

You can get more information by typing:

```
% info make
```

on any of the CSE computers in the labs.