



# Introduction to Make

Prof. Darrell Long  
Edits by Dr. Kerry Veenstra  
CSE 13S

# Why Use a Makefile?

- What about using a command script?

```
$ clang -Wall -Wextra -pedantic -c sorting.c
```

```
$ clang -Wall -Wextra -pedantic -c insert.c
```

```
$ clang -Wall -Wextra -pedantic -c quick.c
```

```
$ clang -Wall -Wextra -pedantic -c heap.c
```

```
$ clang sorting.o insert.o quick.o heap.o -o sorting
```

# Problem #1: Duplicate options

- What about using a command script?

```
$ clang -Wall -Wextra -pedantic -c sorting.c
```

```
$ clang -Wall -Wextra -pedantic -c insert.c
```

```
$ clang -Wall -Wextra -pedantic -c quick.c
```

```
$ clang -Wall -Wextra -pedantic -c heap.c
```

```
$ clang sorting.o insert.o quick.o heap.o -o sorting
```

- Could use variables in the script file

## Problem #2: Duplicate "Rules"

- What about using a command script?

```
$ clang -Wall -Wextra -pedantic -c sorting.c
```

```
$ clang -Wall -Wextra -pedantic -c insert.c
```

```
$ clang -Wall -Wextra -pedantic -c quick.c
```

```
$ clang -Wall -Wextra -pedantic -c heap.c
```

```
$ clang sorting.o insert.o quick.o heap.o -o sorting
```

# Problem #3: Unnecessary Recompilation

- What about using a command script?

```
$ clang -Wall -Wextra -pedantic -c sorting.c
```

```
$ clang -Wall -Wextra -pedantic -c insert.c
```

```
$ clang -Wall -Wextra -pedantic -c quick.c
```

```
$ clang -Wall -Wextra -pedantic -c heap.c
```

```
$ clang sorting.o insert.o quick.o heap.o -o sorting
```

- More of a problem with large projects (thousands of files)

# Problem #4: Only One Target

- What about using a command script?

```
$ clang -Wall -Wextra -pedantic -c sorting.c
```

```
$ clang -Wall -Wextra -pedantic -c insert.c
```

```
$ clang -Wall -Wextra -pedantic -c quick.c
```

```
$ clang -Wall -Wextra -pedantic -c heap.c
```

```
$ clang sorting.o insert.o quick.o heap.o -o sorting
```

- Can't put "all", "clean", "debug", etc. in one file

# Example Makefile

- Example of a generic Makefile that works for many simple programs:

```
1 CC      = clang
2 CFLAGS  = -Wall -Wextra -Wpedantic -Werror
3 SRC     = $(wildcard *.c)
4 OBJ     = $(SRC:.c=.o)
5 EXECBIN = foo
6
7 .PHONY: all clean debug
8
9 all: $(EXECBIN)
10
11 clean:
12     rm -f $(OBJ) $(EXECBIN)
13
14 debug: CFLAGS += -g
15 debug: clean all
16
17 $(EXECBIN): $(OBJ)
18     $(CC) $(CFLAGS) $(OBJ) -o $(EXECBIN)
19
20 %.o: %.c
21     $(CC) $(CFLAGS) -c $<
```

# Targets

- The file to be made

target

dependency

```
sorting.o: sorting.c
```

command(s)

```
clang -Wall -Wextra -pedantic -c sorting.c
```



# Targets

- The file to be made

```
sorting.o: sorting.c
```

```
 clang -Wall -Wextra -pedantic -c sorting.c
```



"tab" character

- Be sure to use a real "tab" character
- In vim/vi may need to use "Ctrl-V Tab"

# Targets

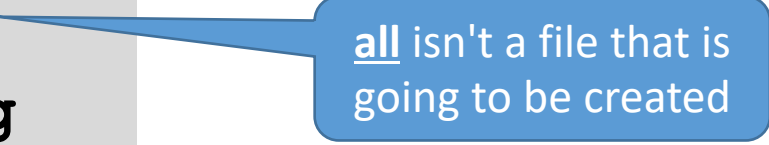
- The file to be made

```
sorting.o: sorting.c
```

```
    clang -Wall -Wextra -pedantic -c sorting.c
```

