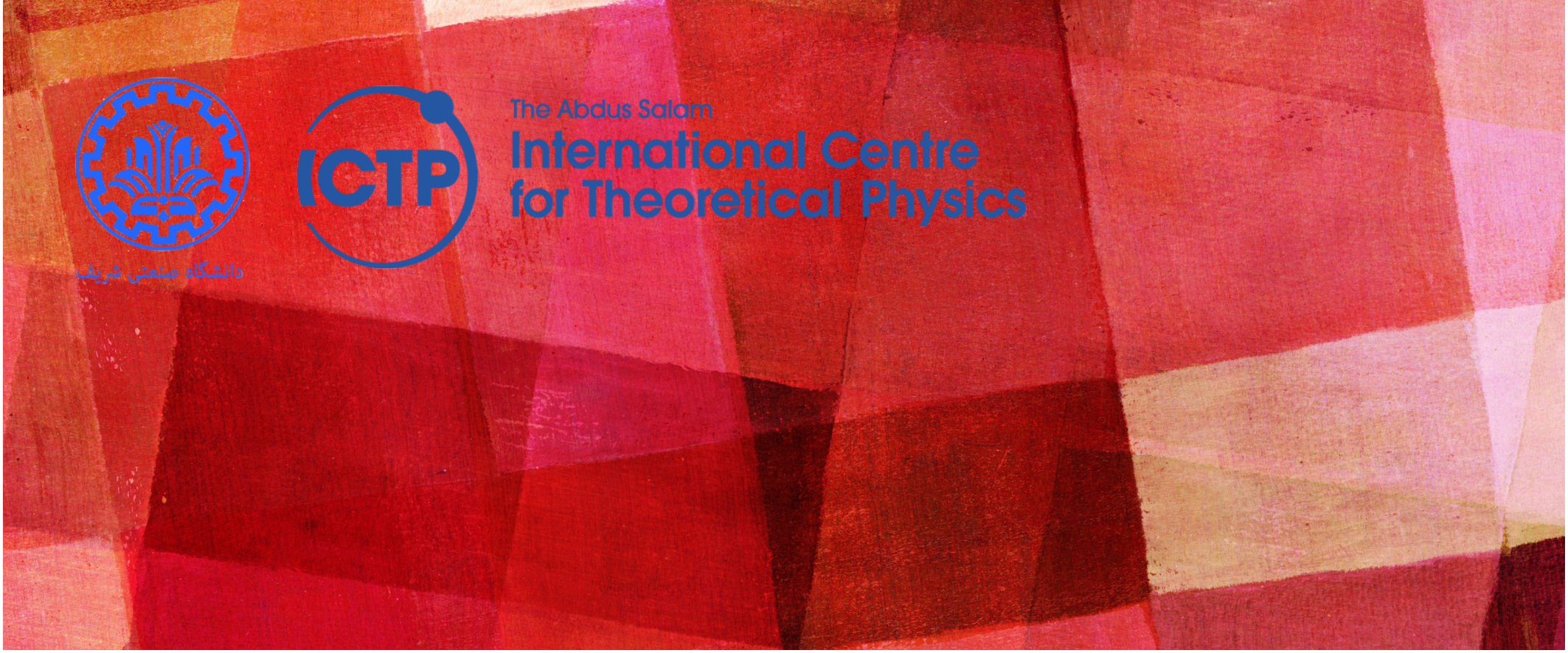


THE MAKEFILE UTILITY

Ali Farnudi

MOTIVATION

small programme

MOTIVATION

small programme

Small file

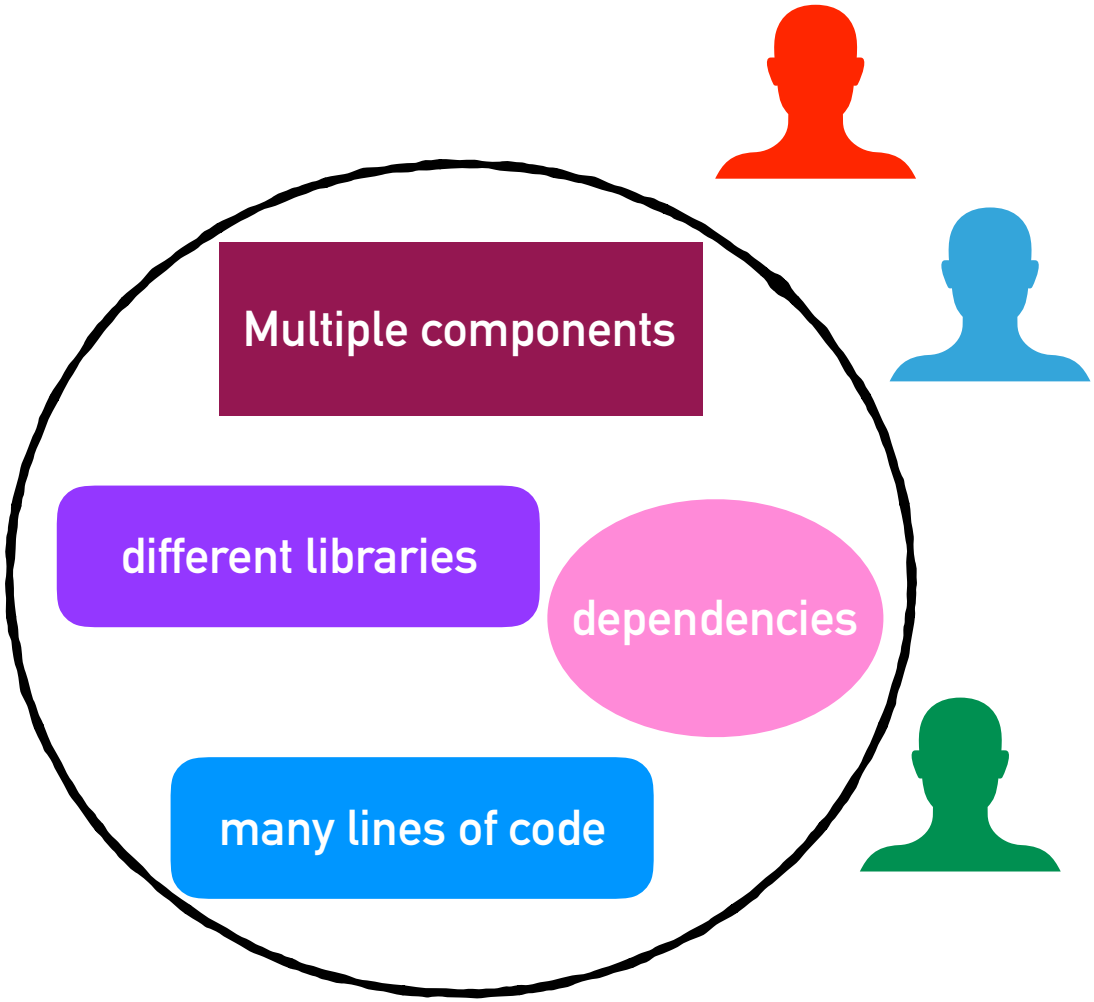
'Not so small' programme

MOTIVATION

small programme

Small file

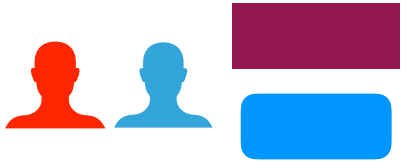
'Not so small' programme



MOTIVATION

small programme

'Not so small' programme



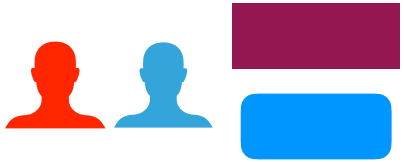
PROBLEMS

- long files are harder to manage for **both** programmer and the machine
- A **big file** means long compilation for a small change
- If possible, its **very confusing** for several programmers to simultaneously modify a file.

