Using `make`

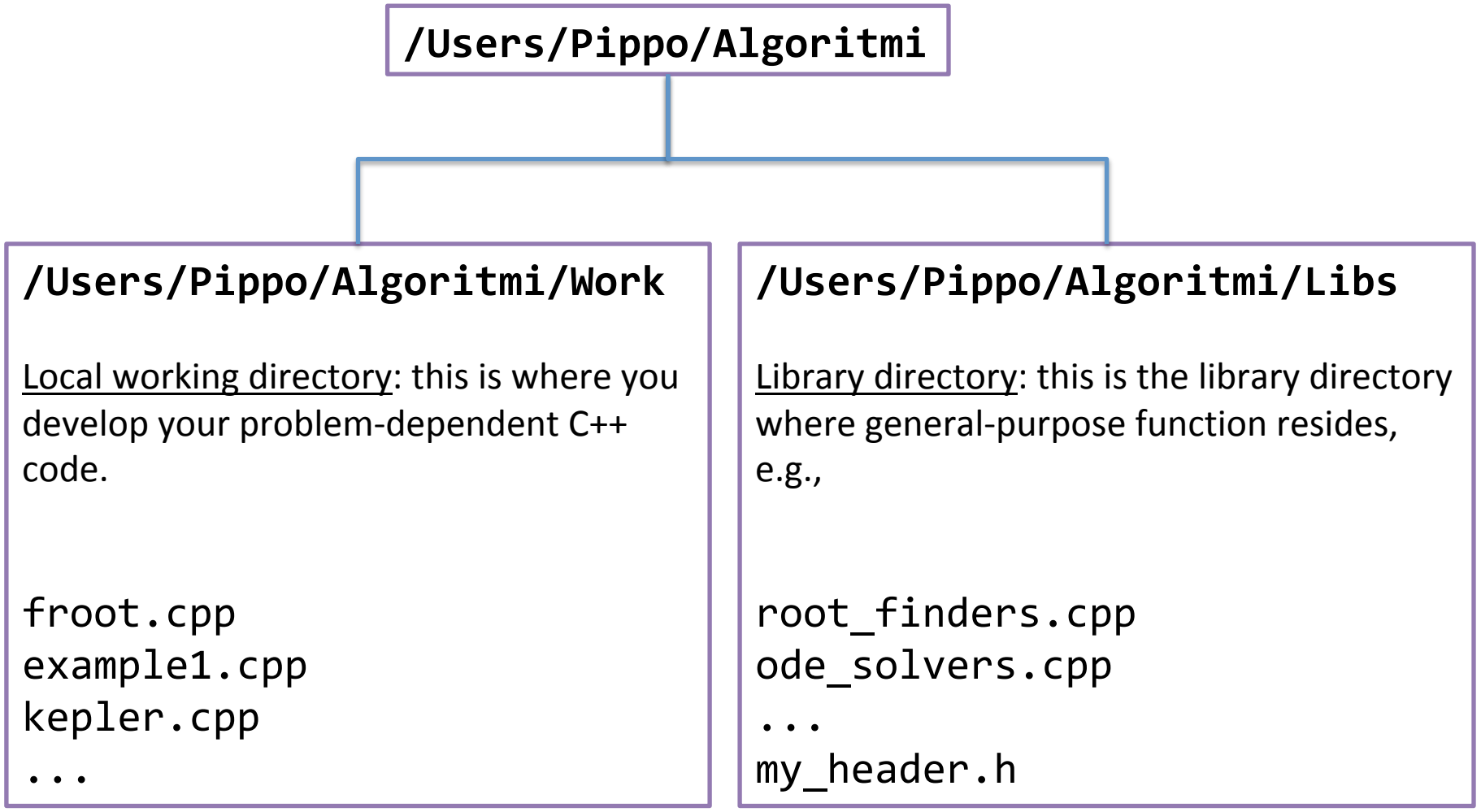
Andrea Mignone
Physics Department, University of Torino
AA 2019-2020

Makefile

- Makefiles are a simple way to organize code compilation.
- With a `makefile` it is possible to compile several source files to produce an executable;
- Source (.cpp) and header (.h) files can be placed in different directories.

An example of code structure

- User ‘Pippo’ has the following directory structure:



Understanding a `makefile`

- The `makefile` is a text file that contains the recipe for building your program.
- It usually resides in the same directory as the sources, and it is usually called "`makefile`" or "`Makefile`"(without any extension).
- Instruction in a `makefile` are called *rules*: a rule is an instruction for building one or more output files from one or more input files.
- Make determines which rules need to be re-executed by checking whether any of the input files has changed since the last time the rule was executed.
- A *rule* has a syntax like this:

```
output_file: input_file
    <actions>
```

How "Rules" Work

```
output_file: input_file
    <actions>
```

- The first line of the rule contains a space-separated list of output files, followed by a colon, followed by a space-separated list of input files.

- The *output files* are also called *targets*, and the *input files* are also called *dependencies*;
- We say that the target file depends on the dependencies, because if any of the dependencies change, the target must be rebuilt.

