GNU Make and makefiles

Managing Linux With the Embedded Perspective TX00EX85-3001

1.09.2024

What is GNU Make?

Aside from being awesome

- Make is a utility
  - Typically used to manage the build process of software projects
  - Very often used for GCC, but is not at all limited to that and can be used for any development or software packaging
- Was created in 1976 and released for free in 1980s
- Developed in Bell Labs (Same place that the C programming language was created)

What can Make do?

- Make is a very intelligent utility
  - It allow users to build and install things without knowledge of how that is done
    - Developer provides a *makefile* and the user has make installed on their system
  - Make figures out what needs to be rerun
    - Make will not re-compile all files, just the ones that need to be re-compiled
  - Make can specify shell commands for building any language

Where is GNU make used?

- As was covered, there are several steps to consider in a build process
  - As such larger projects demand more from a developer
- Make is used very often for large C/C++ projects as it simplifies and automates the build and packaging process for the developers
- We can find GNU Make used in projects that we are all familiar with:
  - GCC (GNU Compiler Collection), The Linux Kernel, LibreOffice, Mozilla Firefox...
- Even used in Web development for repetitive tasks
  - Allows for compilation and installation of sources without having to be a developer
  - Often when downloading a project from Github a *makefile* is to automate the build and installation
    - Requires two commands, *make* and *make install*

How to use Make

- You will need to have make installed on your system
  - Can be installed with your package manager but comes with the *build-essential* meta package that you probably have installed
- You will also need to have any other compilers and build tools required to build your applications installed on your system
  - Again *build-essential will have most everything you need*
- You will need a text editor
- You will need to create a *makefile*

How to use Make (cont.)

- Once a functional *makefile* has been created, you can simply run the command *make* from the same directory as your *makefile*
- *make* can be run with additional information to run specific "targets" or to cleanup your build directories of any unwanted remnants from the compilation process

What is a *makefile*

- A make file is just a standard text file that contains a collection of rules required for building your project
- The developer creates rules as they see fit
  - As such it is essential to understand the build process and what you wish to do

What is a *makefile*

- A make file is just a standard text file that contains a collection of rules required for building your project
- The developer creates rules as they see fit
  - As such it is essential to understand the build process and what you wish to do

makefile rules

- A *makefile* rule can contain the following
  - Target
    - What is going to be created by the rule
    - example: .o file or executable
  - Dependencies
    - What is required to make the target
    - to create .o files we need .c files and to link an executable we need .o files
  - Recipes
    - the action that make will carry out
    - *gcc -o <executable name> .c (or .o) files*

File: makefile

```
1  target(s) : dependencies ...
2      recipe
3      ...
4      ...
5      ...
6
```

Not all targets create files

- Sometimes we want to do something other than compile files with make
- We might want to cleanup or delete unneeded files or remnants from compilation in a directory
- Make rules can be created for this
  - One of the most common rules will have a target "clean" used to cleanup a directory on .o files or old executables

File: makefile

```
1  clean:
2      rm -f *.o
3
```

.PHONY

- .PHONY targets are not linked to the name of any specific file, but rather to a recipe when explicitly named
- Used to avoid conflicts with files that might share a name with the .PHONY target
- Very often used with the clean target
  - clean typically executes the rm command (which doesn't output a file)
  - Every time *make clean* is executed the rm command will be run (regardless of whether the files to be removed exist or not)
  - If the .PHONY target is not used then in the case where there might be a file called clean in the directory, then *make clean* will not work.

Recipes

- Each line of a recipe should start with a tab character
- Teh recipe is the commands that will be executed for a specific target
- Recipes can be made of one or more command
- Recipes describe what to do with the dependencies to create a target
- Can be any terminal command

File: **makefile**

```
1  target(s) : dependencies ...
2      recipe
3      ...
4      ...
5      ...
6
```