MOTIVATION

small programme

'Not so small' programme

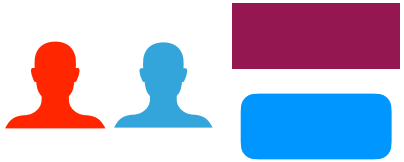


SOLUTION: DIVIDE PROJECT TO MULTIPLE FILES (TARGETS)

MOTIVATION

small programme

'Not so small' programme



SOLUTION: DIVIDE PROJECT TO MULTIPLE FILES (TARGETS)

- **Divide** the code to **components**
- **Minimum compilation** when **something is changed**
- **Easy maintenance** of project **structure**, **dependencies** and **creation**

PROJECT MAINTENANCE

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

PROJECT MAINTENANCE

Project ***structure*** *and* ***dependencies*** *can be represented as a '**Directed Acyclic Graph**'*

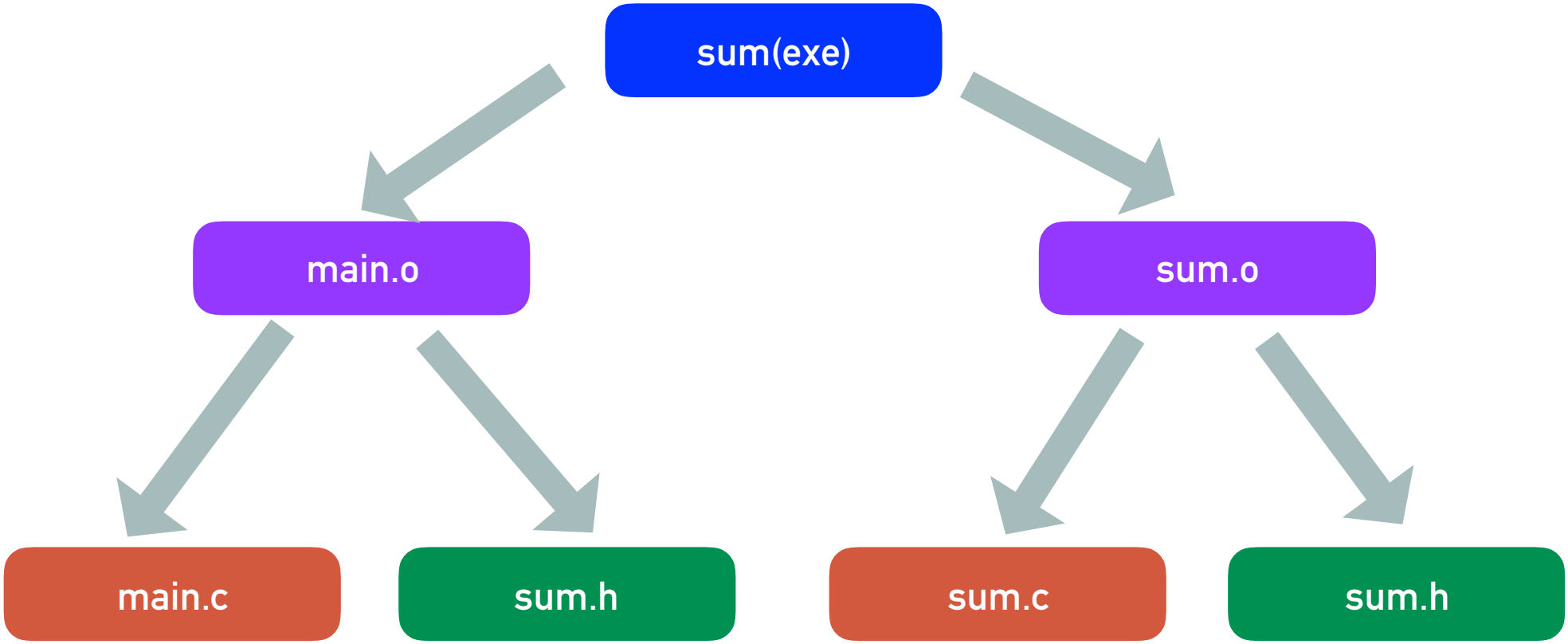
PROJECT MAINTENANCE

*Project **structure** and **dependencies** can be represented as a '**Directed Acyclic Graph**'*

