

Makefile

Makefile: Motivation

- ▶ Small programs $\longrightarrow$ single file
- ▶ “Not so small” programs :
 - ▶ Many lines of code
 - ▶ Multiple components
 - ▶ More than one programmer

Motivation - continued

► Problems:

- Large files are harder to manage
(for both programmers and machines)
- Every change requires long compilation time
- Many programmers cannot modify the
same file simultaneously
- Division to components is desired

Motivation - continued

- ▶ Solution : divide project to multiple files
- ▶ Targets:
 - ▶ Good division to components
 - ▶ Minimum compilation when something is changed
 - ▶ Easy maintenance of project structure, dependencies and creation

Project maintenance

- ▶ Done in Unix or PC by the Makefile mechanism
- ▶ A **makefile** is a file (script) containing :
 - ▶ Project **structure** (files, **dependencies**)
 - ▶ Instructions for files creation
- ▶ The **make** command reads a makefile, understands the project structure and makes up the executable
- ▶ Note that the Makefile mechanism is **not limited to C or C++ programs**

Filenames

- ▶ When you key in `make`, the `make` looks for the default filenames in the current directory. For GNU `make`, these are:
 - ▶ `GNUMakefile`
 - ▶ `makefile`
 - ▶ `Makefile`
- ▶ If there are more than one of the above in the current directory, the first one according to the above is chosen.
- ▶ It is possible to name the makefile anyway you want, then for `make` to interpret it, you may type:

```
make -f <your-filename>
```

Basic Makefile Format

► Basic Format Summary:

```
# Comments
```

```
VAR=value(s)
```

```
target: list of prerequisites and dependencies  
<tab>    command1 to achieve target using $(VAR)  
[<tab>    command2]
```

Basic Makefile Format (cont.)

▶ Comments

- ▶ Just like for most shell scripts, they start with ‘#’

▶ Variables

- ▶ `var=value`
- ▶ Used as: `$(var)`
- ▶ Variables can be scalar as well as lists (“arrays”)

Targets

- ▶ Target name can be almost anything:
 - ▶ just a name
 - ▶ a filename
 - ▶ a variable
- ▶ There can be several targets on the same line if they depend on the same things
- ▶ A target is followed by
 - ▶ a colon “:”
 - ▶ and then by a list of *dependencies*, separated by a space
- ▶ The default target `make` is looking for is called `all`
- ▶ Another common target is `clean`
 - ▶ Developers supply it to clean up their source tree from temporary files, object modules, etc.
 - ▶ Typical invocation is:
`make clean`