Recipes

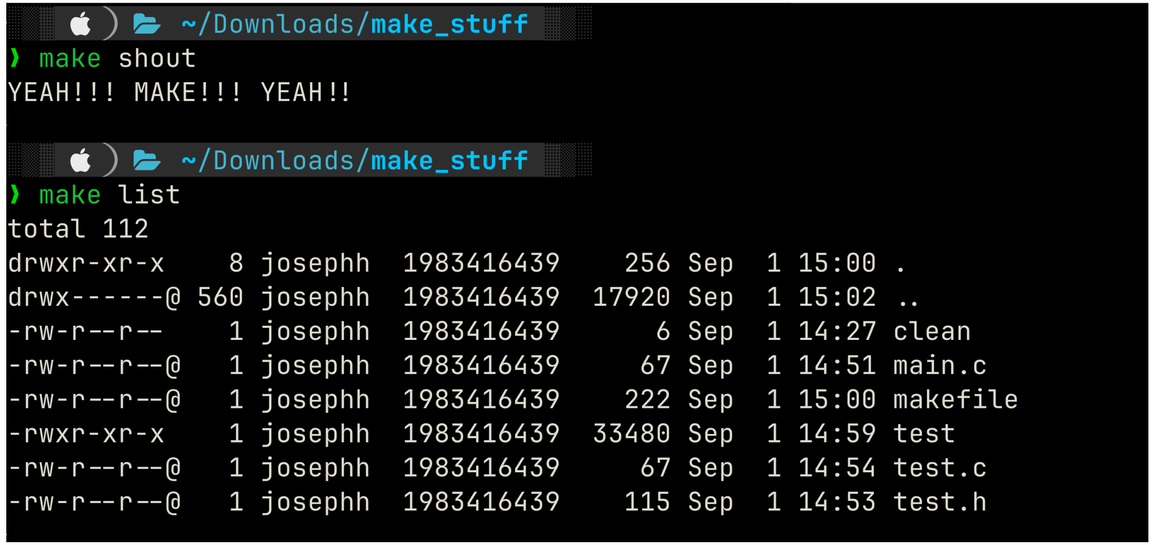
- Each line of a recipe should start with a tab character
- The recipe is the commands that will be executed for a specific target
- Recipes can be made of one or more command
- Recipes describe what to do with the dependencies to create a target
- Can be any terminal command

File: makefile

```
1  test: main.o test.o
2      gcc -o test main.o test.o
3
4  main.o test.o: main.c test.c
5      gcc -c main.c test.c
6
7  .PHONY: shout
8
9  shout:
10     @echo "YEAH!!! MAKE!!! YEAH!!"
11
12 .PHONY: list
13
14 list:
15     @ls -la
16
17 .PHONY: clean
18
19 clean:
20     rm -f *.o
21
```

Recipes

- Each line of a recipe should start with a tab character
- Teh recipe is the commands that will be executed for a specific target
- Recipes can be made of one or more command
- Recipes describe what to do with the dependencies to create a target
- Can be any terminal command



How make parses a *makefile*

- When *make* is instantiated from the terminal it tries to execute the top target (default goal)
- It will try to run the first target, if that target depends on another target it will find and run that target first. If that target depends on another, then... and so on until the default goal is created or an error is encountered
- If a rule is not a dependent of the default goal, then it will not be run
- If a rule doesn't need to be rerun because the target does not need to be updated, it will not be run
- If you specify a target, then it and its dependents will be run only