- Example :
  - Program contains **3 files**
  - **main.c**, **sum.c**, **sum.h**
  - **sum.h** is included in **both** '**.c**' files
  - Executable should be the file '**sum**'

PROJECT MAINTENANCE

Directed Acyclic Graph



MAKEFILE

filename: makefile

Target: Dependancie

←tab→ rules

Target02: Dependancie

←tab→ rules

> cd ‘dir’ of makefile
> make

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
```

Target: Dependancie

←tab→ rules

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
```

EQUIVALENT MAKEFILES

- **‘.o’** depends (**by default**) on corresponding **‘.c’** file.

We may **rewrite** the **makefile** in this **equivalent** format:

EQUIVALENT MAKEFILES

➤ **'.o'** depends (**by default**) on corresponding **'.c'** file.

We may **rewrite** the **makefile** in this **equivalent** format:

```
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

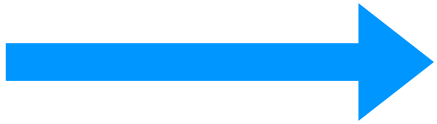
- We can compress **identical dependencies** and use **built-in macros** to get another (shorter) **equivalent makefile**:

```
sum: main.o sum.o
    gcc –o sum main.o sum.o
```

```
sum: main.o sum.o
    gcc –o $@ main.o sum.o
```

```
main.o: sum.h
    gcc –c main.c

sum.o: sum.h
    gcc –c sum.c
```



```
main.o sum.o: sum.h
    gcc –c $*.c
```

MAKE OPERATION

- Project **dependencies tree** is **constructed**
- Target of the **first** rule should be created

- We go **down the tree** to see if there is a **target** that should be **recreated**.
  - This is required 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 was changed, **linking is performed**

