

Programming Techniques for Scientific Simulations I

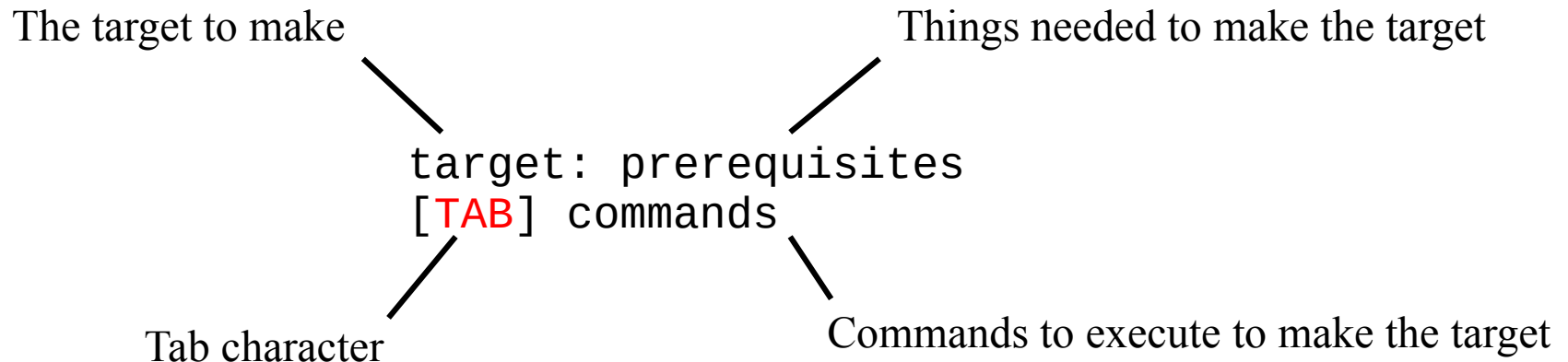
Make

Make

- Make is a build automation tool
- It build targets (e.g. executables, libraries, reports, ...) following rules (e.g. the commands to compile the sources) and manages the dependencies. It keeps track of updates!
- Created by Stuart Feldman in the mid seventies while he was a student intern at Bell Labs (where UNIX was developed)
 - ♦ Feldman, S. I., “Make - a program for maintaining computer programs”, 1979, <https://doi.org/10.1002/spe.4380090402>
- (Probably) most widely used build tool!
- GNU Make is the standard implementation of Make on Linux and macOS... And we will use that!
- Make is a little language
 - ♦ Relatively cryptic
 - ♦ No debugger
 - ♦ Requires an understanding of the *nix command line & shell
 - ♦ Uses some conventions...

Basics

- Make builds targets based on so-called build files
 - ◆ Makefile, makefile, GNUmakefile
- The makefiles builds targets based on rules




Basics

- Example:

```
# comment  
hello:  
    echo Hello students
```

Comment in makefiles



- One target (hello) with no dependencies and one command to execute

- Running make:

\$ make

Will look for default makefile names

\$ make -f hello.mk

Use hello.mk

\$ make -n

Dry run... Useful to see issued commands

A simple example

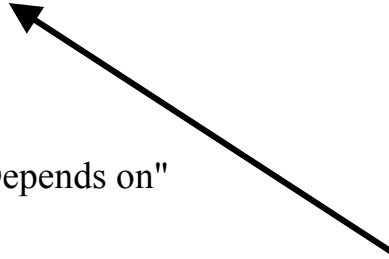
- Consider again the simple square program

```
// square.hpp  
double square(double);
```

```
// square.cpp  
#include "square.hpp"  
double square(double x) {  
    return x*x;  
}
```

```
// main.cpp  
#include <iostream>  
#include "square.hpp"  
int main() {  
    std::cout << square(5.)  
               << std::endl;  
    return 0;  
}
```

"Depends on"



A simple example

- By hand we would build the executable as

```
$ c++ -c square.cpp
$ c++ -c main.cpp
$ c++ -o square.exe main.o square.o
```

