ECS 150 - Makefile tutorial

Prof. Joël Porquet-Lupine

UC Davis - 2020/2021

UC DAVIS

COMPUTER SCIENCE

Manual approach

Code example

main.c

```c
#include <stdio.h>
#include <stdlib.h>

#include "fact.h"

int main(int argc, char **argv)
{
    int n;
    if (argc < 2) {
        fprintf(stderr,
                "Usage: myfact number\n");
        exit(1);
    }
    n = atoi(argv[1]);
    printf("fact(%d) = %d\n",
           n, fact(n));
    return 0;
}
```

main.c

fact.h

```c
#ifndef FACT_H_
#define FACT_H_

int fact(int n);

#endif /* FACT_H_ */
```

fact.h

fact.c

```c
#include "fact.h"

int fact(int n)
{
    if (n == 0)
        return 1;
    return n * fact(n - 1);
}
```

fact.c

README.md

```
# Overview

This program computes the
factorial of a number
```

README.md

Manual approach

Compilation

```
$ gcc -Wall -Wextra -Werror -c -o fact.o fact.c
$ gcc -Wall -Wextra -Werror -c -o main.o main.c
$ gcc -Wall -Wextra -Werror -o myfact main.o fact.o
```

```
$ ./myfact
Usage: myfact number
$ ./myfact 5
fact(5) = 120
```



```
$ pandoc -o README.html README.md
$ firefox README.html
```


On the long run...

Now, what if:

- fact.c changes? main.c changes? fact.h changes?
- I want to change the compilation options?
- I want to recompile this code on another computer?
- I want to share this code?

Solution is to **automate the build process**!

Introduction

Definition

A *Makefile* is a file containing a set of rules used with the *make* build automation tool.

The two following commands are equivalent:

```
$ ls
Makefile ...
$ make
$ make -f Makefile
```


The set of Makefile rules usually represents the various steps to follow in order to build a program: it's the building *recipe*.

Introduction

Anatomy of a rule

```
target: [list of prerequisites]
[ <tab> command ]
```

- For `target` to be generated, the prerequisites must all exists (or be generated if necessary)
- `target` is generated by executing the specified `command`
- `target` is generated only if it does not exist, or if one of the prerequisites is more recent
  - Prevents from building everything each time, but only what is necessary

Commenting

- Lines prefixed with # are not evaluated

```
# This is a comment
```

Version 0.1

Basic rules

```
myfact: main.o fact.o
    gcc -Wall -Wextra -Werror -o myfact main.o fact.o

main.o: main.c fact.h
    gcc -Wall -Wextra -Werror -c -o main.o main.c

fact.o: fact.c fact.h
    gcc -Wall -Wextra -Werror -c -o fact.o fact.c

README.html: README.md
    pandoc -o README.html README.md
```

Makefile_v0.1


```
$ make
gcc -c -o main.o main.c
gcc -c -o fact.o fact.c
gcc -o myfact main.o fact.o
```

```
$ make README.html
pandoc -o README.html README.md
```

Version 0.1

`all` rule

```
all: myfact README.html

myfact: main.o fact.o
    gcc -Wall -Wextra -Werror -o myfact main.o fact.o

main.o: main.c fact.h
    gcc -Wall -Wextra -Werror -c -o main.o main.c

fact.o: fact.c fact.h
    gcc -Wall -Wextra -Werror -c -o fact.o fact.c

README.html: README.md
    pandoc -o README.html README.md
```

Makefile_v0.1


```
$ make
gcc -c -o main.o main.c
gcc -c -o fact.o fact.c
gcc -o myfact main.o fact.o
pandoc -o README.html README.md
```

Version 0.1

`clean` rule

```
all: myfact README.html
...
clean:
    rm -f myfact README.html main.o fact.o
```

Makefile_v0.1

```
$ make
gcc -c -o main.o main.c
gcc -c -o fact.o fact.c
gcc -o myfact main.o fact.o
pandoc -o README.html README.md
```

```
$ make clean
rm -f myfact README.html main.o fact.o
```

Version 0.1

A first and basic Makefile

```
all: myfact README.html

myfact: main.o fact.o
    gcc -Wall -Wextra -Werror -o myfact main.o fact.o

main.o: main.c fact.h
    gcc -Wall -Wextra -Werror -c -o main.o main.c

fact.o: fact.c fact.h
    gcc -Wall -Wextra -Werror -c -o fact.o fact.c

README.html: README.md
    pandoc -o README.html README.md

clean:
    rm -f myfact README.html main.o fact.o
```

Makefile_v0.1

- Was good enough for Project #1
  - *(No need to generate html out of markdown --pandoc is not installed on CSIF, and also it's just for the example)*

Version 1.0

How to avoid redundancy...?

