

Introduction to Programming Advanced 2020 S2

Dr. John Stavrakakis

School of Computer Science, University of Sydney



COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Makefiles

Language agnostic build process

Build an executable binary file

There is a program that we wish to build.

The program when produced is an *executable binary file*.

It containing instructions of what the CPU will execute.

To make the (often) single binary file, multiple source code files are needed to aggregate together

We need a build system

Multiple sources single target

We can have many .c and .h files for our C programming project.

Here is a sample project:

```
simulation_main.c  
simulation.c  
mymath.c  
mymath.h  
mytimer.c  
mytimer.h
```

The project is compiled as follows:

```
gcc -o simulation simulation_main.c simulation.c mymath.c mytimer.c
```

Notice that only one target. Multiple dependencies

Recompilation costs

If we change any file, we have to recompile all the code.

`simulation_main.c` → recompile

change `simulation.c` → recompile

change `simulation.h` → recompile

change `mymath.c` → recompile

change `mytimer.c` → recompile

change `mymath.h` → recompile

change `mytimer.h` → recompile

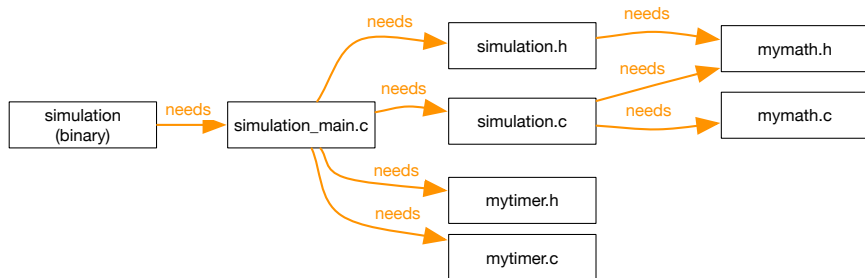
Code can be huge. Unnecessary compilation is unnecessary time wasted.

file	lines of code
<code>simulation_main.c</code>	~10,000
<code>simulation.c</code>	~170,000
<code>simulation.h</code>	~150
<code>mymath.c</code>	~65,000
<code>mymath.h</code>	~800
<code>mytimer.c</code>	~200
<code>mytimer.h</code>	~30

Understanding dependencies

Increasing complex software demands greater modularity. By separating the responsibilities into modules, components or frameworks. We can better understand the roles and interactions between them.

Hence the build process would have an explicit dependency mapping.

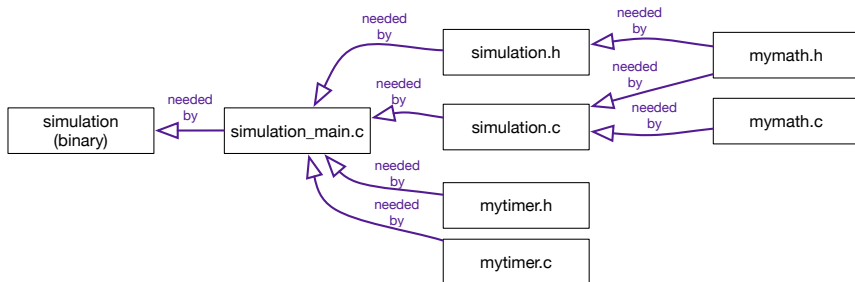
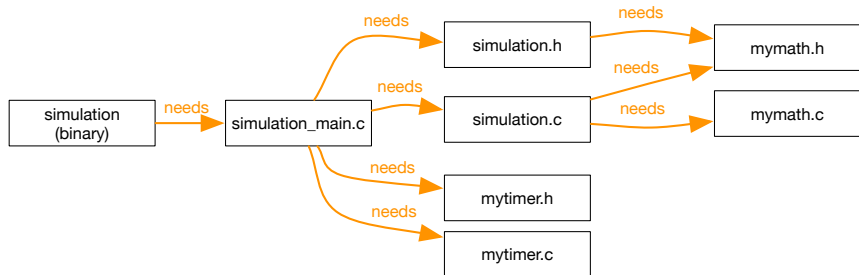


`simulation_main.c` depends on everything. We cannot escape recompiling for that case. But what about the others?

If I change `mytimer.c` should I recompile all the other code?

Let's invert the dependency hierarchy. This allows us to determine which files should be recompiled if they change.

Understanding dependencies (cont.)



Programming language compilers allow you to produce binary code that is incomplete.

Within a module for instance. The functions themselves, how the memory is arranged within the function, local variables, variables local to the module. function prototypes, structure definitions etc.

This compilation of parts internal to a module is independent to outside modules. It should not have to be recompiled each time.

Using the intermediate formats

C/C++/Rust has .o files

Java/Kotlin has .class files

Python has .pyc files

Many languages adopt package like systems for large projects

The compiler accepts different flags as well as filenames of binary code and/or source code.

