Make

1 What is make?

When your code is separated into many several files that have interdependencies, it is natural to look for an automated way of compiling your program, because compilation and linking become very complicated. The `make` program was designed to help developers automatically build executable programs. Once you have set up a `Makefile` that describes the components of your program, their relationships, and how to build the various components, then `make` will do the build for you correctly every time.

2 Why use make?

There are two main reasons to use `make`.

Simpler compilation and linking

`make` allows a shorthand for complicated commands or a large sequence of commands.

Explicitly typing `clang++ -Wall -Wextra -std=c++11 -o foo bar.cpp baz.cpp` for each compilation quickly becomes cumbersome. When you have a Makefile properly configured, the `foo` executable could instead be compiled by typing `make foo`.

Faster re-compilation

Another advantage of `make` is that it only recompiles code that needs to be recompiled. This is particularly useful in large programs with 300+ files. If you change only one of then, you need not recompile everything: make compiles the files you changed and anything that depends on it.

Compiling `C++` code is a multistep process

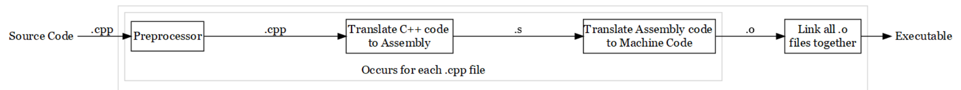
Source Code
.cpp
Preprocessor
.cpp
Translate C++ code to Assembly
.s
Translate Assembly code to Machine Code
.o
Link all .o files together
Executable
Occurs for each .cpp file

Figure 1: Phases of Compilation

Running the command `clang++ -o foo bar.cpp baz.cpp` will follow the steps in `Figure 1` to make `bar.o baz.o`, and will combine the `.o` files into an executable named `foo`. By default, most compilers delete the intermediate files (`.cpp .s .o`) after creating the executable. Which means that every `.cpp` file must always recompiled. If a `.cpp` file hasn't been changed since the last compilation, recompiling it is wasteful since its previous `.o` file could be reused. As projects get larger, recompiling all sources files for each compilation can become very slow.

`make` automatically tracks which components of an executable have changed since the last time the executable was compiled. We typically write `Makefiles` to save `.o` files. That way, compiling an executable only requires remaking `.o` files that are not up to date, and combining them with the old `.o` files.

Further Reading

This document gives information on `make`, but not on the compilation itself. If you want to know more about compilation we suggest you look at the following links.

- http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html
- http://www.cs.ecu.edu/karl/3300/spr14/Notes/C/preprocessor.html

3 Anatomy of a Makefile

Makefile rules

`Makefile rules` are the building blocks of `Makefiles`. They describe an action to perform when the `rule` is evoked, and also when to perform that action.

```
target: prerequisites...
        recipe
```

A `Makefile rule` is made up of

- A `target`, which is the argument given to `make` on the commandline (i.e `make target`). The `target` usually is the name of the file that will be created after the `rule` is executed, but it can also be the name of an action to carry out (`clean`, `install`, etc…).
- A list of 0 or more `prerequisites`, which are files or `targets` that the current `target` depends on.
- A `recipe`, which is the shell command(s) executed by the `Makefile rule`.

Make will execute the `recipe` if a file named `target` does not exist, or if any of the `prerequisites` are more recent than `target`.

Example

```
bar.o: bar.cpp bar_header.h
        clang++ -c bar.cpp
```

- `bar.o` is the `target`
- `bar.cpp bar_header.h` are the `prerequisites`
- `clang++ -c bar.cpp` is the `recipe`

Important: When compiling code, we only compile `.cpp` files, not `.h` files

When you type `make bar.o` into the console, the first thing that `make` will do is check the timestamp of `bar.o` and all of the files it depends (`bar.cpp` and `bar_header.h` in this example). Then it will run the recipe (in this case `clang++ -c bar.cpp`) if `bar.o` doesn't exist, or if either `bar.o` or `bar_header.h` is more recent than `bar.o`.

Makefile variables

`Variables` can be defined in `Makefile` to make them more configurable

`Variables` are declared as:

```
varName = value
```

The value of a `variable` can then be accessed by wrapping the `variable` in `${}` or `$()`:

```
${varName} or $(varName)
```

`Makefile variables` can be used to specify the compiler, and any compilation settings.