Makefile

File: makefile

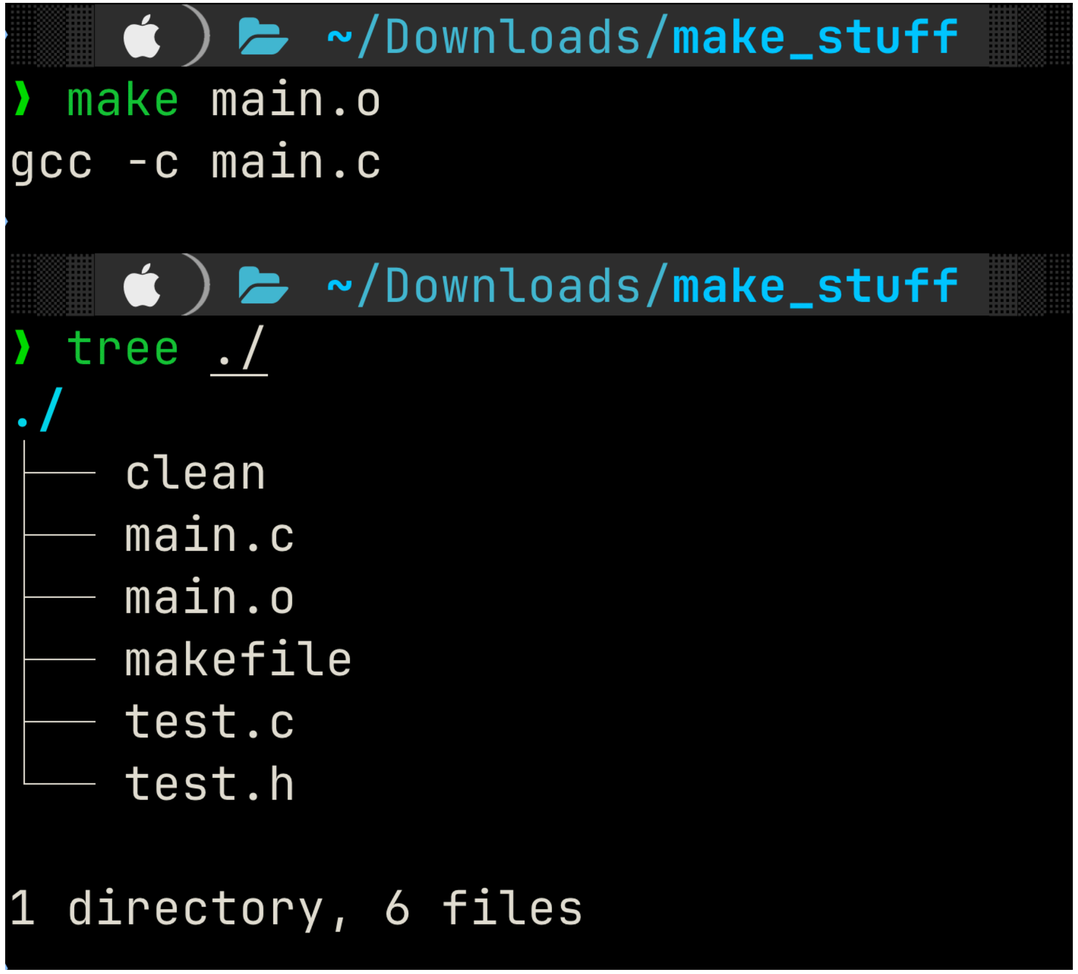
```
1  test: main.o test.o
2      gcc -o test main.o test.o
3
4  main.o: main.c
5      gcc -c main.c
6
7  test.o: test.c
8      gcc -c test.c
9
10 .PHONY: shout
11
12 shout:
13     @echo "YEAH!!! MAKE!!! YEAH!!"
14
15 .PHONY: list
16
17 list:
18     @ls -la
19
20 .PHONY: clean
21
22 clean:
23     rm -f *.o test
```