- A useful name for a "rule"

```
.PHONY: all  
all: sorting
```



all isn't a file that is going to be created

# What's a phony target?

```
.PHONY: clean  
clean:  
    rm *.o
```

- A target that, when its rule is executed, doesn't produce a file with the same name.
- Important since it prevents make from erroneously checking for files since real and phony targets are now differentiated.
- A phony target is marked as phony by simply having “.PHONY <target\_name>” directly above itself.
  - .PHONY is another example of a special built-in target name
- Example of a phony target:

# Some conventional phony targets

- clean:
  - Removes any files produced by running a make command.
  - Be careful to only remove executables and object files, not source code.
- all:
  - Usually consists of a list of dependencies which includes all the rules defined in the Makefile.
  - Usually placed as the first rule so running “make” defaults to executing it.
- debug:
  - Builds a debug version of the program
  - debug: CFLAGS += -g
  - debug: clean all

# Dependencies Can Be Other Targets

```
all: sorting
sorting: sorting.o
        clang sorting.o -o sorting
sorting.o: sorting.c
        clang -Wall -Wextra -pedantic -c sorting.c
```

# Variables

- Eliminating duplicate code

```
$ clang -Wall -Wextra -pedantic -c sorting.c
```

```
$ clang -Wall -Wextra -pedantic -c insert.c
```

```
$ clang -Wall -Wextra -pedantic -c quick.c
```

```
$ clang -Wall -Wextra -pedantic -c heap.c
```

```
$ clang sorting.o insert.o quick.o heap.o -o sorting
```

# Variables

- Eliminating duplicate code

```
CC = clang
```

```
CFLAGS = -Wall -Wextra -pedantic
```

```
sorting.o: sorting.c
```

```
    $(CC) $(CFLAGS) -c sorting.c
```

```
insert.o: insert.c
```

```
    $(CC) $(CFLAGS) -c insert.c
```

# Variables

- Put **updates** in one place

```
CC = clang
```

```
CFLAGS = -Wall -Wextra -Wstrict-prototypes -Werror -pedantic
```

```
sorting.o: sorting.c
```

```
    $(CC) $(CFLAGS) -c sorting.c
```

```
insert.o: insert.c
```

```
    $(CC) $(CFLAGS) -c insert.c
```



# Variables

- Appending to an existing variable's value

```
CFLAGS = -Wall -Wextra -Wstrict-prototypes -Werror -pedantic
```

```
CFLAGS += -g
```

# Automatic Variables

- Avoid even more duplication

```
CC = clang
```

```
CFLAGS = -Wall -Wextra -Wstrict-prototypes -Werror -pedantic
```

```
sorting.o: sorting.c
```

```
    $(CC) $(CFLAGS) -c sorting.c
```

```
insert.o: insert.c
```

```
    $(CC) $(CFLAGS) -c insert.c
```

# Automatic Variables

- Avoid even more duplication

```
CC = clang
```

```
CFLAGS = -Wall -Wextra -Wstrict-prototypes -Werror -pedantic
```

```
%.o: %.c
```

```
$(CC) $(CFLAGS) -c $<
```



"first dependency"

# Automatic Variables

- Avoid even more duplication

```
CC = clang
```

```
CFLAGS = -Wall -Wextra -Wstrict-prototypes -Werror -pedantic
```

```
%.o: %.c
```

```
$(CC) $(CFLAGS) -c $<
```

```
sorting: sorting.o insert.o quick.o heap.o
```

```
$(CC) $^ -o $@
```

all dependencies

target

# Automatic Variables

- Avoid even more duplication

```
CC = clang
```

```
CFLAGS = -Wall -Wextra -Wstrict-prototypes -Werror -pedantic
```

```
OBJS = sorting.o insert.o quick.o heap.o
```

```
%.o: %.c
```

```
    $(CC) $(CFLAGS) -c $<
```

```
sorting: $(OBJS)
```

```
    $(CC) $^ -o $@
```



