CSSE2310 Week 6 – Makefiles

1 Makefiles

A Makefile is a set of instructions that tell `make` how to build your project. A Makefile should have the filename `Makefile`.

Our focus is on building C projects, but Makefiles can be used for other things. In fact, this document was produced using a Makefile that builds the PDF from a text file.

2 Rules

A Makefile is collection of *rules*. Rules have a *target*, a list of *dependencies*, and a *recipe*.

target the name of the file we want to create,
dependencies files that the target relies on, and
recipe the instructions for making the target (just some shell commands).

If all the dependencies are met, then the recipe is executed to build the target. If the dependencies are not met or are newer than the target, then `make` must build those before it can build the target.

Rules typically look like this

```
target: dependencies
    recipe
```

The first rule in the Makefile is the *default goal*.

When we run

```
$ make
```

it will scan the Makefile and try build the default goal and any dependencies of that goal. We can also specify a particular goal

```
$ make somegoal
```

By convention, the first goal is often called `all`.

Note

- The order of the rules doesn't matter, except that the first rule is the default goal.
- The indentation **must** be TAB characters.
- To save rebuilding the whole project each time we make a small change, each target will only be built if it doesn't exist or any of its dependencies are newer.

See rules for further reading.

3 Phony targets

Usually, `make` expects that each target will generate a file with that name. Sometimes we want *phony* targets that don't generate an output file but still have some useful effect.

```
.PHONY atarget anothertarget someothertarget
```

Often we'll have a `clean` rule that can clean up any junk left over from builds or tests.

For example, the following target will clean up any `.o` files.

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

You should make `clean` a phony target, otherwise if you happen to have a file called `clean`, then `make` might not run your rule since it thinks it's up to date.

See phony targets for further reading.

4 Variables

Just like in `bash`, Makefiles can have variables. You can define your own variables and there are also some pre-existing variables with default values, including:

- `CC`: the compiler, e.g `gcc` or `clang`,
- `CFLAGS`: the compilation flags to give the compiler, and
- `LDFLAGS`: the linker flags to give the linker.

Variables can be set like so

```
CC = gcc
```

and used by wrapping the variable name in `$(var)`, like so

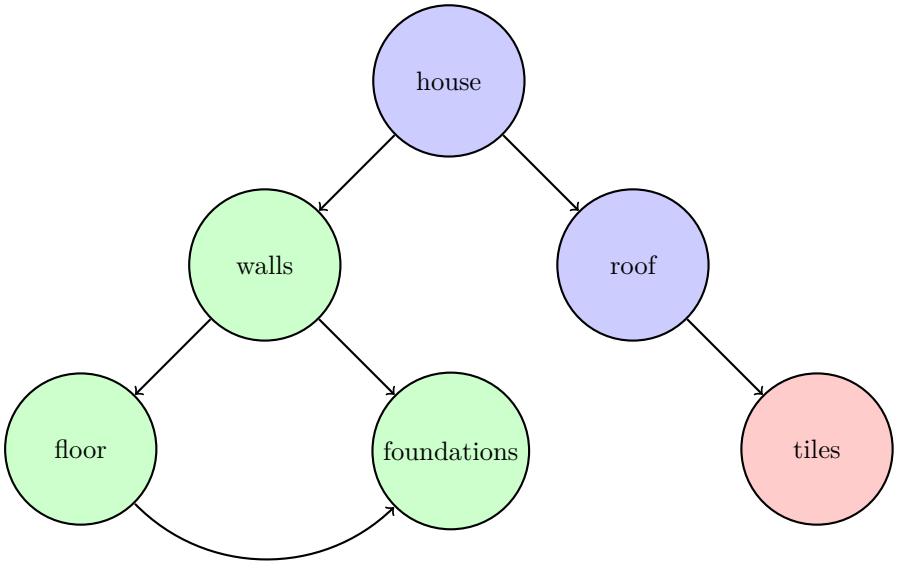
```
    $(CC)
```

See variables for further reading.