Running *make*

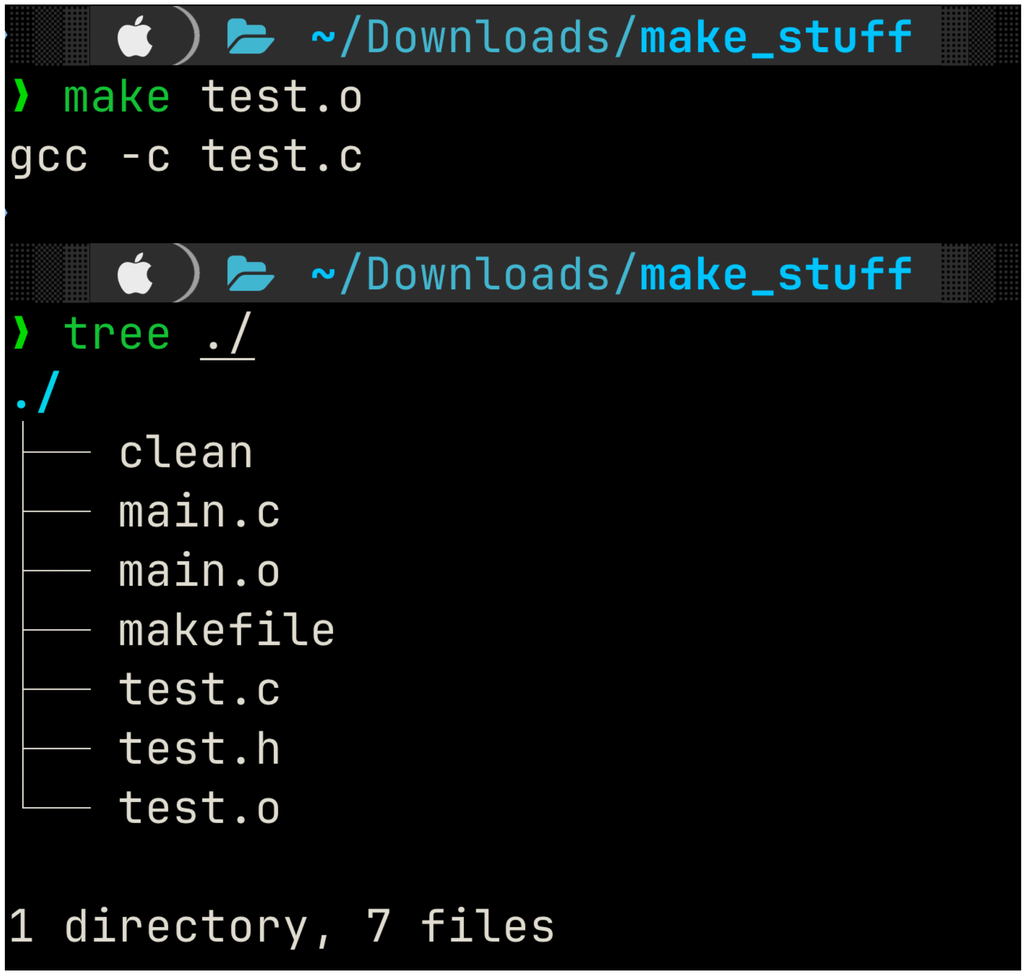
```
~/Downloads/make_stuff                                  took 1m 35s  at 15:24:27
❯ make shout
YEAH!!! MAKE!!! YEAH!!
```

```
tmux (tmux)  ⌘1                                  ..ds/make_stuff (-zsh)  ⌘2
~/Downloads/make_stuff                                                  at 15:25:09
❯ make list
total 40
drwxr-xr-x    7 josephh  1983416439    224 Sep  1 15:17 .
drwx------@ 560 josephh  1983416439  17920 Sep  1 15:25 ..
-rw-r--r--    1 josephh  1983416439      6 Sep  1 14:27 clean
-rw-r--r--@   1 josephh  1983416439     67 Sep  1 14:51 main.c
-rw-r--r--@   1 josephh  1983416439    238 Sep  1 15:16 makefile
-rw-r--r--@   1 josephh  1983416439     67 Sep  1 14:54 test.c
-rw-r--r--@   1 josephh  1983416439    115 Sep  1 14:53 test.h
❯
```