A good programmer is a lazy programmer!

```
all: myfact README.html

myfact: main.o fact.o
    gcc -Wall -Wextra -Werror -o myfact main.o fact.o

main.o: main.c fact.h
    gcc -Wall -Wextra -Werror -c -o main.o main.c

fact.o: fact.c fact.h
    gcc -Wall -Wextra -Werror -c -o fact.o fact.c

README.html: README.md
    pandoc -o README.html README.md

clean:
    rm -f myfact README.html main.o fact.o
```

Makefile_v0.1

Version 1.0

Automatic variables in commands

- `$@`: replaced by name of target
- `$<`: replaced by name of **first** prerequisite
- `$^`: replaced by names of **all** prerequisites

```
all: myfact README.html

myfact: main.o fact.o
    gcc -Wall -Wextra -Werror -o $@ $^

main.o: main.c fact.h
    gcc -Wall -Wextra -Werror -c -o $@ $<

fact.o: fact.c fact.h
    gcc -Wall -Wextra -Werror -c -o $@ $<

README.html: README.md
    pandoc -o $@ $<

clean:
    rm -f myfact README.html main.o fact.o
```

Version 1.0

Pattern rules

A pattern rule `%.o: %.c` says how to generate *any* file `<file>.o` from another file `<file>.c`.

```
all: myfact README.html

myfact: main.o fact.o
    gcc -Wall -Wextra -Werror -o $@ $^

%.o: %.c fact.h
    gcc -Wall -Wextra -Werror -c -o $@ $<

%.html: %.md
    pandoc -o $@ $<

clean:
    rm -f myfact README.html main.o fact.o
```

Version 1.0

Variables

```
CC      := gcc
CFLAGS  := -Wall -Wextra -Werror
CFLAGS  += -g
PANDOC  := pandoc

all: myfact README.html

myfact: main.o fact.o
    $(CC) $(CFLAGS) -o $@ $^

%.o: %.c fact.h
    $(CC) $(CFLAGS) -c -o $@ $<

%.html: %.md
    $(PANDOC) -o $@ $<

clean:
    rm -f myfact README.html \
          main.o fact.o
```

```
$ make
gcc -Wall -Wextra -Werror -g -c -o main.o main.c
gcc -Wall -Wextra -Werror -g -c -o fact.o fact.c
gcc -Wall -Wextra -Werror -g -o myfact main.o fact.o
pandoc -o README.html README.md
```

Version 2.0

More variables

```
targets := myfact README.html
objs    := main.o fact.o

CC      := gcc
CFLAGS  := -Wall -Wextra -Werror
CFLAGS  += -g
PANDOC  := pandoc

all: $(targets)

myfact: $(objs)
    $(CC) $(CFLAGS) -o $@ $^

%.o: %.c fact.h
    $(CC) $(CFLAGS) -c -o $@ $<

%.html: %.md
    $(PANDOC) -o $@ $<

clean:
    rm -f $(targets) $(objs)
```

```
$ make
gcc -Wall -Wextra -Werror -g -c -o main.o main.c
gcc -Wall -Wextra -Werror -g -c -o fact.o fact.c
gcc -Wall -Wextra -Werror -g -o myfact main.o fact.o
pandoc -o README.html README.md
```

Version 2.0

Nice output

```
$ make
CC main.o
CC fact.o
CC myfact
MD README.html
```

```
$ make clean
CLEAN
```

```
...

myfact: $(objs)
    @echo "CC   $@"
    @$(CC) $(CFLAGS) -o $@ $^

%.o: %.c fact.h
    @echo "CC   $@"
    @$(CC) $(CFLAGS) -c -o $@ $<

%.html: %.md
    @echo "MD   $@"
    @$(PANDOC) -o $@ $<

clean:
    @echo "CLEAN"
    @rm -f $(targets) $(objs)
```


- In case of debug, how can we still see the commands that are executed?