5 Non-code example

```
# this is the default goal,
# since it's at the top
house: walls roof
    build house from walls, roof

walls: floor foundations
    build walls on floor, foundations

roof: tiles
    build roof using tiles

floor: foundations
    build floor

foundations:
    build foundations
```

From the Makefile, `make` builds up a directed acyclic graph of dependencies.



For example, running

- `make` will try build `house`, which means it must build all of `house`'s dependencies.
- `make walls` will build `walls`, which means it must build the `floor` and `foundations` first.

Imagine that we make some changes to `tiles` and run `make` to rebuild `house`. The dependencies of `house` are checked and `make` notices that `tiles` (red) is newer than `house`, so to update `house` it first rebuilds `roof` and then rebuilds `house`. It doesn't need to touch the `walls`, the `floor`, or the `foundations`, since they haven't been updated (green).

6 Baby example

```
# default target
boxes: boxes.c
    gcc -g -Wall -pedantic -ansi boxes.c -o boxes
```

Now we can just type `make` to build `boxes`.

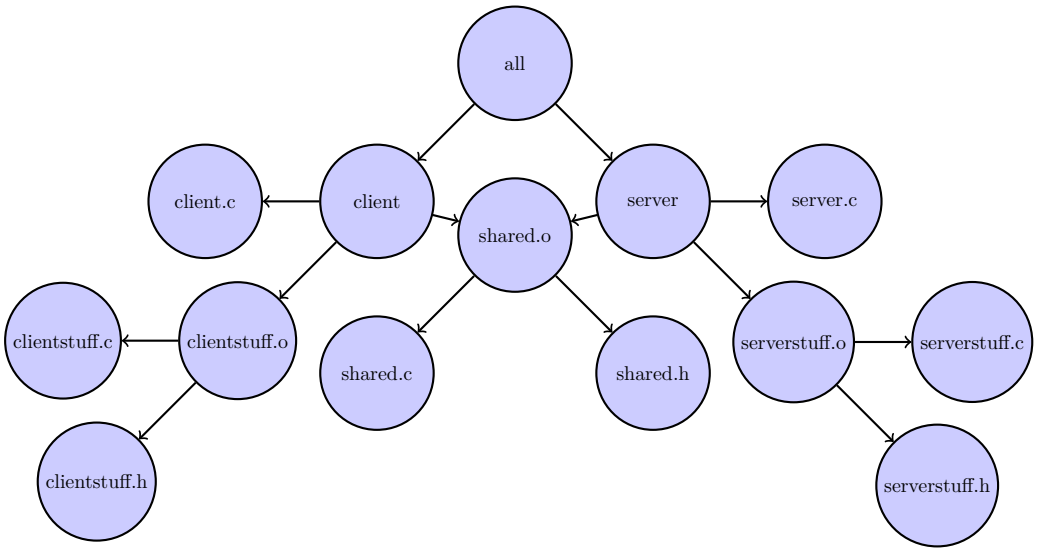
7 Larger project example

As we start adding more rules, using variables saves us having to copy-paste any changes we make in the future.

```
CC = gcc
CFLAGS = -g -Wall -pedantic -std=c99 -Werror

# neither all nor clean produce
# files with those names
.PHONY all clean

# default target
all: client server

# creates the client executable
```

```
client: client.c shared.o clientstuff.o
    $(CC) $(CFLAGS) shared.o clientstuff.o -o client

# creates the server executable
server: server.c shared.o serverstuff.o
    $(CC) $(CFLAGS) shared.o serverstuff.o -o server

shared.o: shared.c shared.h
    $(CC) $(CFLAGS) -c shared.c

clientstuff.o: clientstuff.c clientstuff.h
    $(CC) $(CFLAGS) -c clientstuff.c

serverstuff.o: serverstuff.c serverstuff.h
    $(CC) $(CFLAGS) -c serverstuff.c

clean:
    rm *.o
```

Note: notice that the header files (`.h`) only appear as dependencies, they are never included in the recipe.



8 (Advanced) Automatic variables

There are some special variables that can help us

- `$@`: the file name of the target
- `$<`: the name of the first prerequisite
- `$^`: the name of all the prerequisites

See automatic variables for further reading.