Chapter 30

Managing projects with Make

The *Make* utility helps you manage the building of projects: its main task is to facilitate rebuilding only those parts of a multi-file project that need to be recompiled or rebuilt. This can save lots of time, since it can replace a minutes-long full installation by a single file compilation. *Make* can also help maintaining multiple installations of a program on a single machine, for instance compiling a library with more than one compiler, or compiling a program in debug and optimized mode.

Make is a Unix utility with a long history, and traditionally there are variants with slightly different behaviour, for instance on the various flavours of Unix such as HP-UX, AUX, IRIX. These days, it is advisable, no matter the platform, to use the GNU version of Make which has some very powerful extensions; it is available on all Unix platforms (on Linux it is the only available variant), and it is a *de facto* standard. The manual is available at <http://www.gnu.org/software/make/manual/make.html>, or you can read the book [**?**].

There are other build systems, most notably *Scons* and *Bjam*. We will not discuss those here. The examples in this tutorial will be for the C and Fortran languages, but *Make* can work with any language, and in fact with things like TEX that are not really a language at all; see section **??**.

30.1 A simple example

Purpose. In this section you will see a simple example, just to give the flavour of *Make*.

The files for this section can be found in the repository in the directory `tutorials/make_tutorial_files`.

30.1.1 C

Make the following files:

foo.c

```
#include "bar.h"
int c=3;
int d=4;
int main()
```

```
{
  int a=2;
  return(bar(a*c*d));
}
```

bar.c

```
#include "bar.h"
int bar(int a)
{
  int b=10;
  return(b*a);
}
```

bar.h

```
extern int bar(int);
```

and a makefile:

Makefile

```
fooprog : foo.o bar.o
        cc -o fooprog foo.o bar.o
foo.o : foo.c
        cc -c foo.c
bar.o : bar.c
        cc -c bar.c
clean :
        rm -f *.o fooprog
```

The makefile has a number of rules like

```
foo.o : foo.c
<TAB>cc -c foo.c
```

which have the general form

```
target : prerequisite(s)
<TAB>rule(s)
```

where the rule lines are indented by a `TAB` character.

A rule, such as above, states that a 'target' file `foo.o` is made from a 'prerequisite' `foo.c`, namely by executing the command `cc -c foo.c`. The precise definition of the rule is:

- if the target `foo.o` does not exist or is older than the prerequisite `foo.c`,
- then the command part of the rule is executed: `cc -c foo.c`
- If the prerequisite is itself the target of another rule, than that rule is executed first.

Probably the best way to interpret a rule is:

- if any prerequisite has changed,
- then the target needs to be remade,
- and that is done by executing the commands of the rule;
- checking the prerequisite requires a recursive application of make:
  - if the prerequisite does not exist, find a rule to create it;
  - if the prerequisity already exists, check applicable rules to see if it needs to be remade.

If you call `make` without any arguments, the first rule in the makefile is evaluated. You can execute other rules by explicitly invoking them, for instance `make foo.o` to compile a single file.

Exercise. Call `make`.

> *Expected outcome.* The above rules are applied: `make` without arguments tries to build the first target, `fooprog`. In order to build this, it needs the prerequisites `foo.o` and `bar.o`, which do not exist. However, there are rules for making them, which `make` recursively invokes. Hence you see two compilations, for `foo.o` and `bar.o`, and a link command for `fooprog`.
> *Caveats.* Typos in the makefile or in file names can cause various errors. In particular, make sure you use tabs and not spaces for the rule lines. Unfortunately, debugging a makefile is not simple. *Make*'s error message will usually give you the line number in the make file where the error was detected.

Exercise. Do `make clean`, followed by `mv foo.c boo.c` and `make` again. Explain the error message. Restore the original file name.

> *Expected outcome. Make* will complain that there is no rule to make `foo.c`. This error was caused when `foo.c` was a prerequisite for making `foo.o`, and was found not to exist. *Make* then went looking for a rule to make it and no rule for creating `.c` files exists.

Now add a second argument to the function `bar`. This requires you to edit `bar.c` and `bar.h`: go ahead and make these edits. However, it also requires you to edit `foo.c`, but let us for now 'forget' to do that. We will see how *Make* can help you find the resulting error.