Dependencies

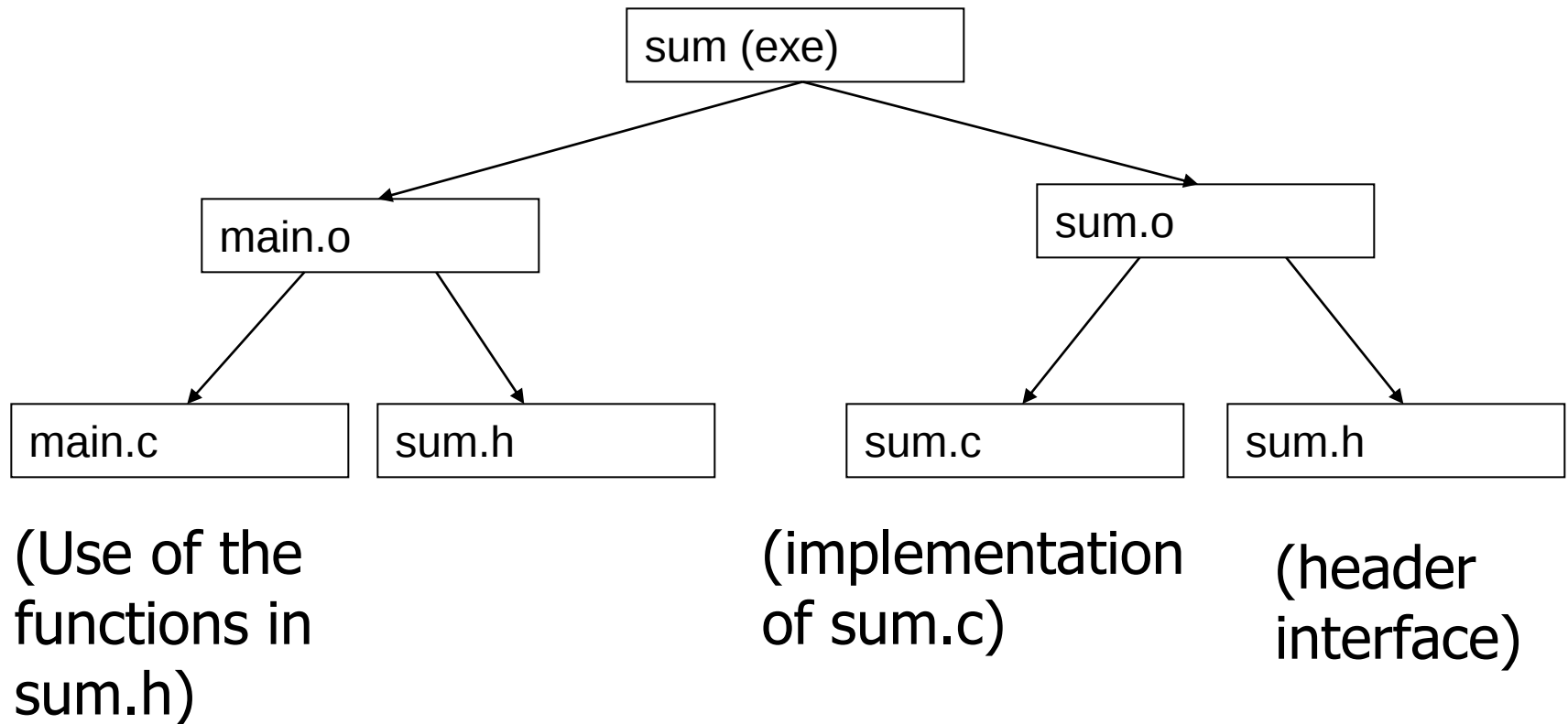
- ▶ The list of dependencies can be:
 - ▶ Filenames
 - ▶ Other target names
 - ▶ Variables
- ▶ Separated by a space
- ▶ May be empty; means “build always”
- ▶ Before the target is built:
 - ▶ it's checked whether it is up-to-date (in case of files) by comparing time stamp of the target of each dependency; if the target file does not exist, it's automatically considered “old”.
 - ▶ If there are dependencies that are “newer” than the target, then the target is rebuilt; else untouched.
 - ▶ If you want to “renew” the target without actually editing the dependencies, “touch” the dependencies with the `touch` command.
 - ▶ If the dependency is a name of another rule, `make` descends recursively to that rule.

Actions

- ▶ A list of actions represents the needed operations to be carried out to arrive to the rule's target.
 - ▶ May be empty.
- ▶ Every action in a rule is usually a typical shell command you would normally type to do the same thing.
- ▶ Every command **MUST** be preceded with a **tab**!
 - ▶ This is how `make` identifies actions as opposed to variable assignments and targets. Do not indent actions with spaces!

Project structure

- ▶ Project structure and dependencies can be represented as a DAG (= Directed Acyclic Graph)
- ▶ Example :
 - ▶ Program contains 3 files
 - ▶ main.c, sum.c, sum.h
 - ▶ sum.h included in both .c files
 - ▶ Executable should be the file sum