- Or with the following makefile

```
# simple_try-01.mk
square.o: square.cpp square.hpp
    c++ -c square.cpp

main.o: main.cpp square.hpp
    c++ -c main.cpp

square: main.o square.o
    c++ -o square main.o square.o
```

- Let's build:

```
$ make -f simple_try-01.mk
```

A simple example

- By hand we would build the executable as

```
$ c++ -c square.cpp
$ c++ -c main.cpp
$ c++ -o square.exe main.o square.o
```

- Or with the following makefile

```
# simple_try-01.mk
square.o: square.cpp square.hpp
    c++ -c square.cpp

main.o: main.cpp square.hpp
    c++ -c main.cpp

square: main.o square.o
    c++ -o square main.o square.o
```

- Builds only square.o! Make builds only the first target by default!

```
$ make -f simple_try-01.mk square — Target to build
```

A simple example

- Introduce a so-called phony target (a target that is not a file)

```
# simple.mk
.PHONY: all
all: square

square.o: square.cpp square.hpp
        c++ -c square.cpp

main.o: main.cpp square.hpp
        c++ -c main.cpp

square: main.o square.o
        c++ -o square main.o square.o
```

- Now the first target is all, which depends on square

```
$ make -f simple.mk
```


A simple example

- Happy?

```
# simple.mk
.PHONY: all
all: square

square.o: square.cpp square.hpp
      c++ -c square.cpp

main.o: main.cpp square.hpp
      c++ -c main.cpp

square: main.o square.o
      c++ -o square main.o square.o
```

- Some concerns:
 - ♦ Repeated names, i.e. it's WET (Write Every Time)... Want it DRY (Don't Repeat Yourself)
 - ♦ What if another compiler is needed?
 - ♦ What if additional compiler flags are needed?
 - ♦ Clean the mess?

A simple example

- Make has so-called automatic variables, which are set after a rule is matched
 - ◆ `$@`: The file name of the target of the rule.
 - ◆ `$<`: The name of the first prerequisite.
 - ◆ `$^`: The names of all the prerequisites, with spaces between them.
- Many more... See the documentation!

```
.PHONY: all
all: square

square.o: square.cpp square.hpp
        c++ -c square.cpp

main.o: main.cpp square.hpp
        c++ -c main.cpp

square: main.o square.o
        c++ -o square main.o square.o
```

A simple example

- Make has so-called automatic variables, which are set after a rule is matched
 - ◆ `$@`: The file name of the target of the rule.
 - ◆ `$<`: The name of the first prerequisite.
 - ◆ `$^`: The names of all the prerequisites, with spaces between them.
- Many more... See the documentation!

```
.PHONY: all
all: square

square.o: square.cpp square.hpp
        c++ -c $<

main.o: main.cpp square.hpp
        c++ -c $<

square: main.o square.o
        c++ -o $@ $^
```

A simple example

- Make has some predefined variables
 - ♦ CC: Program for compiling C programs; default 'cc'
 - ♦ CXX: Program for compiling C++ programs; default 'g++'.
 - ♦ RM: Command to remove a file; default 'rm -f'.
 - ♦ CFLAGS: Extra flags to give to the C compiler.
 - ♦ CXXFLAGS: Extra flags to give to the C++ compiler.
 - ♦ LDFLAGS: Extra flags to give to compilers when linking
 - ♦ LDLIBS: Library flags or names given to compilers when linking
- Many more... See the documentation!

```
.PHONY: all
all: square

square.o: square.cpp square.hpp
        c++ -c $<

main.o: main.cpp square.hpp
        c++ -c $<

square: main.o square.o
        c++ -o $@ $^
```

A simple example

- Make has some predefined variables
 - ♦ CC: Program for compiling C programs; default 'cc'
 - ♦ CXX: Program for compiling C++ programs; default 'g++'.
 - ♦ RM: Command to remove a file; default 'rm -f'.
 - ♦ CFLAGS: Extra flags to give to the C compiler.
 - ♦ CXXFLAGS: Extra flags to give to the C++ compiler.
 - ♦ LDFLAGS: Extra flags to give to compilers when linking
 - ♦ LDLIBS: Library flags or names given to compilers when linking
- Many more... See the documentation!


```
.PHONY: all
all: square

square.o: square.cpp square.hpp
    ${CXX} ${CXXFLAGS} -c $<

main.o: main.cpp square.hpp
    ${CXX} ${CXXFLAGS} -c $<

square: main.o square.o
    ${CXX} ${CXXFLAGS} -o $@ $^
```

Mind the braces {}!
Parentheses () also possible.