# What's the `make` program?

---

- A utility on most Unix systems that automatically builds executable programs and libraries from source code.
- Has several derivatives, one of which is GNU Make (`gmake`), the standard Make implementation on Linux/OSX (also on the UCSC Unix Timeshare and Ubuntu).
- `cmake`, a cross-platform build-system generator, is a program that produces a Makefile for Unix systems.
- Has a variety of command line flags/options to select from.
- Requires a `Makefile`.

# What's a Makefile?

---

- Plaintext file that contains instructions for the `make`.
  - Has a syntax like any programming language
  - You can think of it as a script
- Typically resides in the same directory as source code and is named “Makefile”.
- Composed of rules.

*“A makefile is just a directed acyclic graph” – Darrell Long, 2018*

# What's a rule?

- Associated with/composed of the following:
  1. A target
  2. A set of dependencies
  3. A set of commands
- General structure of a rule:
  - `target ... : dependencies ...`  
`<TAB> command`  
`<TAB> ...`  
`<TAB> ...`



# What's a target?

---

- Basically the name of a *rule*.
- Users specify which target to make by running “make <target\_name>”.
  - Ex: `$ make all` builds the all target.
- Usually the name of the file that is generated by executing the rule's commands but can also be a name of an action to perform (a phony target).
- If a target isn't specified on the command line, make defaults to making the first rule:
  - Users can designate a default make target using `.DEFAULT_GOAL := <target_name>`.
  - `.DEFAULT_GOAL` is an example of a special built-in target name

## What's a phony target?

```
.PHONY: clean  
clean:  
    rm *.o
```

- A target that, when its rule is executed, doesn't produce a file with the same name.
- Important since it prevents make from erroneously checking for files since real and phony targets are now differentiated.
- A phony target is marked as phony by simply having “.PHONY <target\_name>” directly above itself.
  - .PHONY is another example of a special built-in target name
- Example of a phony target:

# Some conventional phony targets

- clean:
  - Removes any files produced by running a make command.
  - Be careful to only remove executables and object files, not source code.
- all:
  - Usually consists of a list of dependencies which includes all the rules defined in the Makefile.
  - Usually placed as the first rule so running “make” defaults to executing it.
- debug:
  - Builds a debug version of the program
  - debug: CFLAGS += -g
  - debug: clean all

# Some useful make flags/options

- `-C <dir_name>, --directory=<dir_name>`
  - Changes to directory before looking/running any Makefiles.
- `-d`
  - Print debug information in addition to any normal processing information.
- `-f <file_name>, --file=<file_name>, --makefile=<file_name>`
  - Specifies the file to be read as the Makefile.
- `-I <dir_name>, --include-dir=<dir_name>`
  - Specifies the directory to search in for Makefiles.
- `--warn-undefined-variables`
  - Warns about referencing of undefined variables.


# Variables in Makefiles

---

- Four types of variable assignments in a Makefile
  1. “=” – lazy assignment
  2. “:=” – immediate assignment
  3. “?=” – conditional assignment
  4. “+=” – concatenation
- To use the value of any variable, surround the variable in parentheses or curly brackets and prepend a dollar sign:
  - `$(<variable_name>)` or `${<variable_name>}`

# Lazy Assignment – “=”

- Variables lazily assigned using the “=” operator are called recursively-expanded variables.
- Contents of the variable assignment are stored as-is, variables as variables, references as references.
- `make` waits to expand the variable’s references until the variable is being used.
- Example:




```
x = foo
y = $(x)
# $(x) and $(y) both now yield "foo"

x = bar
# $(x) and $(y) both now yield "bar"
```