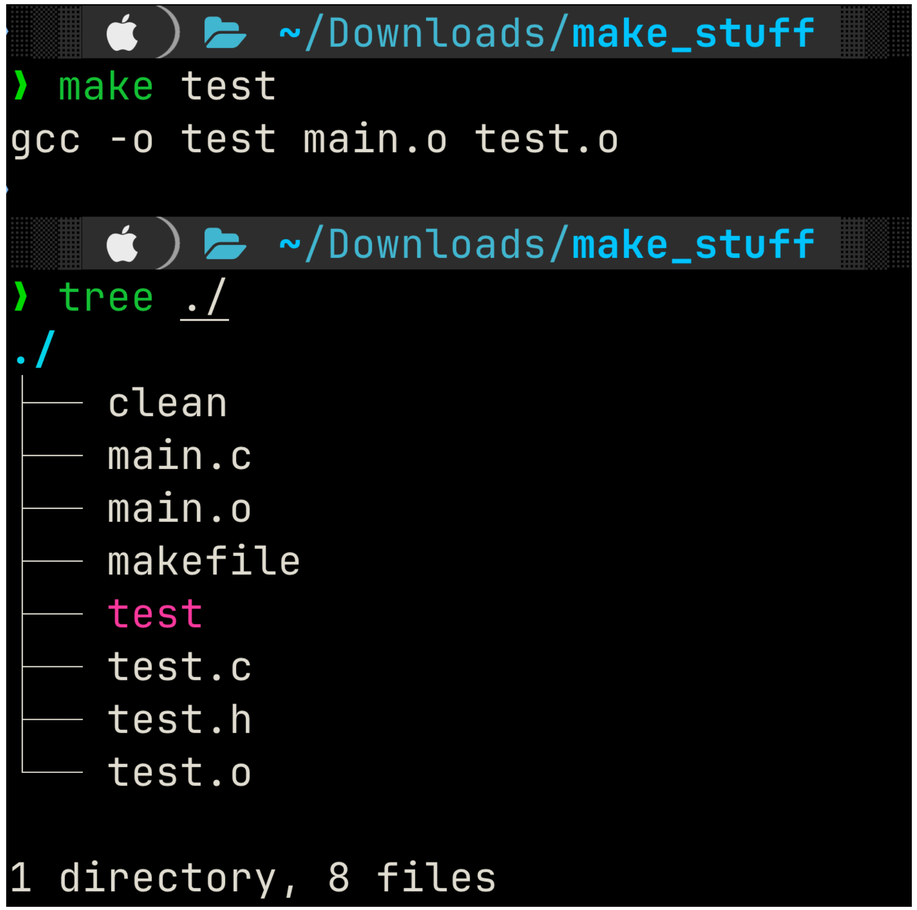
Running *make*



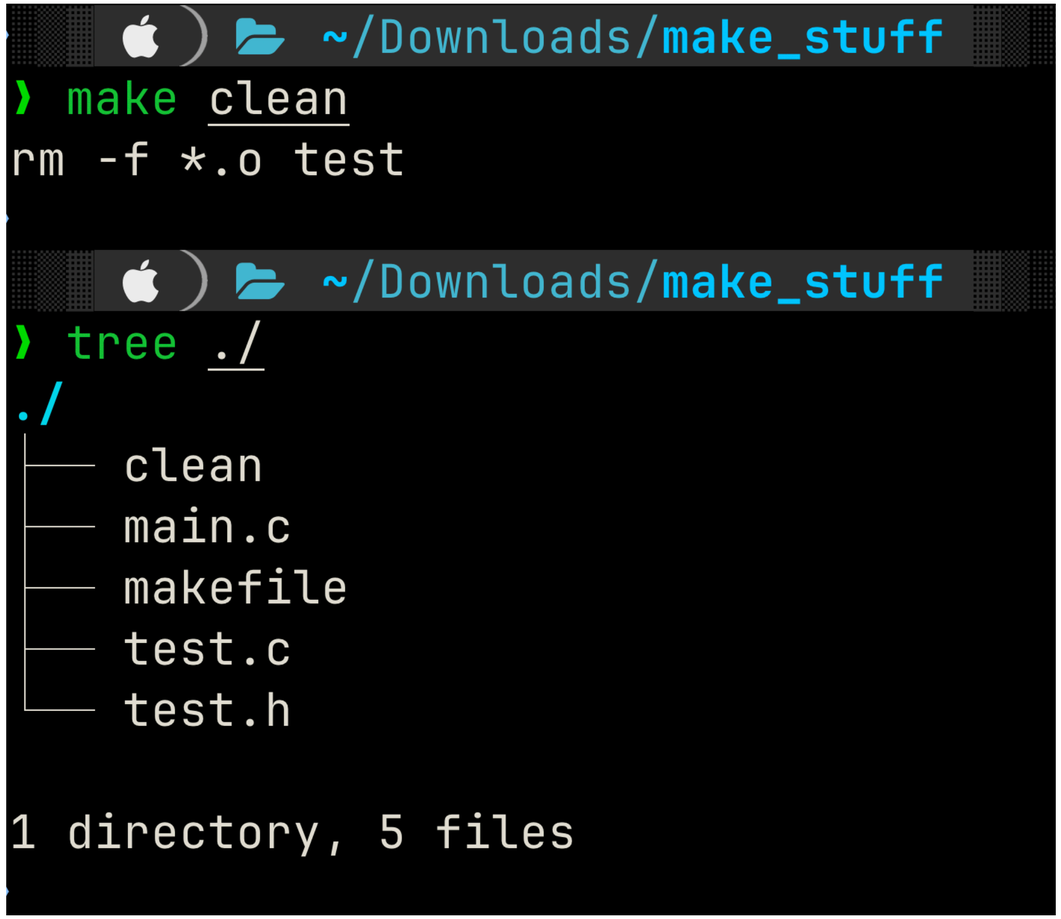
Running *make*



Running *make*



Running *make*



Running *make*

```
~/Downloads/make_stuff
❯ make
gcc -c main.c
gcc -c test.c
gcc -o test main.o test.o
```

```
~/Downloads/make_stuff
❯ tree ./
./
├── clean
├── main.c
├── main.o
├── makefile
├── test
├── test.c
├── test.h
└── test.o

1 directory, 8 files
```