- `CXX` is the name of the C++ compiler
- `CXXFLAGS` are the compiler options for compiling `.cpp` files into `.o` files

(It's possible to also see `CC` and `CFLAGS` which are the C equivalents for `CXX` and `CXXFLAGS`)

4 Guide: Writing a Makefile to compile C++

Consider the following project, that is compiled into the executable `foo`. *`.h` filenames were chosen to indicate which `.cpp` file includes it*:

```
vm-hw09{student01}51: ls
bar.cpp  baz.cpp  qux.cpp baz_header.h qux_and_baz_header.h
```

1. Set up the `Makefile variables`

```
CXX      = clang++
CXXFLAGS = -std=c++11 -Wall -Wextra
```

2. Write a `Makefile rule` to compile each `.cpp` file into a `.o` file. The `prerequisites` should list the `.cpp` file and the `.h` files it includes. The `recipe` should include the `.cpp` file and the `-c` flag instructing the compiler to output a `.o` file. **`Makefile recipes` must be preceded by tabs (not spaces).**

```
CXX      = clang++
CXXFLAGS = -std=c++11 -Wall -Wextra

bar.o: bar.cpp
	${CXX} ${CXXFLAGS} -c bar.cpp

baz.o: baz.cpp baz_header.h baz_and_qux_header.h
	${CXX} ${CXXFLAGS} -c  baz.cpp

qux.o: qux.cpp baz_and_qux_header.h
	${CXX} ${CXXFLAGS} -c qux.cpp
```

3. Write a `Makefile rule` to link `.o` files into an executable. The `prerequisites` contains all `.o` files needed. The `recipe` compiles all `.o` listed in the `prerequisites`, and `-o name_of_target` instructing the compiler to name the executable `name_of_target`. Placing it before all other `rules` makes it the `default rule`, so it can also be invoked by simply typing `make`

```
CXX      = clang++
CXXFLAGS = -std=c++11 -Wall -Wextra

foo: bar.o baz.o qux.o
	${CXX} -o foo bar.o baz.o qux.o
```

```
bar.o: bar.cpp
        ${CXX} ${CXXFLAGS} -c bar.cpp

baz.o: baz.cpp baz_header.h baz_and_qux_header.h
        ${CXX} ${CXXFLAGS} -c  baz.cpp

qux.o: qux.cpp baz_and_qux_header.h
        ${CXX} ${CXXFLAGS} -c qux.cpp
```

4. Pat yourself on the back for a job well done.

5 Going further

This section is not required to understand or write `Makefiles` in COMP15, but instead introduces some of the more powerful aspects of `make`

Phony targets

`Makefile targets` aren't required to be files. It's possible to define `rules` that don't create files, but are instead used as shorthand for longer commands. These are called `phony targets`.

Example

A common `phony target` is `clean`, which deletes all output from build commands.

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

Automatic Variables

`Make` automatically defines some variables that can be used in `makefile` recipes, including:

`$@`: The file name of the `target` of the `rule`.

`$<`: The name of the first `prerequisite`.

`$^`: The names of all the `prerequisites`, with spaces between them.

Example

```
foo: bar.o baz.o qux.o
        ${CXX} -o foo bar.o baz.o qux.o
```

can be rewritten as

```
foo: bar.o baz.o qux.o
        ${CXX} -o $@ $^
```

Pattern rules

Many `Makefile` rules follow a similar template, and explicitly defining each one adds unneccesary bloat to the `Makefile`. `Pattern rules` allow `Makefile` rules to be expressed more generally and concisely:

Example

```
%.o: %.cpp
        ${CXX} ${CXXFLAGS} -c $<
```

`%` is a wildcard that matches against `target` names. The `%.o` `target` will be selected if ever the requested `target` ends with `.o`

This rule says: "To make any `.o` file, use the following `recipe` to compile its corresponding `.cpp` into a `.o` file."

We would also want to add the `.h` files to the prerequisites, but that is a bit more work (see Shell Function)

Shell Function

`Make` also has some in-built functions. Using the `shell` function, the output of shell commands can be used within a `Makefile`. The syntax `$(shell COMMAND_TO_RUN ARGS)` is used to invoke the `shell` function.

Example

```
%.o: %.cpp
        ${CXX} ${CXXFLAGS} -c $<
```

If we have a `pattern rule` for `.o` files, it would be useful to state that the `.o` file also depends any `.h` files included in the `.cpp`. But since the pattern rule can apply to *any* `.cpp` file, there isn't an easy way to specify the correct headers for each file.

One solution, for small projects, would be to say that the `.o` files depend on all `.h` files. The `shell` function can be used to run `echo *.h` on the commandline, getting the names of all `.h` files in the current directory:

```
INCLUDES = $(shell echo *.h)
```

```
%.o: %.cpp ${INCLUDES}
        ${CXX} ${CXXFLAGS} -c $<
```

Putting it all together

We can rewrite the `Makefile` from the guide using these new techniques

```
CXX      = clang++
CXXFLAGS = -std=c++11 -Wall -Wextra
# Shell Function
INCLUDES = $(shell echo *.h)

foo: bar.o baz.o qux.o
        ${CXX} -o $@ $^

# Pattern rule for .o files
%.o: %.cpp ${INCLUDES}
        ${CXX} ${CXXFLAGS} -c $<

# Phony Target
clean:
        rm -f foo *.o
```