A (not so) short introduction to "make"

Luca Formaggia, Pasquale Claudio Africa

MOX
Dipartimento di Matematica Politecnico di Milano

Outline

Controlling the compilation process

The compilation process requires to assemble data from different interrelated sources

- A compilation 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 and header files.

The use of Makefiles may help to automatize the compilation by defining *prerequisite-to-target* rules.

What is in fact make making?

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

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

The rules are written on a file, usually called Makefile or makefile (but this is not compulsory, you can specify another file using the `-f` option.).

A simple example

This simple Makefile 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.tex

Another simple example

```
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`.

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 in a series of blank characters, yet it MUST be there, as it identifies the lines containing a command.

`prerequisites` is a list (zero or more) names separated by a space.

Some nomenclature

- A target is either the name of a file that is generated by a program (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. Prerequisites in a list 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` since `target1` is the first target in our example) the command operates recursively using the following algorithm

- Re-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 with the rule.

Not finding any rule for the original 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.

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)
SRCS+=another.cpp
```

You can substitute substrings, use wildcards to select particular files in the working directory, add content to a variable.

In the example `OBJS` is obtained by replacing `.cpp` with `.o` to all files in `SRCS`.
The `wildcard` in the third statement indicates that `*` has to be considered as wildcard.

Special variables

Make has a lot of predefined variables, called implicit 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: %.C
    $(CXX) $(CXXFLAGS) $(CPPFLAGS) -c $<
```

But the first and last rule are not necessary. `make` already knows them (they are *automatic rules*)!

Explanations

- `CXX` is a predefined variable that contains the name of the C++ compiler, by default g++, but you can change it, for instance `CXX=clang++`.
- 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 the name of the target.

Letting ‘make’ deduce the rules

It is not necessary to spell out the commands for compiling the individual C++ source files (and many other languages as well), because ‘make’ has implicit rules for updating a ‘.o’ file from a correspondingly ‘.cpp’ or ’.C’ file that is equal to `$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c` command automatically.

In this context the variables CXX, CPPFLAGS and CXXFLAGS are implicit variables whose default values are set by make (typically `CXX=g++` while `CPPFLAGS` and `CXXFLAGS` are empty, but 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=$(CC) $(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 contains options for calling the linker on object files `*.o`. By default it uses `cc` (the C compiler) as linker! If you are using C++ it's better to change it so that it loads the standard library!

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)

Note: `make` inherits variables from your environment. So if `CXX` is set as environmental variable it will be used as indicating the C++ compiler by `make`, unless it is overwritten in the Makefile.

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 g++ 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)

How do implicit rules work?

The special target `.SUFFIXES` contains the order of dependence of files in an implicit rule. You can change it.

```
.SUFFIXES: .out .a .o .c .cc .C .cpp .p .f .F
```

`a.out` indicates an executable (which in Unix has no extension). The other values are the extensions that are searched by `make`, in the order of searching: `.o` may depend on a C file (`.c`) or a C++ file (`.cpp` or `.C`), etc.

The full list is much longer! Normally you don't have to change .SUFFIX!

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 whene it is not associated to any file.
Usually a phony target indicates an action to be taken, that do not depend on prerequisites.
It may be useful (but not compulsory) to indicate the phony targets so that make avoids searching for a file of the name of the target using 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.

Targets without prerequisites

A target may be without prerequisites (phony target). For instance the following rules to clean up the directory

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

We could also have written (better!)

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

Invoking a target with no prerequisites means running the associated rule. Target without prerequisites are necessarily *phony*, better declare them as such!

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)
```

Passing macros as arguments of make

Running `make` with a macro as argument will override the macro definition in the file. For instance,

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

will override any definition of `CXXFLAGS` in the Makefile.

Very useful!

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, since the macro replicates possible command parameters you used in the first place.