# Lazy Assignment – “=”

- Variables lazily assigned using the “=” operator are called recursively-expanded variables.
- Contents of the variable assignment are stored as-is, variables as variables, references as references.
- `make` waits to expand the variable’s references until the variable is being used.
- Example:



```
x = foo
y = $(x)
# $(x) and $(y) both now yield "foo"

x = bar
# $(x) and $(y) both now yield "bar"
```

# Immediate Assignment – “:=”

- Variables immediately assigned with “:=” operator are called simply-expanded variables and **behave like variables in C**.
- Variable assignment is evaluated, and the result assigned to the variable.
- If variable assignment is a variable reference, the reference is expanded before the assignment.
- Example:

```
x := foo
y := $(x)
# $(x) and $(y) both now yield "foo"
```

```
x := bar
# $(x) now yields "bar" and $(y) still yields "foo"
```



# Conditional Assignment – “?=”

- “?=” behaves like “=”, except **assignment occurs if and only if the variable hasn’t been assigned a value yet.**
- Example:

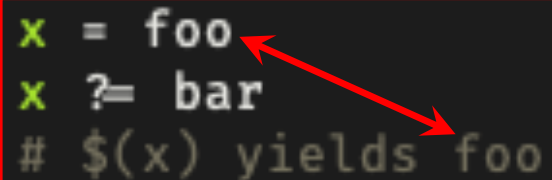
```
x = foo
x = bar
# $(x) yields foo

y <?= baz
# $(y) yields baz</pre
```

# Conditional Assignment – “?=”

- “?=” behaves like “=”, except **assignment occurs if and only if the variable hasn’t been assigned a value yet.**
- Example:

```
x = foo  
x ?= bar  
# $(x) yields foo
```



```
y ?= baz  
# $(y) yields baz
```

# Conditional Assignment – “?=”

- “?=” behaves like “=”, except **assignment occurs if and only if the variable hasn’t been assigned a value yet.**
- Example:

```
x = foo
x = bar
# $(x) yields foo

y <?= baz
# $(y) yields baz</pre
```

# Concatenation – “+=”

- Behaves like “=” if the variable in question hasn’t been defined.
- **Essentially adds extra text to a defined variable separated by space.**
- Example:
  - `x := foo`
  - `x += bar` **x is "foo bar"**
  - Is equivalent to:
    - `x := foo`
    - `x := $(x) bar` **x is "foo bar"**
- **Useful for adding debug flags like “-g” to defined compiler flags.**

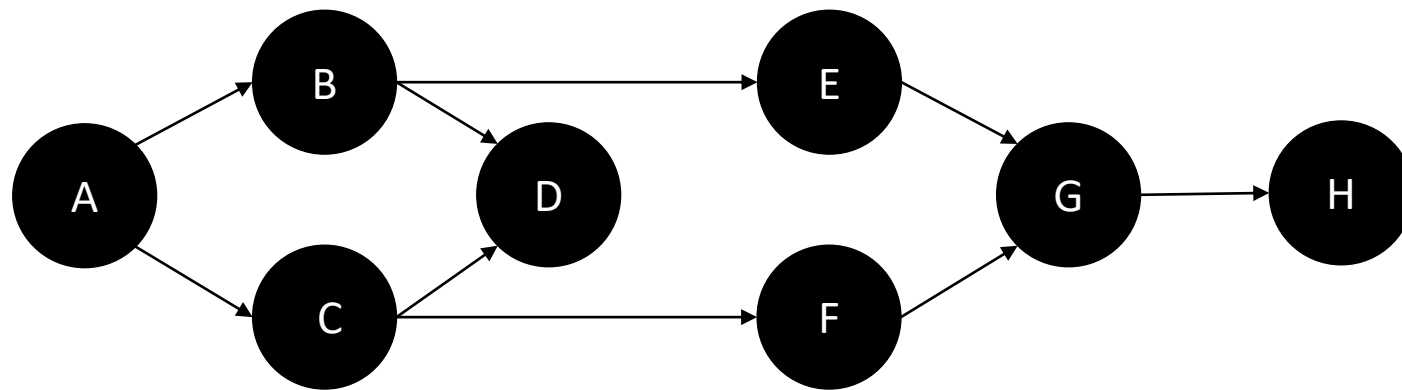
# What's a dependency?

