Advanced Programming for Scientific Computing (PACS)
Lecture title: Make and makefiles

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2023/2024

Controlling the compilation process

The compilation process requires assembling data from different interrelated sources:

- A translation unit may depend on several header files;
- Several compilation units may make up a library;
- The executable may depend on libraries, as well as source, header and object files.

The use of make automatizes this process by defining prerequisite-to-target rules.

What is in fact make making?

The make utility is a tool to produce files according to predefined or user defined rules.

It is mainly used in conjunction with the compilation process, yet it can be adopted in any context where files are produced from other files according to well-defined rules.

The rules are written on a file, usually called makefile or Makefile, but you can specify another file using the `-f` option.).

A simple example (not related to programming)

A `makefile` with these simple instructions can be used to produce a PostScript document or a PDF document from the LaTeXfile `lecture.tex`

```
lecture.ps: lecture.dvi
<TAB> dvips lecture.dvi
lecture.pdf: lecture.dvi
<TAB> dvipdf lecture.dvi
lecture.dvi: lecture.tex
<TAB> latex lecture.tex
```

The command `make lecture.pdf` will produce the file lecture.pdf from the file lecture.dvi, the latter created according to the `tex`→`dvi` rule. `make lecture.dvi` will just produce the intermediate `dvi` file output by LaTeX.

Another simple example (related to programming)

```
main: main.o funct.o
<TAB> g++ -O2 -o main main.o funct.o
funct.o: funct.cpp funct.hpp other.hpp
<TAB> g++ -c -O2 funct.cpp
main.o: main.cpp
<TAB> g++ -c -O2 main.cpp
```

The command `make main` will produce the `main` file, by first compiling `main.cpp` and `funct.cpp`, and then linking the corresponding object files.

The basic layout of a makefile

```
target1: prerequisites1
<TAB> command
target2: prerequisites2
<TAB> command
<TAB> command
```

The <TAB> symbol indicates the tab keystroke (the one normally at the upper-left side of your keyboard). You will not see it, of course, since it translates to a series of spaces yet it MUST be there, as it identifies the lines containing commands.

Make sure your editor is not configured to replace tabs with spaces, when editing a Makefile.

Some nomenclature

- A target is either the name of a file that has to be generated (e.g. executables or object files), or the name of an action to be carried out (*phony target*).
- A prerequisite is a file or an action required to produce the target. More prerequisites are separated by a space.
- A command is the statement (e.g. a shell command or an executable) that make launches whenever the target is out-of-date w.r.t. the prerequisites. A command line ALWAYS starts with a <TAB>;
- A rule is a list of commands, each on its own line;
- A directive is the full set of instructions that indicate how to make a specific target.

How does it work?

Launching `make target1` (or simply `make` if `target1` is the first target) the command operates recursively, using the following algorithm:

- Launch make using as target the prerequisites of `target1`;
- *Return* if no rule for the current target is found;
- Check whether the target file has *an earlier modification date* than any of the prerequisites: if so run the command(s) associated to the rule.

Not finding any rule for a target is an error.

Simplifying your making: variables

In a makefile you can define variables

```
OBJECTS = pippo.o toto.o foo.o \
main.o
SOURCES = pippo.c toto.c foo.c \
main.c
EXEC = main

$(EXEC) : $(OBJECTS)
    g++ $(OBJECTS) -o $(EXEC)
```

Note: the \ at the end of a line indicates that the content continues in the next line.

Other ways of setting variables

```
CPPFLAGS ?= -DNDEBUG
LDLIBS += -llapack
```

In the first line, `CPPFLAGS` will be set to `-DNDEBUG` only if not already set. In the second line `-llapack` will be added to the existing content of `LDLIBS`.

Variables may be specified at the moment of launching the command, and that specification takes the precedence:

```
make all CPPFLAGS=-O3 LDLIBS=-ldl
```

`make` will make target all with `CPPFLAGS=-O3` and `LDLIBS=-ldl -llapack`.

Passing variables as arguments of make

We have already seen that `make` can take a variable as argument, which will override the macro definition in the file. Just remember of using quotes when necessary:

```
make CXXFLAGS="-O3 -Wall"
```

will override any definition of `CXXFLAGS` in the makefile. but you can also do

```
make CXXFLAGS+="-O3 -Wall"
```

and add `-O3 -Wall` to the possible `CXXFLAGS` defined in the makefile. Very useful!

Getting variables from the environment

`make` imports variable from the working environment. If you set (using `bash` shell)

```
>export CXX=clang++
```

and if the `Makefile` does not redefine `CXX`, the value `clang++` will be used.