makefile

sum: main.o sum.o

gcc -o sum main.o sum.o

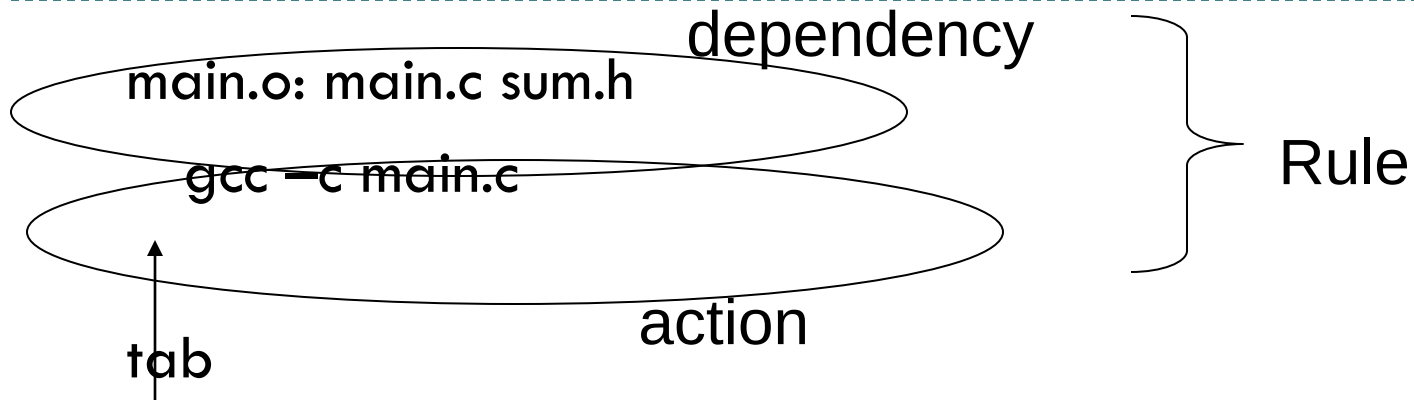
main.o: main.c sum.h

gcc -c main.c

sum.o: sum.c sum.h

gcc -c sum.c # gcc -c sum.c -o sum.o

Rule syntax



Equivalent makefiles

- ▶ .o depends (by default) on corresponding .c file.
Therefore, equivalent makefile is:

```
sum: main.o sum.o
```

```
    gcc -o sum main.o sum.o
```

```
main.o: sum.h
```

```
    gcc -c main.c
```

```
sum.o: sum.h
```

```
    gcc -c sum.c
```

Equivalent makefiles - continued

- ▶ We can compress identical dependencies and use built-in macros to get another (shorter) equivalent makefile :
 - ▶ `$@` is to substitute for the target
 - ▶ `$*` is to take the target prefix element by element

sum: main.o sum.o

gcc -o `$@` main.o sum.o # gcc -o sum main.o sum.o

main.o sum.o: sum.h

gcc -c `$*.c` # gcc -c main.c; gcc -c sum.c

make operation

- ▶ Project dependencies tree is constructed
- ▶ Target of first rule should be created
- ▶ We go down the tree to see if there is a target that should be recreated. This is the case when the target file is older than one of its dependencies
- ▶ In this case we recreate the target file according to the action specified, on our way up the tree. Consequently, more files may need to be recreated
- ▶ If something is changed, linking is usually necessary

make operation - continued

- ▶ make operation ensures **minimum compilation**, when the project structure is written properly

- ▶ **Do not write** something like:

```
prog: main.c sum1.c sum2.c
```

```
gcc -o prog main.c sum1.c sum2.c
```

which requires **compilation of all project** when something is changed

Make operation - example

<u>File</u>	<u>Last Modified</u>
sum	10:03
main.o	09:56
sum.o	09:35
<i>main.c</i>	<i>10:45</i>
sum.c	09:14
sum.h	08:39

Make operation - example

- ▶ Operations performed:

```
gcc -c main.c
```

```
gcc -o sum main.o sum.o
```

- ▶ `main.o` should be **recompiled** (`main.c` is newer).
- ▶ Consequently, `main.o` is newer than `sum` and therefore **sum should be recreated** (by re-linking).