---

- Either a target or a filename (this includes source and header files).
  - Ex: build: clean hello.o
  - The build target depends on the clean target and the hello.o file
- If a rule has a dependency that has been modified or if its target doesn't exist, `make` tries to fill in the dependency by executing the rule with the dependency name.
- Else if the dependency hasn't been modified or if its target already exists, `make` doesn't execute the rule to make the dependency.
- The above two points illustrate how `make` tries to be efficient; don't make something that is already up-to-date.

# Topological Ordering

- Think of targets in a Makefile as vertices in a direct acyclic graph, and their dependency(s) representing a directed edge in said graph
- A topological ordering on a Makefile ensures a linear ordering of dependencies such that for every target  $A$  that depends on some target  $B$ ,  $B$  will always be compiled before  $A$ .



# What's a command?

---

- An action to be executed.
- Can generally use shell scripting commands (think bash commands).
- A rule can have more than one command, each on its own line.
- **Must have a tab character in front of entire command.**
  - clean:  
    <TAB> rm -f \*.o hello

# Commands, Compilers, and Compiler Flags

---

- Variables are used to factor makefiles to make them easier to maintain.
- Some conventional Makefile variables used for compiling C programs:
  - **CC** – the C compiler to use (typically gcc or cc or clang).
  - **CFLAGS** – a list of compiler flags (each flag is prefixed with a hyphen “-”).
    - Some good flags to use when compiling C programs:
      - Wall, Wextra, Wpedantic, Werror, g.
  - OBJ – a list of object files to build and link.
  - SRC – a list of source files (.c files).



# What we can do so far!

- But can we make this better?

```
1 CC      = clang
2 CFLAGS  = -Wall -Wextra -Wpedantic -Werror
3 OBJ     = main.o foo.o
4 EXECBIN = my_program
5
6 .PHONY: all clean
7
8 all: $(EXECBIN)
9
10 $(EXECBIN): $(OBJ)
11     $(CC) $(CFLAGS) $(OBJECTS) -o $(EXECBIN)
12
13 main.o: main.c
14     $(CC) $(CFLAGS) -c main.c
15
16 foo.o: foo.c
17     $(CC) $(CFLAGS) -c foo.c
18
19 clean:
20     rm -f $(OBJ) $(EXECBIN)
```

# Automatic Variables

- Make automatically defines some special variables within the context of an individual rule.

- Some useful automatic variables:

- “\$@”: the name of the target.

```
foo: foo.c
    gcc foo.c -o $@
# $a is set to "foo"
```

- “\$^”: list of all dependencies for target.

```
viewsrc: foo.c bar.c
    less $^
# $^ is set to "foo.c bar.c"
```

- “\$?”: list of dependencies more recent than the target.

```
foo: foo.c
    gcc $? -o $@
# $? set to foo.c if foo.c was
  updated more recently than foo
```

- “\$<”: the name of the first dependency.

```
foo.o: foo.c
    gcc -c $< -o $@
# $c is set to foo.c
```

# The shell function

- Communicates with the world outside of make.
- Performs command expansion– takes a shell command and evaluates to the output of the command.
- Newlines in command output are converted to spaces.
- A useful example of the shell function:

```
SRC := $(shell ls *.c)  
# $(SRC) now yields a list of all C files in the current directory.
```

- The above example can also be done using the wildcard function.

