

# CSE306 Software Quality in Practice

Dr. Carl Alphonse  
alphonse@buffalo.edu  
343 Davis Hall



Lecture question



# make

What is it good for?

"You can use [make] to describe any task where some files must be updated automatically from others whenever the others change."

[<https://www.gnu.org/software/make/manual/make.pdf>, page 1]



# make and makefiles

- makefile contains rules that describe update dependencies



# rules

target : prerequisites  
recipe



# rules

target : prerequisites

recipe

Must be a tab!



# target

- A target is usually the name of a file that needs to be generated/updated during the 'make' process
- The rule will be used by 'make' when the target is out-of-date, and so should say how to update the target

A target is out-of-date if it is older than any of its prerequisite files.

For example:  
foo.o : foo.c foo.h

The object file foo.o is considered out-of-date if either its .c or .h file is newer.



# target

- A target is usually the name of a file that needs to be generated/updated during the 'make' process
- The rule will be used by 'make' when the target is out-of-date, and so should say how to update the target

"Bear in mind that make does not know anything about how the recipes work. It is up to you to supply recipes that will update the target file properly. All make does is execute the recipe you have specified when the target file needs to be updated." [p. 5]



# target

```
primOpt.o: primOpt.c primOpt.h  
gcc -c -Wall primOpt.c
```



# target

- A target can be "phony" - an arbitrary label for an action given by the rest of the rule



# target

clean:

```
rm -f primOpt.o main
```

"...the clean target will not work properly if a file named clean is ever created in this directory. Since it has no prerequisites, clean would always be considered up to date and its recipe would not be executed. To avoid this problem you can explicitly declare the target to be phony by making it a prerequisite of the special target .PHONY" [p. 29]



# target

.PHONY: clean

clean:

```
rm -f primOpt.o main
```



# Example

[https://www.gnu.org/software/make/manual/html\\_node/Simple-Makefile.html](https://www.gnu.org/software/make/manual/html_node/Simple-Makefile.html)

```
edit : main.o kbd.o command.o display.o \  
insert.o search.o files.o utils.o  
    cc -o edit main.o kbd.o command.o display.o \  
        insert.o search.o files.o utils.o  
main.o : main.c defs.h  
    cc -c main.c  
kbd.o : kbd.c defs.h command.h  
    cc -c kbd.c  
command.o : command.c defs.h command.h  
    cc -c command.c  
display.o : display.c defs.h buffer.h  
    cc -c display.c  
insert.o : insert.c defs.h buffer.h  
    cc -c insert.c  
search.o : search.c defs.h buffer.h  
    cc -c search.c  
files.o : files.c defs.h buffer.h command.h  
    cc -c files.c  
utils.o : utils.c defs.h  
    cc -c utils.c  
clean :  
    rm edit main.o kbd.o command.o display.o \  
        insert.o search.o files.o utils.o
```



# variables

"A variable is a name defined in a makefile to represent a string of text, called the variable's value. These values are substituted by explicit request into targets, prerequisites, recipes, and other parts of the makefile."

[p. 59]



# variables

"A variable name may be any sequence of characters not containing ':', '#', '=', or whitespace. However, variable names containing characters other than letters, numbers, and underscores should be considered carefully, as in some shells they cannot be passed through the environment to a sub-make [...] Variable names beginning with '\_' and an uppercase letter may be given special meaning in future versions of make.

Variable names are case-sensitive. The names 'foo', 'FOO', and 'Foo' all refer to different variables."



# variables

"To substitute a variable's value, write a dollar sign followed by the name of the variable in parentheses or braces: either '\$(foo)' or '\${foo}' is a valid reference to the variable foo."

[p. 59]



objects  
variable  
defined

```
objects = main.o kbd.o command.o display.o \  
insert.o search.o files.o utils.o  
edit : $(objects)  
    cc -o edit $(objects)  
main.o : main.c defs.h  
    cc -c main.c  
kbd.o : kbd.c defs.h command.h  
    cc -c kbd.c  
command.o : command.c defs.h command.h  
    cc -c command.c  
display.o : display.c defs.h buffer.h  
    cc -c display.c  
insert.o : insert.c defs.h buffer.h  
    cc -c insert.c  
search.o : search.c defs.h buffer.h  
    cc -c search.c  
files.o : files.c defs.h buffer.h command.h  
    cc -c files.c  
utils.o : utils.c defs.h  
    cc -c utils.c  
clean :  
    rm edit $(objects)
```