Exercise. Call `make` to recompile your program. Did it recompile `foo.c`?

> *Expected outcome.* Even through conceptually `foo.c` would need to be recompiled since it uses the `bar` function, *Make* did not do so because the makefile had no rule that forced it.

In the makefile, change the line

```
foo.o : foo.c
```

to

```
foo.o : foo.c bar.h
```

which adds `bar.h` as a prerequisite for `foo.o`. This means that, in this case where `foo.o` already exists, *Make* will check that `foo.o` is not older than any of its prerequisites. Since `bar.h` has been edited, it is younger than `foo.o`, so `foo.o` needs to be reconstructed.

Exercise. Confirm that the new makefile indeed causes `foo.o` to be recompiled if `bar.h` is changed. This compilation will now give an error, since you 'forgot' to edit the use of the `bar` function.

30.1.2 Fortran

Make the following files:

foomain.F

```
        program test
        use testmod

        call func(1,2)

        end program
```

foomod.F

```
        module testmod

        contains

        subroutine func(a,b)
        integer a,b
        print *,a,b,c
        end subroutine func

        end module
```

and a makefile:

Makefile

```
    fooprog : foomain.o foomod.o
        gfortran -o fooprog foo.o foomod.o
    foomain.o : foomain.F
        gfortran -c foomain.F
    foomod.o : foomod.F
        gfortran -c foomod.F
    clean :
        rm -f *.o fooprog
```

If you call `make`, the first rule in the makefile is executed. Do this, and explain what happens.

Exercise. Call `make`.

> *Expected outcome.* The above rules are applied: `make` without arguments tries to build the first target, `foomain`. In order to build this, it needs the prerequisites `foomain.o` and `foomod.o`, which do not exist. However, there are rules for making them, which `make` recursively invokes. Hence you see two compilations, for `foomain.o` and `foomod.o`, and a link command for `fooprog`.
>
> *Caveats.* Typos in the makefile or in file names can cause various errors. Unfortunately, debugging a makefile is not simple. You will just have to understand the errors, and make the corrections.

Introduction to High Performance Scientific Computing

Exercise. Do `make clean`, followed by `mv foomod.c boomod.c` and `make` again. Explain the error message. Restore the original file name.

> *Expected outcome. Make* will complain that there is no rule to make `foomod.c`. This error was caused when `foomod.c` was a prerequisite for `foomod.o`, and was found not to exist. *Make* then went looking for a rule to make it, and no rule for making `.F` files exists.

Now add an extra parameter to `func` in `foomod.F` and recompile.

Exercise. Call `make` to recompile your program. Did it recompile `foomain.F`?

> *Expected outcome.* Even through conceptually `foomain.F` would need to be recompiled, *Make* did not do so because the makefile had no rule that forced it.

Change the line

```
foomain.o : foomain.F
```

to

```
foomain.o : foomain.F foomod.o
```

which adds `foomod.o` as a prerequisite for `foomain.o`. This means that, in this case where `foomain.o` already exists, *Make* will check that `foomain.o` is not older than any of its prerequisites. Recursively, *Make* will then check if `foomode.o` needs to be updated, which is indeed the case. After recompiling `foomode.F`, `foomode.o` is younger than `foomain.o`, so `foomain.o` will be reconstructed.

Exercise. Confirm that the corrected makefile indeed causes `foomain.F` to be recompiled.

30.1.3 About the make file

The make file needs to be called `makefile` or `Makefile`; it is not a good idea to have files with both names in the same directory. If you want *Make* to use a different file as make file, use the syntax `make -f My_Makefile`.

30.2 Variables and template rules

> **Purpose.** In this section you will learn various work-saving mechanisms in *Make*, such as the use of variables and of template rules.

30.2.1 Makefile variables

It is convenient to introduce variables in your makefile. For instance, instead of spelling out the compiler explicitly every time, introduce a variable in the makefile:

```
CC = gcc
FC = gfortran
```

and use `${CC}` or `${FC}` on the compile lines:

```
foo.o : foo.c
        ${CC} -c foo.c
foomain.o : foomain.F
        ${FC} -c foomain.F
```

Exercise. Edit your makefile as indicated. First do `make clean`, then `make foo` (C) or `make fooprog` (Fortran).

