A short introduction to "make"

Luca Formaggia

MOX
Dipartimento di Matematica Politecnico di Milano

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 production of an 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++ -O3 -o main main.o funct.o
funct.o: funct.cpp funct.hpp other.hpp
<TAB> g++ -c -O3 funct.cpp
main.o: main.cpp
<TAB> g++ -c -O3 main.cpp
```

The command `make main` will produce the main file.

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 of 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 launch 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

How it works

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

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

Simplifying your life: 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)
```

Manipulating variables

Gnu 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....

Special variables

Make has a lot of predefined variables, some may are used when defining rules

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

But the fist and last rule are not necessary. `make` already knows them!.

Letting ‘make’ deduce the commands

It is not necessary to spell out the commands for compiling the individual C source files, because ‘make’ has implicit rules for updating a ‘.o’ file from a correspondingly named ‘.c’ file using the `$(CC) $(CCFLAGS) -c’` command automatically.

In this context the variables CC and CFLAGS are special variables whose default values are set by make (typically `CC=cc` while `CFLAGS` is empty), but may be changed by the user.
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`.

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
main: main.o other.o
```

The macro LDFLAGS contains flags passed to the linker to produce the executable file `main` from the object files `main.o` and `other.o`

Main variables for implicit rules

CXX	the c++ compiler (g++)
CPPFLAGS	Flags for the C preprocessor
CXXFLAGS	Flags for the c++ compiler
CCFLAGS	Flags for the C compiler
FFLAGS	flags for the Fortran compiler
LDFLAGS	Flags for the linker (not for indicating libraries!
LDLIBS	To indicate libraries to be loaded

Other variables for implicit rules

RM	Command to remove files (rm -f)
CC	The C compiler (gcc)
FC	The Fortran compiler (gfortran)
CPP	The C preprocessor ($(GCC) -E)

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
```

Phony targets

A target is called phony whene it is not associated to any file.
Usually a phony target indicates an action.
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
```

Targets without prerequisites

A target may be without prerequisites. For instance the following rule clean up the directory

```
clean:
    rm *.o main
```

Invoking target with no prerequisites implies running the associated rule . Target without prerequisite are necessarily *phony targets*.

Handling errors

You'd better write the previous rule as

```
.PHONY : clean
clean:
    -rm *.o main
```

The predefined macro `.PHONY` is used as a target to indicate that `clean` is a in fact a phony target. It is not necessary to do so, but it may avoid problems if you have a file called *clean* in your directory!

The – in front of the `rm` command tells make to ignore command errors (e.g. if a file does not exist). As default, `make` stops on errors.

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 -f $(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.

Launching make from make

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

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

Here, `make optimised` will launch `make CXXFLAGS="-O3 -Wall"`
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.

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 FILENAMES...
```

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`

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 the make if no files `*.o` and `*.a` are present.
If you want to be sure that make executes the command ignoring aliases introduced in your environment, prefix it with the backslash

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

But in this case it is much better to write

```
clean:
$(RM) *.o *.a
```

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`.

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 (at least in the Unix community, but not only) is today the *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 prettyprint version of the manual available on line at the site indicated before.

Some useful options of `make`

Many make options may be given either in short or long form.

- `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` of `-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 inbuilt macros.

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 :autoconf , automake and libtool utilities.
You find a manual on this tools in
http://sourceware.org/autobook/

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 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).
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.