Another makefile example

----- # Makefile to compare sorting routines -----

```
BASE = /home/blufox/base
CC      = gcc
CFLAGS = -O -Wall           # Optimize and issue warnings, if any
EFILE   = $(BASE)/bin/compare_sorts
INCLS   = -I$(LOC)/include
LIBS    = $(LOC)/lib/g_lib.a \
          $(LOC)/lib/h_lib.a
LOC     = /usr/local

OBS = main.o  another_qsort.o  chk_order.o \
      compare.o  quicksort.o

$(EFILE): $(OBS)
    @echo "linking ..."           # only print "linking..." without echo "linking"
    @$$(CC) $(CFLAGS) -o $$@ $(OBS) $(LIBS)  # turn off the printing of this line

$(OBS): compare_sorts.h
    $(CC) $(CFLAGS) $(INCLS) -c $*.c

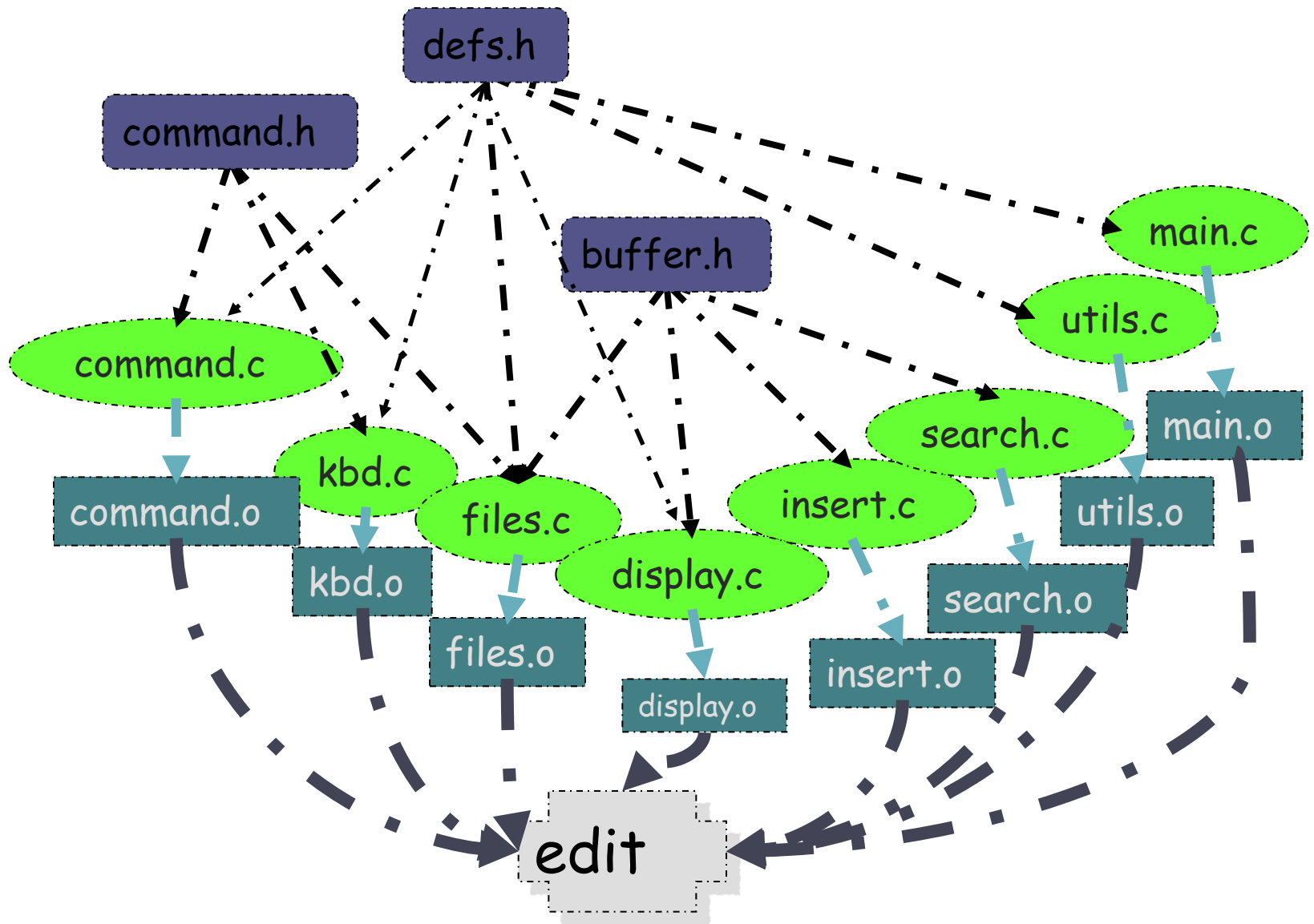
# Clean intermediate files
clean:
    rm *~ $(OBS)
```

Additional Remarks

- ▶ A leading `@` means the command will not be displayed on the screen.
 - ▶ For example, a line `echo "Compiling ..."` displays `echo "Compiling ..."` and `Compiling ...`
 - ▶ BUT `@echo "Compiling ..."` displays `Compiling ...` only
- ▶ We can define **multiple targets** in a makefile
- ▶ Target **clean** - has an empty set of dependencies. Used to clean intermediate files.
- ▶ **make clean**
 - ▶ Will remove intermediate files

Another More Complex Example

- ▶ Suppose you have a text editor which consists of eight C source files
 - ▶ `main.c` includes `defs.h`
 - ▶ `kbd.c` includes `defs.h` and `command.h`
 - ▶ `command.c` includes `defs.h` and `command.h`
 - ▶ `display.c` includes `defs.h` and `buffer.h`
 - ▶ `insert.c` includes `defs.h` and `buffer.h`
 - ▶ `search.c` includes `defs.h` and `buffer.h`
 - ▶ `files.c` includes `defs.h` and `command.h` and `buffer.h`
 - ▶ `utils.c` includes `defs.h`



```

edit : main.o kbd.o command.o display.o \
      insert.o search.o files.o utils.o
      gcc -o edit main.o kbd.o command.o \
          display.o insert.o \
          search.o files.o utils.o

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

```

make
make edit
make main.o
make clean

Variables

- ▶ Allow us to define variables to make files more flexible and to avoid certain errors.
- ▶ Consider:

```
edit : main.o kbd.o command.o display.o \  
      insert.o search.o files.o utils.o  
gcc -o edit main.o kbd.o command.o \  
      display.o insert.o \  
      search.o files.o utils.o
```

Variables

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

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

```
...
```

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

Cleaning

- ▶ Here was how we wrote a make rule for cleaning our example editor:

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