> *Expected outcome.* You should see the exact same compile and link lines as before.
>
> *Caveats.* Unlike in the shell, where braces are optional, variable names in a makefile have to be in braces or parentheses. Experiment with what hapens if you forget the braces around a variable name.

One advantage of using variables is that you can now change the compiler from the commandline:

```
make CC="icc -O2"
make FC="gfortran -g"
```

Exercise. Invoke *Make* as suggested (after `make clean`). Do you see the difference in your screen output?

> *Expected outcome.* The compile lines now show the added compiler option `-O2` or `-g`.

Make also has *automatic variables*:

- `$@` The target. Use this in the link line for the main program.
- `$^` The list of prerequisites. Use this also in the link line for the program.
- `$<` The first prerequisite. Use this in the compile commands for the individual object files.
- `$*` In *template rules* (section **??**) this matches the template part, the part corresponding to the `%`.

Using these variables, the rule for `fooprog` becomes

```
fooprog : foo.o bar.o
        ${CC} -o $@ $^
```

and a typical compile line becomes

```
foo.o : foo.c bar.h
        ${CC} -c $<
```

You can also declare a variable

```
THEPROGRAM = fooprog
```

and use this variable instead of the program name in your makefile. This makes it easier to change your mind about the name of the executable later.

Exercise. Edit your makefile to add this variable definition, and use it instead of the literal program name. Construct a commandline so that your makefile will build the executable `fooprog_v2`.

> *Expected outcome.* You need to specify the `THEPROGRAM` variable on the commandline using the syntax `make VAR=value`.
>
> *Caveats.* Make sure that there are no spaces around the equals sign in your commandline.

The full list of these automatic variables can be found at `https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html`.

30.2.2 Template rules

So far, you wrote a separate rule for each file that needed to be compiled. However, the rules for the various `.c` files are very similar:

- the rule header (`foo.o : foo.c`) states that a source file is a prerequisite for the object file with the same base name;
- and the instructions for compiling (`${CC} -c $<`) are even character-for-character the same, now that you are using *Make*'s built-in variables;
- the only rule with a difference is

```
foo.o : foo.c bar.h
        ${CC} -c $<
```

where the object file depends on the source file and another file.

We can take the commonalities and summarize them in one *template rule*[1]:

```
%.o : %.c
        ${CC} -c $<
%.o : %.F
        ${FC} -c $<
```

This states that any object file depends on the C or Fortran file with the same base name. To regenerate the object file, invoke the C or Fortran compiler with the `-c` flag. These template rules can function as a replacement for the multiple specific targets in the makefiles above, except for the rule for `foo.o`.

The dependence of `foo.o` on `bar.h`, or `foomain.o` on `foomod.o`, can be handled by adding a rule

---

1. This mechanism is the first instance you'll see that only exists in GNU make, though in this particular case there is a similar mechanism in standard make. That will not be the case for the wildcard mechanism in the next section.

```
# C
foo.o : bar.h
# Fortran
foomain.o : foomod.o
```

with no further instructions. This rule states, 'if file `bar.h` or `foomod.o` changed, file `foo.o` or `foomain.o` needs updating' too. *Make* will then search the makefile for a different rule that states how this updating is done, and it will find the template rule.

Exercise. Change your makefile to incorporate these ideas, and test.

30.2.3   Wildcards

Your makefile now uses one general rule for compiling any source file. Often, your source files will be all the `.c` or `.F` files in your directory, so is there a way to state 'compile everything in this directory'? Indeed there is.

Add the following lines to your makefile, and use the variable `COBJECTS` or `FOBJECTS` wherever appropriate. The command `wildcard` gives the result of `ls`, and you can manipulate file names with `patsubst`.

```
# wildcard: find all files that match a pattern
CSOURCES := ${wildcard *.c}
# pattern substitution: replace one pattern string by another
COBJECTS := ${patsubst %.c,%.o,${SRC}}

FSOURCES := ${wildcard *.F}
FOBJECTS := ${patsubst %.F,%.o,${SRC}}
```

30.2.4   Conditionals

There are various ways of making the behaviour of a makefile dynamic. You can for instance put a shell conditional in an action line. However, this can make for a cluttered makefile; an easier way is to use makefile conditionals. There are two types of conditionals: tests on string equality, and tests on environment variables.