- The remaining lines of the rule (the actions) are shell commands to be executed.
- **Each action must be indented with a tab character**. Usually, there's just one action line, but there can be as many as you want; each line is executed sequentially, and if any one of them fails, the remainder are not executed. The rule ends at the first line which is not indented.

The `makefile`

- You should create a new text file, named "`makefile`" (no extensions), using your editor of choice.
- This is how a typical (simple) makefile looks like:

```
KEPLER_OBJ = kepler.o  ode_solvers.o

CXX     = g++
CFLAGS = -c
SRC     = $(HOME)/Didattica/Algoritmi_Numerici/Lib
VPATH   = ./:$(SRC)
INCLUDE_DIRS = -I. -I$(SRC)
LDFLAGS      = -lm

kepler:  $(KEPLER_OBJ)
        $(CXX) $(KEPLER_OBJ) $(LDFLAGS) -o $@

%.o: %.cpp
        $(CXX) $(CFLAGS) $(INCLUDE_DIRS) $<
```

```
KEPLER_OBJ = kepler.o  ode_solvers.o

CXX     = g++
CFLAGS = -c
SRC     = $(HOME)/Didattica/Algoritmi_Numerici/Lib
VPATH   = ./:$(SRC)
INCLUDE_DIRS = -I. -I$(SRC)
LDFLAGS      = -lm

kepler:  $(KEPLER_OBJ)
        $(CXX) $(KEPLER_OBJ) $(LDFLAGS) -o $@

%.o: %.cpp
        $(CXX) $(CFLAGS) $(INCLUDE_DIRS) $<
```

KEPLER_OBJ: a list of all the object files that must be linked together to produce the executable

```
KEPLER_OBJ = kepler.o  ode_solvers.o

CXX     = g++
CFLAGS = -c
SRC     = $(HOME)/Didattica/Algoritmi_Numerici/Lib
VPATH   = ./:$(SRC)
INCLUDE_DIRS = -I. -I$(SRC)
LDFLAGS      = -lm

kepler:  $(KEPLER_OBJ)
        $(CXX) $(KEPLER_OBJ) $(LDFLAGS) -o $@

%.o: %.cpp
        $(CXX) $(CFLAGS) $(INCLUDE_DIRS) $<
```

CXX: the name of the C++ compiler (or others) used to compile source codes

```
KEPLER_OBJ = kepler.o  ode_solvers.o

CXX     = g++
CFLAGS = -c
SRC     = $(HOME)/Didattica/Algoritmi_Numerici/Lib
VPATH   = ./:$(SRC)
INCLUDE_DIRS = -I. -I$(SRC)
LDFLAGS      = -lm

kepler:  $(KEPLER_OBJ)
        $(CXX) $(KEPLER_OBJ) $(LDFLAGS) -o $@

%.o: %.cpp
        $(CXX) $(CFLAGS) $(INCLUDE_DIRS) $<
```

CFLAGS: list of flags to pass to the compilation command. Here -c means "compile only and produce object file .o".

```
KEPLER_OBJ = kepler.o  ode_solvers.o

CXX     = g++
CFLAGS  = -c
SRC     = $(HOME)/Didattica/Algoritmi_Numerici/Lib
VPATH   = ./:$(SRC)
INCLUDE_DIRS = -I. -I$(SRC)
LDFLAGS      = -lm

kepler:  $(KEPLER_OBJ)
        $(CXX) $(KEPLER_OBJ) $(LDFLAGS) -o $@

%.o: %.cpp
        $(CXX) $(CFLAGS) $(INCLUDE_DIRS) $<
```

SRC: location of the main source directory, where all of yours library routines are placed. This is *not* the local working directory.
Environment variables that make sees when it starts up is transformed into a make variable with the same name and value.

```
KEPLER_OBJ = kepler.o  ode_solvers.o

CXX     = g++
CFLAGS = -c
SRC     = $(HOME)/Didattica/Algoritmi_Numerici/Lib
VPATH   = ./:$(SRC)
INCLUDE_DIRS = -I. -I$(SRC)
LDFLAGS      = -lm

kepler:  $(KEPLER_OBJ)
        $(CXX) $(KEPLER_OBJ) $(LDFLAGS) -o $@

%.o: %.cpp
        $(CXX) $(CFLAGS) $(INCLUDE_DIRS) $<
```

VPATH: special name used by GNU Make to specify a list of directories that make should search. Thus, if a file that is listed as a target or dependency does not exist in the current directory, make searches the directories listed in VPATH for a file with that name.

```
KEPLER_OBJ = kepler.o  ode_solvers.o

CXX     = g++
CFLAGS = -c
SRC     = $(HOME)/Didattica/Algoritmi_Numerici/Lib
VPATH   = ./:$(SRC)
INCLUDE_DIRS = -I. -I$(SRC)
LDFLAGS       = -lm

kepler:  $(KEPLER_OBJ)
        $(CXX) $(KEPLER_OBJ) $(LDFLAGS) -o $@

%.o: %.cpp
        $(CXX) $(CFLAGS) $(INCLUDE_DIRS) $<
```