Exported variables

When running a sub-make you may 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
```

Including other Makefiles

The include directive tells 'make' to suspend reading the current Makefile and read one or more other Makefiles 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 make to ignore the error, prefix the command with a -:

```
-include FILENAME
```

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*. `$<` and `$@` are the automatic variables that hold the name of the prerequisite and of the target, respectively.

Some rules for the rules

A rule may containg long commands. You can split a long command using the \. E.g. the 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
```

Some rules for the rules

If an error occurs while executing a command, make stops. If you want to avoid it prefix the rule with `-`. The rule

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

will not stop `make` if no files `*.o` and `*.a` are present (in this case, however, better use `$(RM)`).

Alternatively, if you do not want make to stop on errors, call it with the `-k` (or `--keep-going`) option:

```
make -k
```

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`.

Defining (or redefining) an implicit rule

Suppose that you have a command `jpeg2gif` that transform a jpeg image to a gif and you want to create a Makefile so that typing `make pict.gif` will automatically produce the gif file from the corresponding jpeg.

You may want to do this by writing your own rule. You have (a) to tell make that files ending with `.gif` depend from the corresponding `.jpg`, and (b) specify the command to use for the conversion.

From jpg to gif

The Makefile may contain

```
.SUFFIXES: .gif .jpg
%.gif:%.jpg
    jpeg2gif $^
```

`.SUFFIXES` is a macro used as dummy target to indicate the mutual dependency of suffixes. The default list contains the list of known dependencies.

Here, `$^` is a automatic variable which converts to the names of all prerequisites, with spaces between them.

Automatic variables

These variable are 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`

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
```

Calling bash variables inside Makefiles

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

I can use shell commands inside a Makefile. A shell variable `X` is recalled by using `$$X`.

Using the compiler to generate rules

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 rules starting from a source file. An example of usage of `-MM`

```
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 of an example

```
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 of life. There is also an external utility, called makedepend that may be used for the same purpose (but I prefer the compiler option).

Other 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, *i.e.* an empty file). Useful to see the built-in macros.

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.

For the real gurus

The make utility is the basic utility for software developers. Other utilities may help you to develop portable programs and to assit you to compilations on different architectures, and handle automatic search for libraries etc.

In particular the autotools utilities.
You find a manual in http://sourceware.org/autobook/.

An alternative to autotools is CMake.

But for small projects writing your Makefile directly is often simpler.

Autotools

Autotools are a rather complex set of programs developed by the Free Software Foundation for the Unix system. They allow to finely control the compilation of complex programs. They normally produce a *configure* script which, when launched, verifies the availability of the required libraries, the compiler version and the computer architecture and eventually generate the Makefiles. They control also the generation of libraries allowing to separate the *development phase* (libraries are local) to the *deployment phase* (libraries and header files are installed).

Compiling a package with the configure script

- Download the sources in a directory, for instance `/home/me/packagedir`.
- Create a directory for the build `/home/me/packagebuild` (this step may be not required)
- `cd /home/me/packagebuild`, `/home/me/packagedir/configure [options]` will produce the Makefiles
- `make all` (to compile), `make install` to install (normally in `/usr/local/`);
- `make clean` to clean everything.

Main options of configure

Normally software compiled from scratch and not part of a unix distribution should be installed in `/usr/local/include` (header files), in `/usr/local/lib` (libraries) and `/usr/local/bin` (executables). Normally open source software compiled using autotools follows this rule.
However you may change the destination root directory using the option `--prefix=`*path*, where *path* is a directory path. So `configure --prefix=/opt/prog` will prepare a Makefile which installs header file in `/opt/prog/include`, libraries in `/opt/prog/lib` etc.
It is also possible to indicate the target directories separately.

CMake

`CMake` (Cross-Platform Makefile Generator) is replacing the autotools and becoming the *de facto* standard in the compilation in many software packages. The advantage of CMake is that it is multi-architecture (it works natively also on Windows) and has a graphic interface to ease configuration and is integrated into many IDE (among which Eclipse and CLion).
The main file containing CMake information is normally called `CMakeLists.txt`.

Compiling a package with CMake

- Download the sources in a directory, for instance `/home/me/packagedir`.
- Create a directory for the build `/home/me/packagebuild` (this step may be not required)
- `cd /home/me/packagebuild`, `cmake /home/me/packagedir` will produce the Makefiles
- `make all` (to compile), `make install` to install (normally in `/usr/local/`);
- `make clean` to clean everything.