The first type looks like

```
ifeq "${HOME}" "/home/thisisme"
  # case where the executing user is me
else
  # case where it's someone else
endif
```

and in the second case the test looks like

```
ifdef SOME_VARIABLE
```

The text in the true and false part can be most any part of a makefile. For instance, it is possible to let one of the action lines in a rule be conditionally included. However, most of the times you will use conditionals to make the definition of variables dependent on some condition.

Exercise. Let's say you want to use your makefile at home and at work. At work, your employer has a paid license to the Intel compiler `icc`, but at home you use the open source Gnu compiler `gcc`. Write a makefile that will work in both places, setting the appropriate value for `CC`.

30.3   Miscellania

30.3.1   What does this makefile do?

Above you learned that issuing the `make` command will automatically execute the first rule in the makefile. This is convenient in one sense[2], and inconvenient in another: the only way to find out what possible actions a makefile allows is to read the makefile itself, or the – usually insufficient – documentation.

A better idea is to start the makefile with a target

```
info :
        @echo "The following are possible:"
        @echo "  make"
        @echo "  make clean"
```

Now `make` without explicit targets informs you of the capabilities of the makefile.

If your makefile gets longer, you might want to document each section like this. This runs into a problem: you can not have two rules with the same target, `info` in this case. However, if you use a double colon it *is* possible. Your makefile would have the following structure:

```
info ::
        @echo "The following target are available:"
        @echo "  make install"
install :
        # ..... instructions for installing
info ::
        @echo "  make clean"
clean :
        # ..... instructions for cleaning
```

---

2.   There is a convention among software developers that a package can be installed by the sequence `./configure ; make ; make install`, meaning: Configure the build process for this computer, Do the actual build, Copy files to some system directory such as `/usr/bin`.

30.3.2 Phony targets

The example makefile contained a target `clean`. This uses the *Make* mechanisms to accomplish some actions that are not related to file creation: calling `make clean` causes *Make* to reason 'there is no file called `clean`, so the following instructions need to be performed'. However, this does not actually cause a file `clean` to spring into being, so calling `make clean` again will make the same instructions being executed.

To indicate that this rule does not actually make the target, you use the `.PHONY` keyword:

```
.PHONY : clean
```

Most of the time, the makefile will actually work fine without this declaration, but the main benefit of declaring a target to be phony is that the *Make* rule will still work, even if you have a file (or folder) named `clean`.

30.3.3 Predefined variables and rules

Calling `make -p yourtarget` causes make to print out all its actions, as well as the values of all variables and rules, both in your makefile and ones that are predefined. If you do this in a directory where there is no makefile, you'll see that make actually already knows how to compile `.c` or `.F` files. Find this rule and find the definition of the variables in it.

You see that you can customize make by setting such variables as `CFLAGS` or `FFLAGS`. Confirm this with some experimentation. If you want to make a second makefile for the same sources, you can call `make -f othermakefile` to use this instead of the default `Makefile`.

Note, by the way, that both `makefile` and `Makefile` are legitimate names for the default makefile. It is not a good idea to have both `makefile` and `Makefile` in your directory.

30.3.4 Using the target as prerequisite

Suppose you have two different targets that are treated largely the same. You would want to write:

```
PROGS = myfoo other
${PROGS} : $@.o # this is wrong!!
        ${CC} -o $@ $@.o ${list of libraries goes here}
```

and saying `make myfoo` would cause

```
cc -c myfoo.c
cc -o myfoo myfoo.o ${list of libraries}
```

and likewise for `make other`. What goes wrong here is the use of `$@.o` as prerequisite. In Gnu Make, you can repair this as follows[3]:

---

3.   Technical explanation: Make will now look at lines twice: the first time `$$` gets converted to a single `$`, and in the second pass `$@` becomes the name of the target.

```
.SECONDEXPANSION:
${PROGS} : $$@.o
        ${CC} -o $@ $@.o ${list of libraries goes here}
```

Exercise. Write a second main program `foosecond.c` or `foosecond.F`, and change your makefile so that the calls `make foo` and `make foosecond` both use the same rule.