Variables in Make

- Just like with programming the use of variables can make things more flexible, useful, and even re-usable
- You can assign things like .c, .o, or .h file names into variables so that you do not need to type them out multiple times to be used in targets, dependencies, or recipes
- There are 4 ways to assign values to a variable
- The method chosen will determine how the variable is expanded

Recursive Expansion

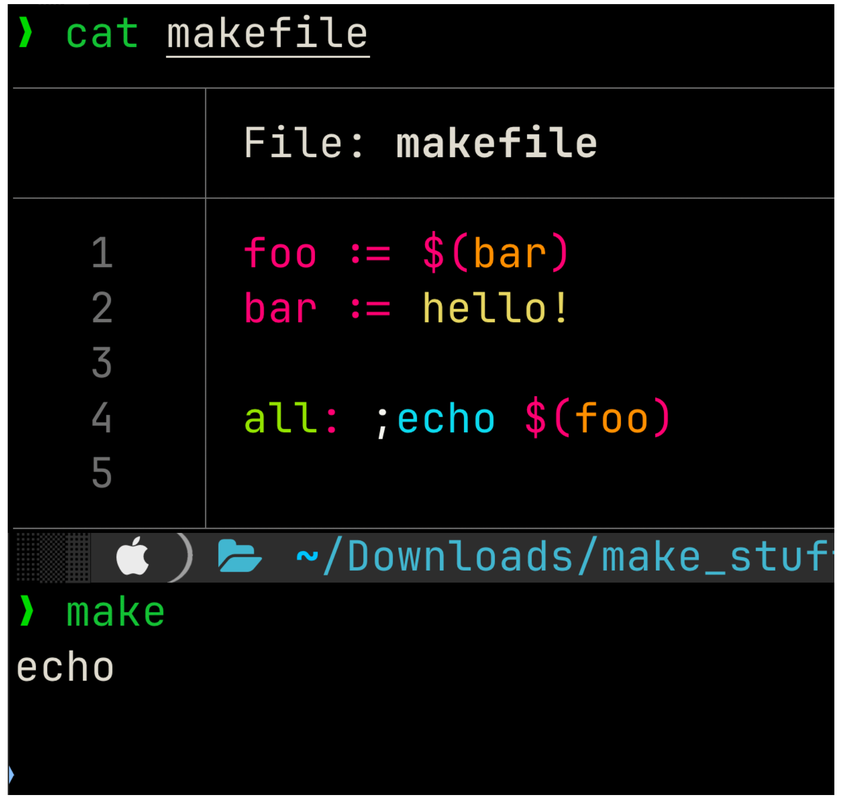
- Expansion used when assigning with '='
- If a var contains a reference to another variable, it will be re-expanded anytime a variable reference is substituted
- Consider the figure on the right
- It is possible to create an infinite loop

```
5 foo = $(bar)
4 bar = $(ugh)
3 ugh = huh?
2
1 all: ;echo $(foo)
```

```
~/Downloads/make_stuf
❯ make all
echo huh?
huh?
❯
```

Simply Expanded Variables

- Expansion used when assigning with ":=" or "::="
- Variables are expanded just once at definition
- If a variable contains a reference to another, then it will be expanded immediately once when defined
- Consider the figure



Simply Expanded Variables

- Expansion used when assigning with ":=" or "::="
- Variables are expanded just once at definition
- If a variable contains a reference to another, then it will be expanded immediately once when defined
- Consider the figure

```
❯ cat makefile
       File: makefile
   1   bar := hello!
   2   foo := $(bar)
   3
   4   all: ;echo $(foo)
   5
❯ make
echo hello!
hello!
```

Condition Expansion

- Expansion used when assigning with "?="
- Variables are expanded only if the variable does not have a value yet
- Consider the figure

File: makefile

```
1  foo := TEST
2  foo ?= hello!
3
4  all: ;echo $(foo)
5
```

```
echo TEST
TEST
```

Condition Expansion

- Expansion used when assigning with "?="
- Variables are expanded only if the variable does not have a value yet
- Consider the figure