This is the feature used in the `Makefiles` of the examples for the PACS course to define some variables by using environment variables set by the module system.

If you want to see the environment (exported) variable currently set, do

```
>env
```

Manipulating variables

Make provides a huge set of tools to manipulate or interrogate variables

```
SRCS=main.cpp other.cpp
OBJS = $(SRCS:.cpp=.o)
HEADERS=$(wildcard *.hpp)
```

You can substitute substrings, or use wildcards to select particular files in the working directory.

In the example `OBJS` is obtained by replacing `.cpp` with `.o` to all files in `SRCS`, while `HEADERS` collects all files with extension `hpp` in the current directory.

The `wildcard` directive indicates that `*` has to be considered as wildcard. Not always necessary, but it is safer to use it.

Calling shell commands

A rule may containg long commands. You can split a long command using \. You can use bash shell commands as a rule:

```
all:
   for i in $(wildcard *.c) do \
cc -c $$i; done
```

compiles all the file `*.c` in your directory. Please note the use of the `wildcard` specificator *(not really needed in this case)* and the use of the `$$` to indicate a *shell variable*.

Normally make echoes the commands. The commands in a rule may be made silent (no echoing) by prefixing them with `@`.

```
clean:
   @rm *.o *.a
```

Letting 'make' deduce the rules

A very interesting feature of `make` are implicit in-built rules: `make` already knows how to create certain targets! For instance, `make` has and implicit rules for updating a '.o' file from a correspondingly '.cpp' or '.C' file. Example, if `main.cpp` is present and we have just

```
main: main.cpp
```

then `make main` will launch

```
$(CXX) $(CPPFLAGS) $(CXXFLAGS) $(LDFLAGS) -o main main.cpp $(LDLIBS)
```

automatically!

Here, variables `CXX`, `CPPFLAGS` and `CXXFLAGS` etc. are predefined variables whose default values may be changed by the user.

Using implicit rules

```
CXX=clang++
OPTFLAGS=-g this is not a Makefile var.
CPPFLAGS=-DHAS_FLOAT -I./include
CXXFLAGS=$(OPTFLAGS) -Wall
LDFLAGS=$(OPTFLAGS)
LDLIBS=-L/mylibdir -lmylib
LINK.o = $(CXX) $(LDFLAGS) $(TARGET_ARCH)
main: main.o other.o
other.o: other.cpp ./include/other.hpp
```

`make` will look if in the current directory and if it finds a `main.o` or `other.o` newer than `main`, it will produce `main` by calling the implicit rule. The same apply for `other.o` and `main.o`.

Main variables for implicit rules

`CXX`	the c++ compiler (g++)
`CPPFLAGS`	Options for the C preprocessor
`CXXFLAGS`	Options for the C++ compiler
`CCFLAGS`	Options for the C compiler
`FFLAGS`	Options for the Fortran compiler
`LDFLAGS`	Options for the linker (not for indicating libraries!)
`LDLIBS`	To indicate libraries to be loaded
`LINK.o`	The command used for the linking stage
`TARGET_ARCH`	The target architecture

The macro LINK.o the command for calling the linker on object files `*.o`. By default it is equal to cc, i.e. it uses the C linker) as linker! If you are using C++ it's better to change it so that it loads the standard library, by setting

```
LINK.o = $(CXX) $(LDFLAGS) $(TARGET_ARCH)
```

Other useful implicit variables

`RM`	Command to remove files (rm -f)
`CC`	The C compiler (gcc)
`FC`	The Fortran compiler (gfortran)
`CPP`	The C preprocessor ($(CC) -E)
`AR`	The command to produce static library (ar)
`ARFLAGS`	The flags for AR (rv)

Common Preprocessor options in CPPFLAGS

-I<dirname>	Add <dirname> to the directories to search for included (header) files
-D<Macro>	Define pre-processor variable <Macro>
-D<Macro=value>	Provide value to pre-processor variable <Macro>
-DNDEBUG	Activate the NDEBUG cpp variable, used to indicate that the code should be optimized.