MAKE OPERATION

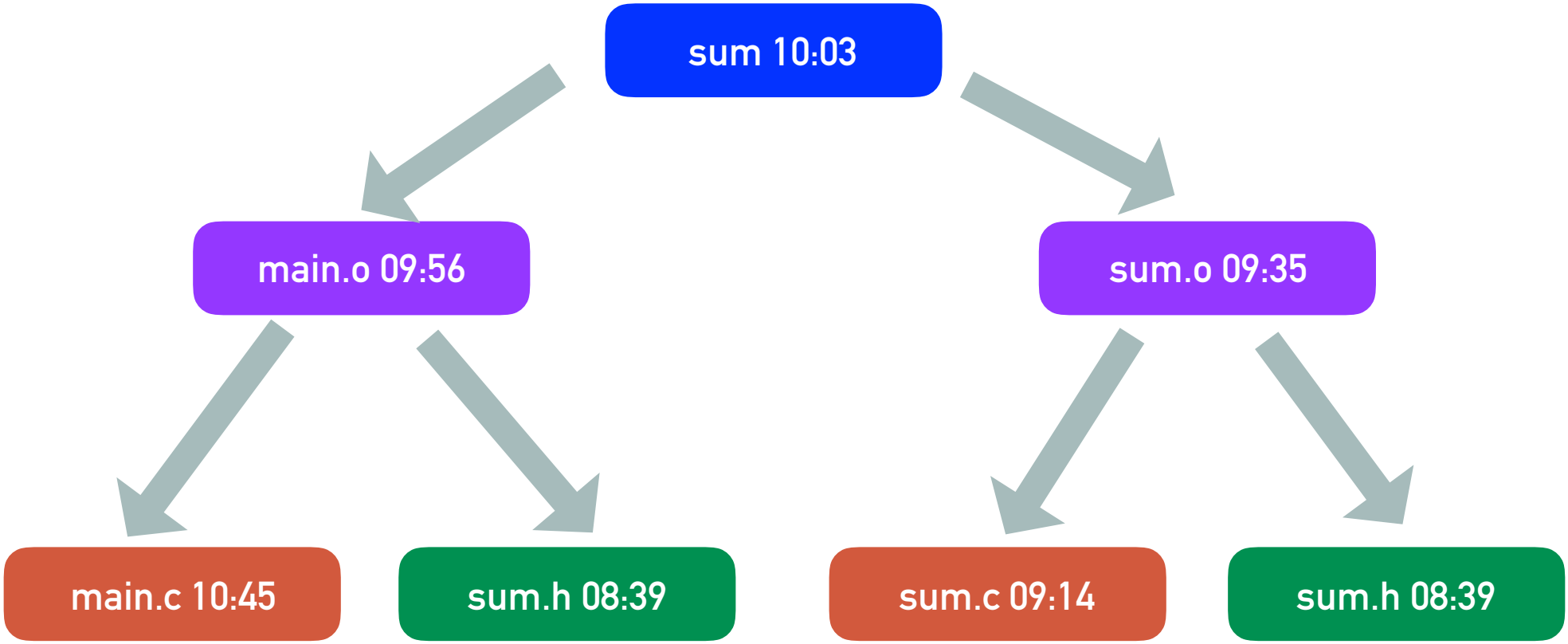
- Project **dependencies tree** is **constructed**
- Target of the **first** rule should be created

- We go **down the tree** to see if there is a **target** that should be **recreated**.
  - This is required 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 was changed, **linking is performed**

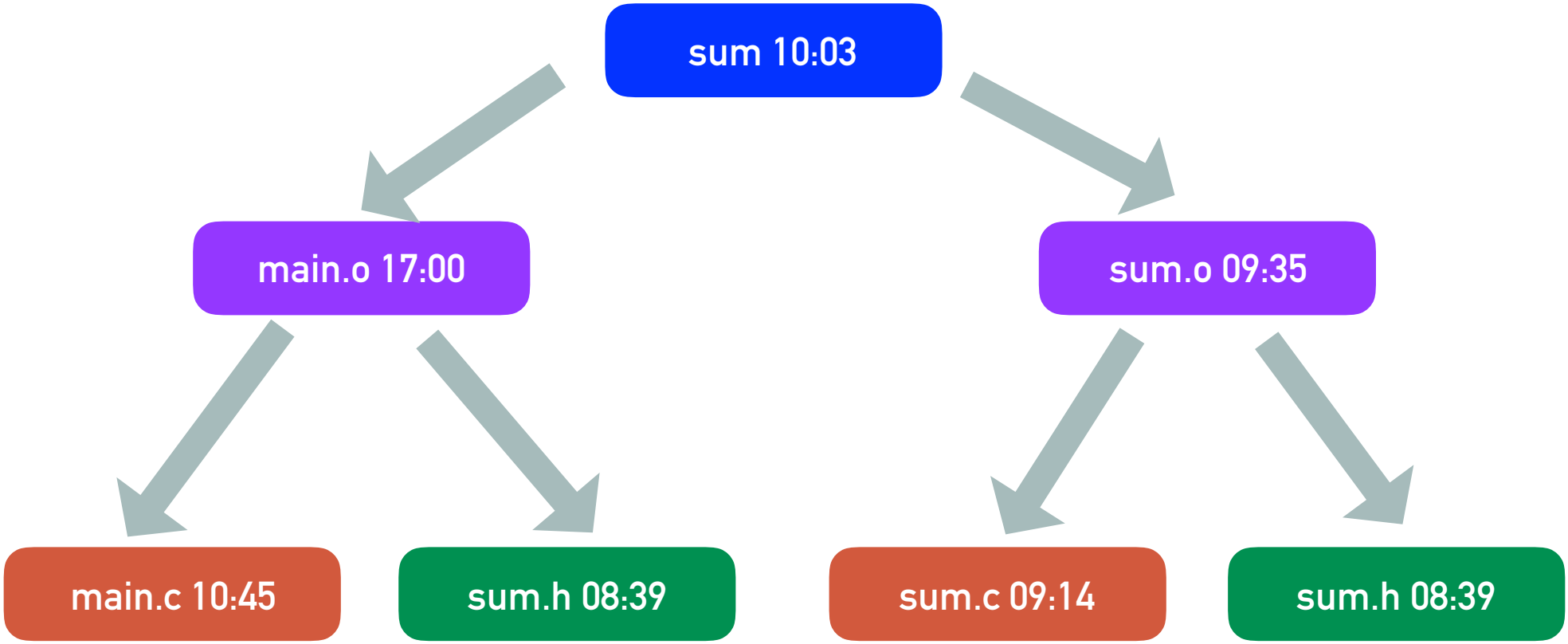
PROJECT MAINTENANCE

Directed Acyclic Graph



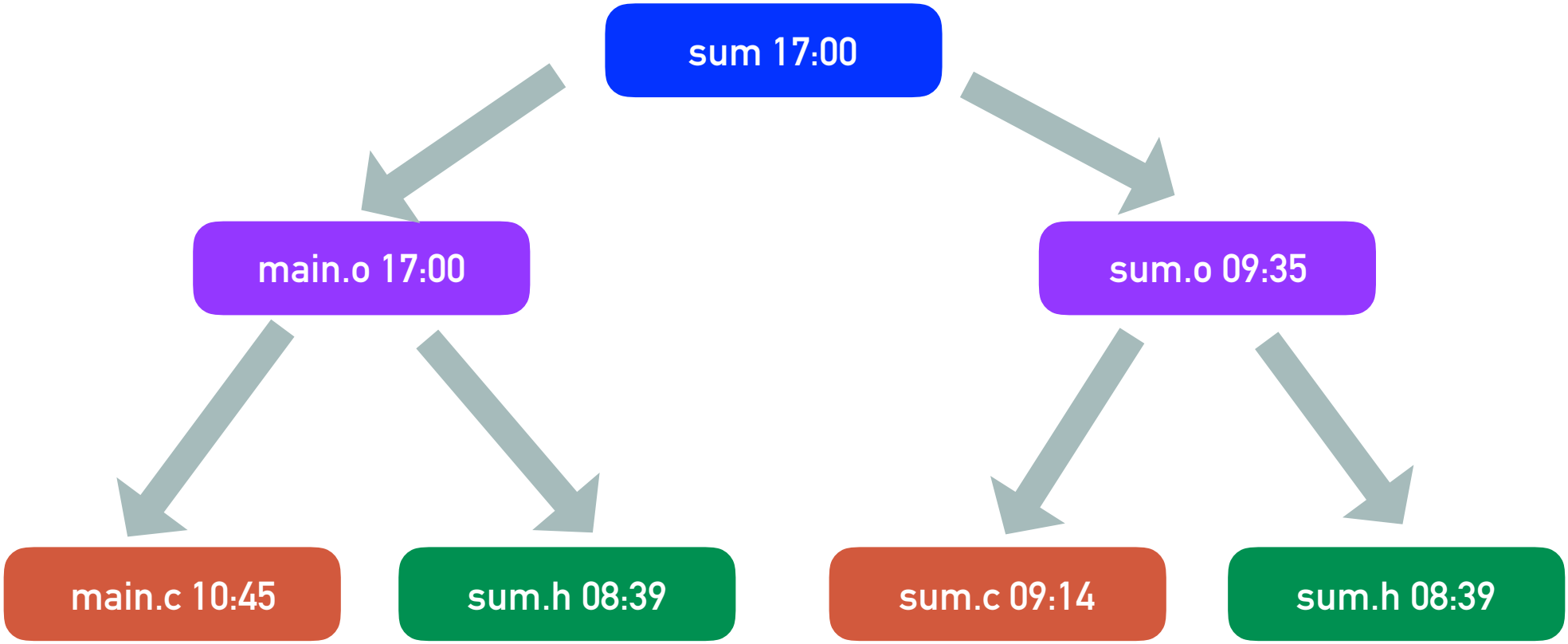
PROJECT MAINTENANCE

Directed Acyclic Graph



PROJECT MAINTENANCE

Directed Acyclic Graph



MAKE OPERATION

- '**make**' operation can be used to ensure **minimum compilation**, when the project structure is **written properly**

- **Do not write this at home**:

```
prog: main.c sum1.c sum2.c
    gcc –o prog main.c sum1.c sum2.c
```

USEFUL GCC OPTIONS

- Include: -I<path>
- Define: -D<identifier>
- Optimization: -O<level>

USEFUL GCC OPTIONS

- Include: -I<path>
- Define: -D<identifier>
- Optimization: -O<level>

Example:

gcc –DDEBUG –O2 –I/usr/include example.c –o example -lm

ANOTHER EXAMPLE

```
# Makefile to compare sorting routines
BASE    = /home/blufox/base
CC      =  gcc
CFLAGS  =  -O –Wall
EFILE   =  $(BASE)/bin/compare_sorts
INCLS   =  -I$(LOC)/include
LIBS    =  $(LOC)/lib/g_lib.a \
           $(LOC)/lib/h_lib.a
LOC     =  /usr/local
```

ANOTHER EXAMPLE

```
# Makefile to compare sorting routines
BASE     = /home/blufox/base
CC       =  gcc
CFLAGS   =  -O –Wall
EFILE    =  $(BASE)/bin/compare_sorts
INCLS    =  -I$(LOC)/include
LIBS     =  $(LOC)/lib/g_lib.a \
            $(LOC)/lib/h_lib.a
LOC      =  /usr/local
OBJS = main.o   another_qsort.o   chk_order.o \
       compare.o   quicksort.o

$(EFILE): $(OBJS)
    @echo “linking …”
    @$(CC) $(CFLAGS) –o $@ $(OBJS) $(LIBS)

$(OBJS): compare_sorts.h
    $(CC) $(CFLAGS) $(INCLS) –c $*.c

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

```
# Makefile to compare sorting routines
BASE     = /home/blufox/base
CC       =  gcc
CFLAGS   =  -O –Wall
EFILE    =  $(BASE)/bin/compare_sorts
INCLS    =  -I$(LOC)/include
LIBS     =  $(LOC)/lib/g_lib.a \
            $(LOC)/lib/h_lib.a
LOC      =  /usr/local
OBJS = main.o   another_qsort.o   chk_order.o \
       compare.o   quicksort.o

$(EFILE): $(OBJS)
    @echo "linking ..."
    @$(CC) $(CFLAGS) –o $@ $(OBJS) $(LIBS)

$(OBJS): compare_sorts.h
    $(CC) $(CFLAGS) $(INCLS) –c $*.c

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

- 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
- make will create the 'compare_sorts' executable

```
# Makefile to compare sorting routines
BASE     = /home/blufox/base
CC       =  gcc
CFLAGS   =  -O –Wall
EFILE    =  $(BASE)/bin/compare_sorts
INCLS    =  -I$(LOC)/include
LIBS     =  $(LOC)/lib/g_lib.a \
            $(LOC)/lib/h_lib.a
LOC      =  /usr/local
OBJS = main.o   another_qsort.o   chk_order.o \
       compare.o   quicksort.o

$(EFILE): $(OBJS)
    @echo "linking ..."
    @$(CC) $(CFLAGS) –o $@ $(OBJS) $(LIBS)

$(OBJS): compare_sorts.h
    $(CC) $(CFLAGS) $(INCLS) –c $*.c

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

- 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
- make will create the **'compare_sorts'** executable

GET MORE OUT OF MAKEFILE

http://www.gnu.org/software/make/manual/make.html