Condition Expansion

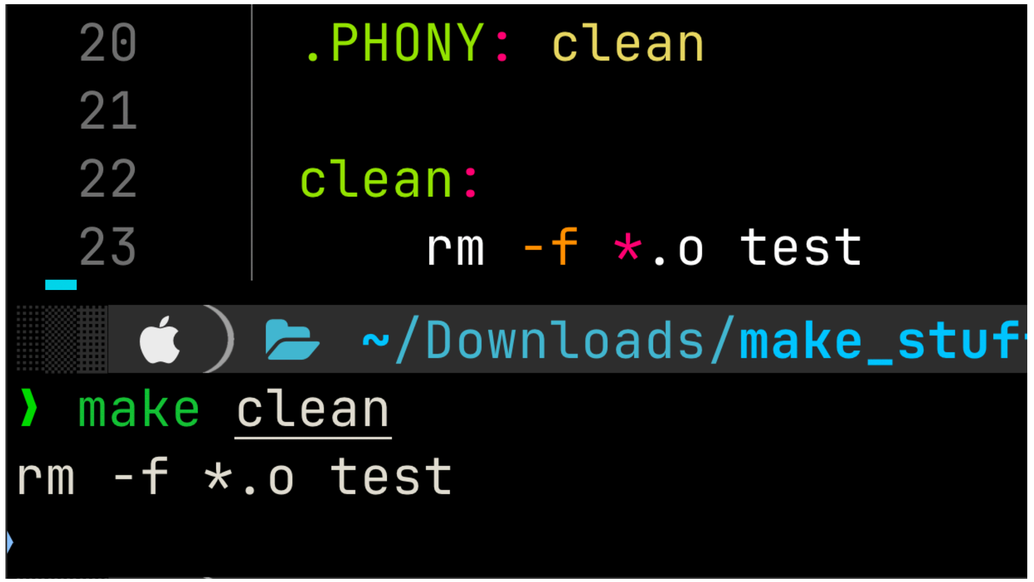
- Expansion used when assigning with "?="
- Variables are expanded only if the variable does not have a value yet
- Consider the figure
- We will not cover the fourth type of variable expansion: Immediately expanded variables ":::="
  - They are expanded at definition as well as expanded recursively

```
       File: makefile
   1   foo := TEST
   2   foo ?= hello!
   3
   4   all: ;echo $(foo)
   5
```

```
echo TEST
TEST
```

Wildcards

- When using terminal commands from your makefile, standard POSIX wildcards work
- The POSIX wildcard '*'
- Often used with the clean target to remove all .o files in the build directory
- *rm -f *.o*
- Wildcards behave as expected when given as deps
- wildcards are not expanded in the same way for make variables



Wildcards

- When using terminal commands from your makefile, standard POSIX wildcards work
- The POSIX wildcard '*'
- Often used with the clean target to remove all .o files in the build directory
- *rm -f *.o*
- Wildcards behave as expected when given as deps
- wildcards are not expanded in the same way for make variables

```
2   objs := main.o test.o
3
4   test: $(objs)
5       gcc -o test $(objs)
6
7   o: *.c
8       gcc -c *.c
9
```

```
~/Downloads/make_stuff
❯ make
cc    -c -o main.o main.c
cc    -c -o test.o test.c
gcc -o test main.o test.o
❯
```

Wildcards in GNU Make

- When using wildcards to store things in variables we must use the make wildcard function. Otherwise the expansion will not work as expected
- This is because "*.c" is not expanded at assignment
- *$(wildcard *.c)* Is how this expansion can be solved.

File: makefile

```
1  sources := $(wildcard *.c)
2  objs := main.o test.o
3
4  test: $(objs)
5      gcc -o test $(objs)
6
7  o: $(sources)
8      gcc -c $(sources)
```

tmux (tmux)

~/Downloads/make_stuff

```
❯ make
cc    -c -o main.o main.c
cc    -c -o test.o test.c
gcc -o test main.o test.o
```

Additional tricks

- *$(patsubst pattern, replacement, $(var))*
- Pattern substitute
  - find a patern, replace that pattern with the substitute, in the expanded variable of reult of function
- Automatic variables
  - '$@' Name of a target in a given rule
  - '$?' or '$^' expanded to the names of all dependencies
  - There are more they can be found from here

```
File: makefile
1  sources := $(wildcard *.c)
2  objs := $(patsubst %.c, %.o, $(sources))
3
4  test: $(objs)
5      gcc -o test $(objs)
6
7  o: $(sources)
8      gcc -c $(sources)
```

```
 ~/Downloads/make_stuff
❯ make
cc    -c -o main.o main.c
cc    -c -o test.o test.c
gcc -o test  main.o  test.o
```

Time to make your own makefiles

- Exercise 3 will require you to make two make files
  - The first is to make a basic file that builds your C project from last week one step at a time
  - The second will be to create a makefile using as compact as possible