30.4 Shell scripting in a Makefile

Purpose. In this section you will see an example of a longer shell script appearing in a makefile rule.

In the makefiles you have seen so far, the command part was a single line. You can actually have as many lines there as you want. For example, let us make a rule for making backups of the program you are building.

Add a `backup` rule to your makefile. The first thing it needs to do is make a backup directory:

```
.PHONY : backup
backup :
        if [ ! -d backup ] ; then
          mkdir backup
        fi
```

Did you type this? Unfortunately it does not work: every line in the command part of a makefile rule gets executed as a single program. Therefore, you need to write the whole command on one line:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
```

or if the line gets too long:

```
backup :
        if [ ! -d backup ] ; then \
          mkdir backup ; \
        fi
```

Next we do the actual copy:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
```

But this backup scheme only saves one version. Let us make a version that has the date in the name of the saved program.

The Unix `date` command can customize its output by accepting a format string. Type the following: `date` This can be used in the makefile.

Exercise. Edit the `cp` command line so that the name of the backup file includes the current date.

> *Expected outcome.* Hint: you need the backquote. Consult the Unix tutorial, section 27.3.3, if you do not remember what backquotes do.

If you are defining shell variables in the command section of a makefile rule, you need to be aware of the following. Extend your `backup` rule with a loop to copy the object files:

```
#### This Script Has An ERROR!
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
        for f in ${OBJS} ; do \
          cp $f backup ; \
        done
```

(This is not the best way to copy, but we use it for the purpose of demonstration.) This leads to an error message, caused by the fact that *Make* interprets `$f` as an environment variable of the outer process. What works is:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
        for f in ${OBJS} ; do \
          cp $$f backup ; \
        done
```

(In this case *Make* replaces the double dollar by a single one when it scans the commandline. During the execution of the commandline, `$f` then expands to the proper filename.)

30.5   Practical tips for using Make

Here are a couple of practical tips.

- *Debugging* a makefile is often frustratingly hard. Just about the only tool is the `-p` option, which prints out all the rules that Make is using, based on the current makefile.
- You will often find yourself first typing a make command, and then invoking the program. Most Unix shells allow you to use commands from the *shell command history* by using the up arrow key. Still, this may get tiresome, so you may be tempted to write

Introduction to High Performance Scientific Computing

```
make myprogram ; ./myprogram -options
```

and keep repeating this. There is a danger in this: if the make fails, for instance because of compilation problems, your program will still be executed. Instead, write

```
make myprogram && ./myprogram -options
```

which executes the program conditional upon make concluding successfully.

30.6 A Makefile for LATEX

The *Make* utility is typically used for compiling programs, but other uses are possible too. In this section, we will discuss a makefile for LATEX documents.

We start with a very basic makefile:

```
info :
        @echo "Usage: make foo"
        @echo "where foo.tex is a LaTeX input file"


%.pdf : %.tex
        pdflatex $<
```

The command `make myfile.pdf` will invoke `pdflatex myfile.tex`, if needed, once. Next we repeat invoking `pdflatex` until the log file no longer reports that further runs are needed:

```
%.pdf : %.tex
        pdflatex $<
        while [ `cat ${basename $@}.log | grep "Rerun to get" \
            | wc -l` -gt 0 ] ; do \
          pdflatex $< ; \
        done
```

We use the `${basename fn}` macro to extract the base name without extension from the target name.

In case the document has a bibliography or index, we run `bibtex` and `makeindex`.