INCLUDE_DIRS: specifies the directories to be searched for header files.
Note the usage of "-I"

```
KEPLER_OBJ = kepler.o  ode_solvers.o

CXX     = g++
CFLAGS = -c
SRC     = $(HOME)/Didattica/Algoritmi_Numerici/Lib
VPATH   = ./:$(SRC)
INCLUDE_DIRS = -I. -I$(SRC)
LDFLAGS      = -lm

kepler:  $(KEPLER_OBJ)
        $(CXX) $(KEPLER_OBJ) $(LDFLAGS) -o $@

%.o: %.cpp
        $(CXX) $(CFLAGS) $(INCLUDE_DIRS) $<
```

`kepler`: this is the main target. It tells that the executable must be built from the object file list specified by `$(KEPLER_OBJ)`. The second line is the actual command to be used to accomplish the target.
The "`$@`" says to put the output of the compilation in the file named on the left side of the ":"

```
KEPLER_OBJ = kepler.o  ode_solvers.o

CXX     = g++
CFLAGS = -c
SRC     = $(HOME)/Didattica/Algoritmi_Numerici/Lib
VPATH   = ./:$(SRC)
INCLUDE_DIRS = -I. -I$(SRC)
LDFLAGS      = -lm

kepler:  $(KEPLER_OBJ)
        $(CXX) $(KEPLER_OBJ) $(LDFLAGS) -o $@

%.o: %.cpp
        $(CXX) $(CFLAGS) $(INCLUDE_DIRS) $<
```

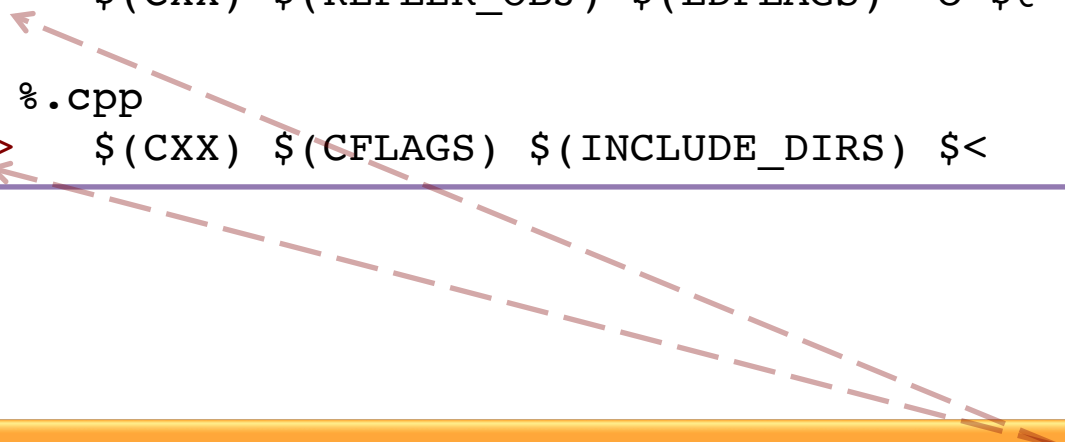
`%.o: %.cpp`: this is the suffix rule. It instruct how to create an object file (.o) from a source file (`.cpp`). The "`$<`" is the first item in the dependencies list

```
KEPLER_OBJ = kepler.o  ode_solvers.o

CXX     = g++
CFLAGS = -c
SRC     = $(HOME)/Didattica/Algoritmi_Numerici/Lib
VPATH   = ./:$(SRC)
INCLUDE_DIRS = -I. -I$(SRC)
LDFLAGS      = -lm

kepler:  $(KEPLER_OBJ)
<tab>    $(CXX) $(KEPLER_OBJ) $(LDFLAGS) -o $@

%.o: %.cpp
<tab>    $(CXX) $(CFLAGS) $(INCLUDE_DIRS) $<
```



!VERY IMPORTANT : actions must be preceded by a single <tab> character and not spaces !!!

Compiling the Code

- Now that you have built the makefile, simply type

```
> make
```

or, if you have more than one target,

```
> make kepler
```

- If your program has already been built and no changes were made, make will tell you that nothing has to be done.

The make clean target

- In some case one would like to “clean” and rebuild the target from scratch.
- A way to achieve this is having make delete all the *.o files.
- A simple rule can be added to the purpose:

```
KEPLER_OBJ = kepler.o  ode_solvers.o

CXX     = g++
CFLAGS = -c
SRC     = $(HOME)/Didattica/Algoritmi_Numerici/Lib
VPATH   = ./:$(SRC)
INCLUDE_DIRS = -I. -I$(SRC)
LDFLAGS       = -lm

kepler:  $(KEPLER_OBJ)
<tab>    $(CXX) $(KEPLER_OBJ) $(LDFLAGS) -o $@

%.o: %.cpp
<tab>    $(CXX) $(CFLAGS) $(INCLUDE_DIRS) $<

clean:
<tab>    @rm -f  *.o
<tab>    @echo make clean: done
```