Common c++ compiler options in `CXXFLAGS`

`-g`	Activate debugging (it implies =00)
`-O[0-3]`	Optimization level (0 none, 3 maximal)
`-Og`	Perform only optimizations that allow reasonable debugging
`-Ofast`	Perform also optimizations that don't comply with IEEE standard (implies -O3)
`-fPIC`	Generate position-independent code suitable for use in a shared library
`-fpic`	Another version of `-fPIC`
`-std=[standard]`	Use a specific standard. Possible `[standard]` may be `c++11` or `c++14` or `c++17`
`-Wall`	Activate (almost) all warnings
`-pedantic`	Be pedantic, warn about use of compiler extensions to the standard

Common linker options in LDFLAGS

`-O<lev>`	Optimization level (usually the same used for compilation)
`-shared`	Create a shared library
`-static`	Link only with static libraries (use with care!)
`-dynamic`	Link only with dynamic libraries (use with care!)
`-e`	Create an executable (the default in Linux and Windows systems)
`-Wl,-rpath=<dir>`	Set the loader to look also in `<dir>` for dynamic (shared) libraries
`-o <output>`	The name of the produced file (executable or shared lib)

Common linker options in LDLIBS

`-L<dir>`	Consider also `<dir>` as directory where to search for libraries
`-l<name>`	Link with library `lib<name>.so` or `lib<name>.a`
`<libname>`	Link with library `<libname>` (alternative way to link a library)

Including other make files

The include directive tells 'make' to include one or more other files before continuing. The directive is a line in the makefile that looks like this:

```
include FILENAME
```

If FILENAMES is empty, nothing is included an error is issued and make stops
If you want instead that `make` ignores the error, prefix the command with a `-`:

```
-include FILENAME
```

Using the compiler to find prerequisites

The main compilers (like gnu and LVM compilers) have a set of nice option `-M`, `-MM`, `-MP`, `-MT` ... that exploits the preprocessor to generate a file of prerequisites by examining a source file. An example of usage of `-MM`

```
SRCS=main.cpp readParameters.cpp
make.dep: $(SRCS)
$(RM) make.dep
for f in $(SRCS); do \
  $(CXX) $(CPPFLAGS) -MM $$f >> make.dep;
done
-include make.dep
```

Here `$SRCS` is a variable containing a list of source files. Note the `-` to avoid make to stop with error if `make.dep` is still missing.

The result

The `-MM` option scans the source files given in input and looks for dependencies, in particular included header files, excluding system header files. The other options mentioned before (they all start with `-M`) may operate differently. Here is the result, stored in `make.dep`

```
readParameters.o: readParameters.cpp GetPot.hpp \
readParameters.hpp parameters.hpp
parameters.o: parameters.cpp parameters.hpp
main.o: main.cpp readParameters.hpp parameters.hpp \
GetPot.hpp gnuplot-iostream.hpp
```

All those dependencies has been found automatically starting from `readParameters.cpp`,`parameters.cpp`,`main.cpp`. It's a great simplification. There is also an external utility, called makedepend, which may be used for the same purpose (but I prefer the compiler option).

List of Implicit rules

If you want to see the current rules type

```
make -p -n -f /dev/null >rules.txt
```

The file contains the default rule (you have launched make on the null device)

```
make -p -n -f Makefile >rules.txt
```

will give the rules after processing your `Makefile`.

Phony targets

A target is called phony when it is not associated to any prerequisite but it expresses an action.
It may be useful (but not compulsory) to indicate the targets that are phony, so make avoids searching for a file with the name of the target. You can di the special variable `.PHONY`:

```
.PHONY: all clean distclean
```

Now `all` (often used as first target), `clean` (normally used to clean temporary files but leaving executables untouched), `distclean` (used to clean all temporaries, executables etc..) are phony targets.

Use of phony targets

```
clean:
    $(RM) *.o
distclean:clean
    $(RM) main
```

Using a phony target as prerequisites means running its associated rule.

A more complex example

```
CXXFLAGS = -g
SRCS=main.cpp other.cpp
OBJS = $(SRCS:.cpp=.o)
HEADERS=$(wildcard *.hpp)
EXEC=main
.PHONY=all
all: $(EXEC)
clean:
   $(RM) $(EXEC) $(OBJS) results.dat
$(EXEC): $(OBJS)
$(OBJS): $(SRCS) $(HEADERS)
```

Where to search prerequisites?

Make search the prerequites in your the directory where the makefile resides. If you want to extend the search use the spacial variable `VPATH`

```
VPATH= ./includes /myhome/includes
```

tells make to search prerequisites also in the directories indicated. If you want to be more precise you may use the directive `vpath`:

```
vpath %.hpp ./include
```

tells make to search files ending with `.hpp` in `./include`.

Useful options of `make`

Many make options may be given either in short or long form (use `man make` to see the manual).

- `-j N` Compile in parallel using `N` processes.
- `-d` Give some more detail (a little verbose)
- `-B` Unconditionally make all targets.
- `make MACRO=VALUE` Replace `VALUE` as the value of the variable `MACRO`. It overrides internal definitions.
- `-f filename`. Input is taken from `filename` (instead of `Makefile`)
- `-n` or `-just-print`. Prints the commands that will normally be executed, without executing them.
- `make -p -f/dev/null` Prints the database and does not execute any `Makefile` (`/dev/null` in a Unix system is the null file: an empty file) . Useful to see the in-built macros and rules.