```
%.pdf : %.tex
        pdflatex ${basename $@}
        -bibtex ${basename $@}
        -makeindex ${basename $@}
        while [ `cat ${basename $@}.log | grep "Rerun to get" \
                | wc -l` -gt 0 ] ; do \
          pdflatex ${basename $@} ; \
        done
```

The minus sign at the start of the line means that *Make* should not abort if these commands fail.

Finally, we would like to use *Make*'s facility for taking dependencies into account. We could write a makefile that has the usual rules

```
mainfile.pdf : mainfile.tex includefile.tex
```

but we can also discover the include files explicitly. The following makefile is invoked with

```
make pdf TEXTFILE=mainfile
```

The `pdf` rule then uses some shell scripting to discover the include files (but not recursively), and it calls *Make* again, invoking another rule, and passing the dependencies explicitly.

```
pdf :
        export includes=`grep "^.input " ${TEXFILE}.tex \
            | awk '{v=v FS $$2".tex"} END {print v}'` ; \
        ${MAKE} ${TEXFILE}.pdf INCLUDES="$$includes"

%.pdf : %.tex ${INCLUDES}
        pdflatex $< ; \
        while [ `cat ${basename $@}.log \
                | grep "Rerun to get" | wc -l` -gt 0 ] ; do \
          pdflatex $< ; \
        done
```

This shell scripting can also be done outside the makefile, generating the makefile dynamically.

30.7   Cmake

Makefiles can be cumbersome to make, and they may need customization for any specific installation. For this reason, *Cmake* is a build system that first generates the makefiles, which can then be used in the normal manner.

Here are the main components:

- The source directory has a file `CMakeLists.txt` that describes the structure of the application.
- Generated files can be kept separate from the source:
  ```
  sourcedir=.... # source location
  builddir=....  # place for temporaries
  installdir=... # here goes the finished stuff
  cd ${builddir}
  cmake \
  ```

```
          -DCMAKE_PREFIX_PATH=${installdir} \
          ${sourcedir} # do the setup
    make            # compile
    make install  # move finished stuff in place
```

30.7.1   Dependency analysis

Above you learned how Make can be used for projects with complicated dependencies. You can tell Cmake about this using clauses

```
    include_directories(include)
    file(GLOB SOURCES "*/*.c")
    file(GLOB HEADERS "*/*.h")
    add_executable(hello_world hello.c ${SOURCES} )
```

When invoked, Cmake will then report

```
    Scanning dependencies of target hello_world
```

and generate makefiles accordingly.

One advantage of this setup is that the cmake stage only needs to be redone if the structure of the application changes, for instance because files are added. For any ordinary edits during program development it is enough to repeat the make and make install part.

Example. Consider a main file hello.c that uses two auxiliary files compute.c and output.c. After cmake, the make; make install gives as output:

```
    Scanning dependencies of target hello_world
    [ 25%] Building C object CMakeFiles/hello_world.dir/hello.c.o
    [ 50%] Building C object
                  CMakeFiles/hello_world.dir/compute/compute.c.o
    [ 75%] Building C object CMakeFiles/hello_world.dir/output/output.c.o
    [100%] Linking C executable hello_world
    [100%] Built target hello_world
    [100%] Built target hello_world
    Install the project...
```

After editing the compute.c file only that is recompiled, and the project is relinked:

```
    Scanning dependencies of target hello_world
    [ 25%] Building C object

    CMakeFiles/hello_world.dir/compute/compute.c.o
    [100%] Built target hello_world
    [100%] Built target hello_world
```

```
Install the project...
```

Clearly, cmake has generated makefiles with the right structure. The disadvantage of using cmake is that the automatically generated makefiles are close to unreadable.

30.7.2   Re-Cmaking

The structure sketched just now, with a separate build and install directory, means that completely deleting these libraries is enough to start with a clean slate. For a more modest approach, you can delete the file CMakeCache.txt, which is in the build directory.

30.7.3   System dependencies

You can give Cmake options that find their way into the makefile structure. For instance, on the Apple laptop where this book was written, a straight invocation make Cmake discover the native C compiler:

```
-- The C compiler identification is AppleClang 9.1.0.9020039
```

However, specifying

```
cmake -DCMAKE_C_COMPILER=gcc
```

gives:

```
-- The C compiler identification is GNU 7.4.0
```

30.7.4   Details

CMake has variables:

```
set(myvar myvalue)
message(STATUS "my variable has value ${myvar}")
```