# The wildcard function

- Can be used in rules as the “\*” operator, where it is expanded by the shell.
  - A rule using a phony target to delete object files using wildcard:

```
.PHONY: clean  
clean:  
    rm *.o
```

- If used for a variable assignment, wildcard expansion doesn't occur using “\*” unless the wildcard function is explicitly specified.
- An example to get source files using wildcard:

```
SRC := $(wildcard *.c)  
# $(SRC) now yields a list of all C files in the current directory.
```

# The `patsubst` function

- Is formatted as `$(patsubst pattern, replacement, text)`.
- Finds whitespace-separated words in `<text>` that match `<pattern>` and replaces them with `<replacement>`.
- An example of using `patsubst` to get a list of object files:

```
OBJ := $(patsubst %.c,%.o,$(wildcard *.c))  
# $(OBJ) now yields a list of object files corresponding to its source file.
```

- Note that the above example uses the pattern match placeholder operator “%” instead of the wildcard operator “\*”.
  - This wildcard matches any number of any characters within a word.

## “\*” VS “%”

---

- Difference is that “\*” performs expansion and “%” serves as a pattern match placeholder.
- Consider the previous example where object filenames are generated by source filenames:
  - \$(wildcard \*.c) performs wildcard expansion to get all the source files.
  - %.c makes % match all text leading up to “.c”.
  - “.c” is then replaced by “.o” in all instances of a match from %.c.
- The previous example can also be written as:
  - OBJ = \$(SRC:%.c=%.o):
    - This effectively replaces “.c” with “.o”, which is more intuitive, but `patsubst` is good to know.

# Pattern matching

- Utilizes the pattern match placeholder "%".

- Example:

```
%.o: %.c  
gcc -c $< -o $@
```

- When “make foo.o” is typed, this rule pattern matches to:

```
foo.o: foo.c  
gcc -c $< -o $@
```

- Note that automatic variables are needed to designate the first dependency and target names.

# Pattern Matching (cont'd)

- Great for scalability (easier to apply one rule for many many files).
- Doesn't match every filename.
- Only executes if there's a dependency that needs to be created which matches the rule.
- Example:

```
foo: foo.o
    gcc $? -o $@

%.o: %.c
    gcc -c $<
```

- Running `$ make foo` looks for “foo.o”, so the rule becomes `foo.o: foo.c`.
- Even if there are other C files in the directory, the pattern matching isn't executed.



# Putting it all together

- Example of a generic Makefile that works for many simple programs:

```
1 CC      = clang
2 CFLAGS  = -Wall -Wextra -Wpedantic -Werror
3 SRC     = $(wildcard *.c)
4 OBJ     = $(SRC:.c=.o)
5 EXECBIN = foo
6
7 .PHONY: all clean debug
8
9 all: $(EXECBIN)
10
11 clean:
12     rm -f $(OBJ) $(EXECBIN)
13
14 debug: CFLAGS += -g
15 debug: clean all
16
17 $(EXECBIN): $(OBJ)
18     $(CC) $(CFLAGS) $(OBJ) -o $(EXECBIN)
19
20 %.o: %.c
21     $(CC) $(CFLAGS) -c $<
```

# Recursive use of make

- Using make as a command in a Makefile.
- Useful for when you want separate Makefiles for various subsystems that are part of a larger system.
- For example:

```
subsystem:  
    $(MAKE) -C subsystem
```

- The subsystem target recursively calls `make` and uses the Makefile in the subsystem directory.
- You can specify options to a sub-`make` for flexibility when building subsystems.

```
subsystem-debug:  
    $(MAKE) -C subsystem DEBUG=y
```

# Including Makefiles

- A Makefile can include other Makefiles.
- Useful for when various programs need to use a common set of variable definitions
  - You can think of it like including a header file and having a common set of variables and constants.
- Another common use case is generating a prerequisite by including its Makefile and building it where needed.
- For example, in a Makefile you can have the following:

```
include common.mk
```

- It is convention for secondary Makefiles that themselves do not build the module to have a .mk extension.
- However, including a Makefile overrides variables and targets with the same name. Hence, it is usually better to use a recursive approach.