More advanced stuff: Launching make from make

The `MAKE` macro is put automatically equal to the make command. It is used to run another instance of `make` (sub-make) from the makefile

```
...
optimised:
   $(MAKE) CXXFLAGS="-O3 -Wall" all
other:
   $(MAKE) -C subdir
```

Here, `make optimised` launches `make CXXFLAGS="-O3 -Wall"`, while `make other` launches make in the directory `subdir` (equivalent to `cd subdir; $(MAKE)`).
Note: using the `MAKE` macro instead of writing simply `make` is usually better, as the macro exports exported variables.

More advanced stuff: Exported variables

When running a sub-make you may want to export variable defined in the external make to the sub-make. This may be important if the sub-make uses another `Makefile`.
You may use the `export` directive:

```
export  -> all variables will be exported
export variable ->variable will be exported
export variable=value -> you can also give a value
unexport variable -> this variable is not exported
unexport -> all variable are unexported
```

More advanced stuff: automatic variables

Make has a lot of predefined and automatic variables, which may are used in implicit rules and are useful in user defined rules.

```
OBJS=main.o a.o b.o c.o d.o
main : $(OBJS)
$(CXX) -o $@ $^
%.pdf:%.tex
pdflatex $<
%.o:%.cpp
$(CXX) $(CXXFLAGS) $(CPPFLAGS) -c $<
```

But the first and last rule are not necessary. `make` already knows them, they are implicit rules)!.

Explanations

- `CXX` is a predefined variable that contains the name of the c++ compiler, by default is `g++`, but you can change it, for instance with `CXX=clang++`. Also `CXXFLAGS` and `CPPFLAGS` are predefined variables, by default empty.
- The command `%.o:%.C` introduces a user-defined rule to convert files named `something.C` into `something.o`.
- `$<` is an automatic variable that indicates the prerequisite of a target. `$@` indicates instead the name of the target.

There are more automatic variables, you find the list in the gnu make manual.

The main automatic variables

These variable may be used when writing a rule

```
$@ File name of the target of a rule
$< The first prerequisite
$? The name of all prerequisites newer than the target
$^ The names of all prerequisites, with spaces among them
$* The stem with which an implicit rule matches.
```

if the target pattern is `%.o` and the target is `src/pippo.c` then the stem is `src/pippo`

But we will see soon that things may be made simpler with implicit rules!

More advanced stuff: Pattern substitution

Sometimes some names are repeated with just the suffix changed. You may use the so called static pattern rule technique to avoid repetitions:

```
objects = foo.o bar.o
all: $(objects)
$(objects): %.o: %.c
   $(CC) -c $(CFLAGS) $< -o $@
```

The string `%.o: %.c` means *replace the suffix `.o` with `.c`*.
I recall that `$<` and `$@` are the automatic variables that hold the name of the prerequisite and of the target, respectively.

More advanced stuff: Conditionals

It is possible to have conditional constructs

```
main: $(OBJECTS)
ifeq ($(CC),gcc )
   $(CC) -o main $(OBJECTS) $(LIBS_FOR_GCC)
else
   $(CC) -o main $(OBJECTS) $(NORMAL_LIBS)
endif
```

Adv. Stuff: Calling bash variables inside Makefiles

Another example of calling shell commands in a Makefile

```
dist: $(SRCS)
   for X in $(SRCS) ; do \
    sed 's/AUTHOR/Luca/g' $$X > tmp.dir/$$X ;\
done
```

A shell variable `X` is recalled by using `$$X`.

In this example `make dist` will loop on all files whose name is in `SRCS` and replace any occurrence of `AUTHOR` with `Luca`, writing the result in a file with the same name but in another directory.

The unix shell (and bash in particular) has dozens of very powerful commands that may make life easier... but this would be another lecture...

What more

A lot. Current version of make support multiple target per rule, a full set of functions and the capability of working with files residing in different directories.

Much more that can be said in a short course. Yet, you do not need to know all that if you want to start using make! Already with the basic stuff you can simplify your (programming) life.

And if you want to know more

The make utility is rather old. It has been born with the UNIX operative system in the 80's. It has evolved a lot since.
The most used version (the one I have followed in this lecture) is *GNU make*, developed by the Free Software Foundation. More info on
http://www.gnu.org/software/make

There is also a book:

GNU Make: A Program for Directing Recompilation by Richard M. Stallman, Roland McGrath and Paul D. Smith, Free Software Foundation.

which is a pretty-printed version of the manual available on line at the site indicated above.