Version 3.0

Conditional variables

```
...

ifneq ($(V),1)
Q = @
endif

myfact: $(objs)
    @echo "CC   $@"
    $(Q)$(CC) $(CFLAGS) -o $@ $^

%.o: %.c fact.h
    @echo "CC   $@"
    $(Q)$(CC) $(CFLAGS) -c -o $@ $<

%.html: %.md
    @echo "MD   $@"
    $(Q)$(PANDOC) -o $@ $<

clean:
    @echo "CLEAN"
    $(Q)rm -f $(targets) $(objs)
```

```
$ make
CC main.o
CC fact.o
CC myfact
MD README.html
```

```
$ make V=1
CC main.o
gcc -Wall -Wextra -Werror -g -c -o main.o main.c
CC fact.o
gcc -Wall -Wextra -Werror -g -c -o fact.o fact.c
CC myfact
gcc -Wall -Wextra -Werror -g -o myfact main.o fact.o
MD README.html
pandoc -o README.html README.md
```

Version 3.0

Generic rules vs dependency tracking

Non-generic rule

```
%.o: %.c fact.h
    @echo "CC   $@"
    $(Q)$(CC) $(CFLAGS) -c -o $@ $<
```

Generic rule

```
%.o: %.c
    @echo "CC $@"
    $(Q)$(CC) $(CFLAGS) -c -o $@ $<
```

Makefile_v3.0


```
$ make
CC main.o
CC fact.o
CC myfact
MD README.html
```

```
$ make
make: Nothing to be done for 'all'.
```

- How can we preserve the generic rule but also have accurate dependency tracking?

```
$ touch fact.h
$ make
make: Nothing to be done for 'all'.
```

Version 3.0

Rule composition

```
%.o: %.c
    @echo "CC $@"
    $(Q)$(CC) $(CFLAGS) -c -o $@ $<
```
Makefile_v3.0

```
main.o: main.c fact.h
fact.o: fact.c fact.h
```

- How can we have these additional rules be generated automatically and included in the Makefile?

```
$ make
CC main.o
CC fact.o
CC myfact
MD README.html
```

```
$ make
make: Nothing to be done for 'all'.
```

```
$ touch fact.h
$ make
CC main.o
CC fact.o
CC myfact
```

Version 3.0

Use GCC for dependency tracking

```c
#include <stdio.h>
#include "fact.h"

int fact(int n) {
    if (n == 0)
        return 1;
    return n * fact(n - 1);
}
```



```
$ gcc -Wall -Wextra -Werror -MMD -c -o fact.o fact.c
```



```
$ cat fact.d
fact.o: fact.c fact.h
```

Version 3.0

Dependency tracking Makefile integration

```
targets := myfact README.html
objs    := main.o fact.o
...
CFLAGS  := -Wall -Wextra -Werror -MMD
...
all: $(targets)

# Dep tracking *must* be below the 'all' rule
deps := $(patsubst %.o,%.d,$(objs))
-include $(deps)
...
%.o: %.c
    @echo "CC $@"
    $(Q)$(CC) $(CFLAGS) -c -o $@ $<
...
clean:
    @echo "clean"
    $(Q)rm -f $(targets) $(objs) $(deps)
```

Makefile_v3.0

- `$(deps)` will be computed from `$(obj)` into `main.d fact.d`
- Prefix `-` ignores inclusion errors

Version 3.0

First run

- Dependency files don't exist but make won't complain
- GCC generates them

```
$ ls *.d
```

```
$ make
CC main.o
CC fact.o
CC myfact
MD README.html
$ ls *.d
main.d fact.d
```

```
$ cat main.d
main.o: main.c fact.h
$ cat fact.d
fact.o: fact.c fact.h
```

Following runs

- Dependency files are included by the Makefile
- They are used to compose the generic rule for object generation

```
$ make
make: Nothing to be done for 'all'
$ touch fact.h
$ make
CC main.o
CC fact.o
CC myfact
```

Final Makefile

```
targets := myfact README.html
objs    := main.o fact.o

CC      := gcc
CFLAGS  := -Wall -Wextra -Werror -MMD
CFLAGS  += -g
PANDOC  := pandoc

ifneq ($(V),1)
Q = @
endif

all: $(targets)

# Dep tracking *must* be below the 'all' rule
deps := $(patsubst %.o,%.d,$(objs))
-include $(deps)

myfact: $(objs)
    @echo "CC $@"
    $(Q)$(CC) $(CFLAGS) -o $@ $^

%.o: %.c
    @echo "CC $@"
    $(Q)$(CC) $(CFLAGS) -c -o $@ $<

%.html: %.md
    @echo "MD $@"
    $(Q)$(PANDOC) -o $@ $<

clean:
    @echo "clean"
    $(Q)rm -f $(targets) $(objs) $(deps)
```

Makefile_v3.0