objects  
variable  
used

Example



# implicit rules

primOpt.o : primOpt.h primOpt.c

expands to

primOpt.o : primOpt.h primOpt.c

\$(CC) \$(CPPFLAGS) \$(CFLAGS) -c

CC has default value cc



# Example

[https://www.gnu.org/software/make/manual/html\\_node/make-Deduces.html#make-Deduces](https://www.gnu.org/software/make/manual/html_node/make-Deduces.html#make-Deduces)

```
objects = main.o kbd.o command.o display.o \  
insert.o search.o files.o utils.o
```

```
edit : $(objects)  
      cc -o edit $(objects)
```

```
main.o : defs.h  
kbd.o : defs.h command.h  
command.o : defs.h command.h  
display.o : defs.h buffer.h  
insert.o : defs.h buffer.h  
search.o : defs.h buffer.h  
files.o : defs.h buffer.h command.h  
utils.o : defs.h
```

```
.PHONY : clean  
clean :  
      rm edit $(objects)
```

use  
implicit  
rules



# Example

[https://www.gnu.org/software/make/manual/html\\_node/Combine-By-Prerequisite.html#Combine-By-Prerequisite](https://www.gnu.org/software/make/manual/html_node/Combine-By-Prerequisite.html#Combine-By-Prerequisite)  
[https://www.gnu.org/software/make/manual/html\\_node/Cleanup.html#Cleanup](https://www.gnu.org/software/make/manual/html_node/Cleanup.html#Cleanup)

```
objects = main.o kbd.o command.o display.o \  
insert.o search.o files.o utils.o
```

```
edit : $(objects)  
    cc -o edit $(objects)
```

```
$(objects) : defs.h
```

```
kbd.o command.o files.o : command.h
```

```
display.o insert.o search.o files.o : buffer.h
```

```
.PHONY : clean
```

```
clean :  
    rm edit $(objects)
```

abbreviate  
prerequisites  
⌘  
reorganize rules



# Variables

```
objects = main.o kbd.o command.o display.o \  
insert.o search.o files.o utils.o
```

# Rules

```
edit : $(objects)  
      cc -o edit $(objects)
```

```
$(objects) : defs.h
```

```
kbd.o command.o files.o : command.h
```

```
display.o insert.o search.o files.o : buffer.h
```

```
.PHONY : clean
```

```
clean :  
      rm edit $(objects)
```



add  
comments

EXAMPLE



# Naming your makefile

- Typical names: makefile or Makefile
- GNU-make specific: GNUmakefile
- Can also use another name and invoke make with `-f filename`



# GNU make, cont'd

text and examples from gmake documentation

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

• So far:

- rules, targets, prerequisites, recipes
- explicit and implicit rules
- makefile naming
- phony targets
- variables
- comments



## 4.3 Types of Prerequisites

<https://www.gnu.org/software/make/manual/make.html#Prerequisite-Types>

- There are two different types of prerequisites: **normal prerequisites** and **order-only prerequisites**.
- A **normal prerequisite** makes **two** statements: **first**, it imposes an order in which recipes will be invoked: the recipes for all prerequisites of a target will be completed before the recipe for the target is run. **Second**, it imposes a dependency relationship: if any prerequisite is newer than the target, then the target is considered out-of-date and must be rebuilt.
- Occasionally, however, you have a situation where you want to impose a specific ordering on the rules to be invoked without forcing the target to be updated if one of those rules is executed. In that case, you want to define order-only prerequisites. **Order-only prerequisites** can be specified by placing a pipe symbol (|) in the prerequisites list: any prerequisites to the left of the pipe symbol are normal; any prerequisites to the right are order-only.



## 4.3 Types of Prerequisites

<https://www.gnu.org/software/make/manual/make.html#Prerequisite-Types>

- Consider an example where your targets are to be placed in a separate directory, and that directory might not exist before make is run. In this situation, you want the directory to be created before any targets are placed into it but, because the timestamps on directories change whenever a file is added, removed, or renamed, we certainly don't want to rebuild all the targets whenever the directory's timestamp changes. One way to manage this is with order-only prerequisites: make the directory an order-only prerequisite on all the targets:

```
OBJDIR := objdir
OBS := $(addprefix $(OBJDIR)/,foo.o bar.o baz.o)

$(OBJDIR)/%.o : %.c
    $(COMPILE.c) $(OUTPUT_OPTION) $<

all: $(OBS)

$(OBS): | $(OBJDIR)

$(OBJDIR):
    mkdir $(OBJDIR)
```

- Now the rule to create the objdir directory will be run, if needed, before any '.o' is built, but no '.o' will be built because the objdir directory timestamp changed.