This allows us to compile different parts of our program with different switches enabled

Compile each module independently

```
gcc -c simulation_main.c
gcc -c simulation.c
gcc -c mymath.c
gcc -c mytimer.c
```

But what about the .h files? Suppose that `simulation_main.c` uses this function prototype in `simulation.h`:

```
int simulate(void *state, double timestep_delta);
```

and it was changed to:

```
int simulate(void *state, double timestep_delta, int
*converged);
```

Need to recompile all source code files that depend on `simulation.h`

This is not captured in the process.

Makefiles can be used to define dependency rules and actions to take

the *make* program automates the process of recompiling. It will automatically recompile all the source code files that have been changed as well as any code dependent on the changes.

make interprets a set of rules specified in the Makefile as the dependencies.

When *make* is run, it will check the last change timestamps on the dependencies specified in the rules and if dependencies have been modified, it will invoke the appropriate action of recompilation.

Makefiles (cont.)

In file called Makefile:

```
simulation.o: simulation.c simulation.h
```

```
    gcc -c simulation.c
```

```
    echo "hello!"
```

```
simulation_main.o: simulation_main.c simulation.h mytmer.h
```

```
    gcc -c simulation_main.c
```

Three main ingredients for a rule to be defined. Target, Dependencies, Actions

target: the name of the file you want to make

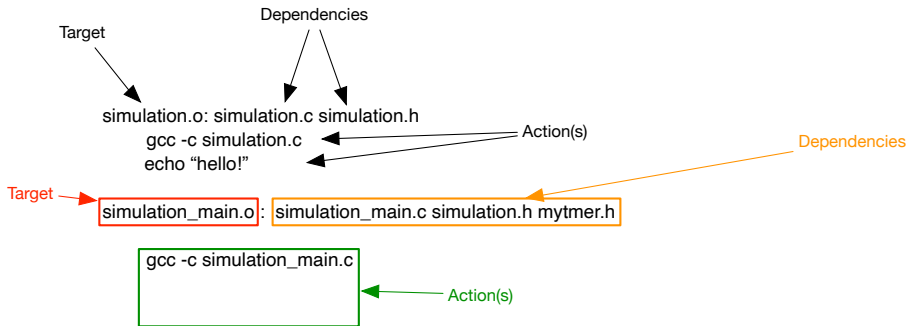
one or more *dependencies*: files the target depends on

an *action*: a shell command that creates the target

Makefiles (cont.)

Two rules defined here. One target describes the actual file that we require to produce, it's dependencies, and actions needed to build it.

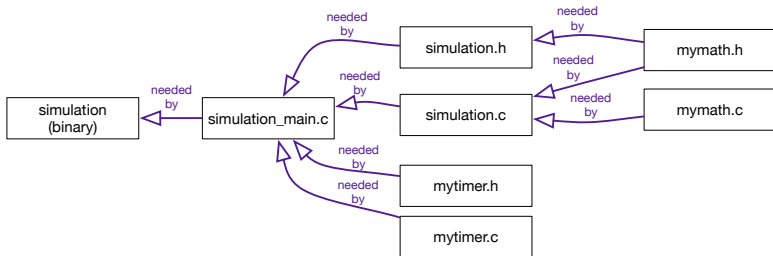
We can make any action we like after a rule has been triggered. Notice echo "hello!"



Type

```
$ make
```


Dependency graph revisited



Notice we have a dependency relationship with object files and header files of other modules, **not** their .c files directly

Revise our rules based on object files rather than source code files

Full Makefile

```
1  # our compiler
2  CC=gcc
3
4  # flags that apply for any compilation
5  CFLAGS=-Werror
6
7  # flags that apply for final linking stage
8  LDFLAGS=-lm
9
10 simulation: simulation_main.o simulation.o mytimer.o mymath.o
11     $(CC) $(LDFLAGS) -o simulation simulation_main.o simulation.o mytimer.o mymath.o
12
13 simulation_main.o: simulation_main.c simulation.h mytimer.h
14     $(CC) $(CFLAGS) -c simulation_main.c
15
16 simulation.o: simulation.c simulation.h
17     $(CC) $(CFLAGS)-c simulation.c
18
19 mymath.o:    mymath.c mymath.h
20     $(CC) $(CFLAGS) -c mymath.c
21
22 mytimer.o:   mytimer.c mytimer.h
23     $(CC) $(CFLAGS) -c mytimer.c
24
25 clean:
26     echo "removing all object files"
27     rm simulation_main.o simulation.o mytimer.o mymath.o
```

Excellent software that is versatile across languages and even purposes^[1]

You can create any dependencies and rules you need. Software can be built to different targets:

```
make i686
```

```
make test
```

```
make release
```

Plenty of macro expansions, makefile hierarchies, pattern matching rules (supremely useful!).

More information about make:

<https://www.gnu.org/software/make/manual/make.html>

^[1]These L^AT_EX slides were generated using a Makefile!