A simple example

■ Cleaning?

```
.PHONY: all
all: square

square.o: square.cpp square.hpp
    ${CXX} ${CXXFLAGS} -c $<

main.o: main.cpp square.hpp
    ${CXX} ${CXXFLAGS} -c $<

square: main.o square.o
    ${CXX} ${CXXFLAGS} -o $@ $^

.PHONY: clean
clean:
    ${RM} -v *.o square
```



A simple example

■ Using a different compiler?

```
CXX      = c++
CXXFLAGS = -std=c++11
CXXFLAGS += -Wall -Wextra -Wpedantic
CXXFLAGS += -O3
```



Set variables

Appending

```
.PHONY: all
all: square
```

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

```
main.o: main.cpp square.hpp
    ${CXX} ${CXXFLAGS} -c $<
```

```
square: main.o square.o
    ${CXX} ${CXXFLAGS} -o $@ $^
```

```
.PHONY: clean
clean:
```

```
    ${RM} -v *.o square
```

A simple example

- Suppose your makefile is under version control and your collaborators need different compilers & flags... Then the makefile always changes depending on who committed!

```
include config.mk
```

```
.PHONY: all  
all: square
```

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

```
main.o: main.cpp square.hpp  
    ${CXX} ${CXXFLAGS} -c $<
```

```
square: main.o square.o  
    ${CXX} ${CXXFLAGS} -o $@ $^
```

```
.PHONY: clean  
clean:  
    ${RM} -v *.o square
```

```
# config.mk
```

```
CXX      = g++-mp-9
```

```
CXXFLAGS = -std=c++11
```

```
CXXFLAGS += -Wall -Wextra -Wpedantic
```

```
CXXFLAGS += -O3
```

Not under version control!

A simple example

- Or even slightly better:

← Ignores if config.mk does not exist... So the default variable values are used!

```
-include config.mk
```

←

```
.PHONY: all
all: square

square.o: square.cpp square.hpp
    ${CXX} ${CXXFLAGS} -c $<

main.o: main.cpp square.hpp
    ${CXX} ${CXXFLAGS} -c $<

square: main.o square.o
    ${CXX} ${CXXFLAGS} -o $@ $^

.PHONY: clean
clean:
    ${RM} -v *.o square
```

config.mk
(Symbolic link to) personal config file
based on example config!

↑ Not under version control!

Under version control!

```
# config.mk.example
CXX      = c++
CXXFLAGS = -std=c++11
CXXFLAGS += -Wall -Wextra -Wpedantic
CXXFLAGS += -O3
```

A simple example

- Happy?

```
-include config.mk

.PHONY: all
all: square


square.o: square.cpp square.hpp
    ${CXX} ${CXXFLAGS} -c $<

main.o: main.cpp square.hpp
    ${CXX} ${CXXFLAGS} -c $<

square: main.o square.o
    ${CXX} ${CXXFLAGS} -o $@ $^

.PHONY: clean
clean:
    ${RM} -v *.o square
```

Automatic dependency generation...
See e.g. online
(or "[Managing Projects with Make](#)"
by Mecklenburg)



Make

- For further information see the documentation
 - ♦ <https://www.gnu.org/software/make/>
 - ♦ On the command line: `$ man make # get help by typing h, quit with q`
`$ info make # get help by typing h or H, quit with q`
- There are excellent tutorials for special purposes on “the Internets”

Usually, one never starts from scratch... Use an example!

- Make is not only useful for building software...
 - ♦ Example of “building” a paper... Excellent for reproducibility! Traces your workflow
 - <http://swcarpentry.github.io/make-novice/>