Using `mk`
Linking against shared libraries

Alberto Artoni

01/03/2024

Toolchains

A **toolchain** is a set of software which constitutes a close environment for the development of further software. The toolchain encapsulates:

- a compiler (and related standard library)
- a linker
- basic tools for software development
- basic libraries for scientific development

The software **must** be independent from the hosting OS.

How do I confine my work inside a specific toolchain?

The `Bash` environmental variables define paths for executables and libraries; the tool to manage these variables is commonly called the **environment module system**.
There are two main module systems, the one we use is named `Lmod` (the other is `TCL Environment Modules`).

A **module** is basically a set of instructions that define a specific environment for a specific software: where are the *executables*, where are its *libraries*, where are the *header files*, ...
<u>If the software is built inside a toolchain, the installation and the environment will be portable.</u>

A specific module is loaded and can be unloaded as well when not needed. The module system can define a hierarchy: a module may need another one, and the system automatically handles these dependencies.

Initialization of `mk`

```
$ source /u/sw/etc/bash.bashrc
```

```
$ module avail
---------- /u/sw/modules/toolchains ----------
gcc-glibc/11.2.0 (D)
```

Choice of the toolchain

```
$ module load gcc-glibc/11.2.0
```

Inside the toolchain I can find a lot of packages

```
$ module avail
```

Now a specific software package can be selected: usually many other requested modules will be automatically loaded

```
$ module avail
---------- /u/sw/pkgs/toolchains/gcc-glibc/11.2.0/modules ----------
R/4.1.0                 gsl/2.7          (L)    pybind11/2.6.2
adol-c/2.7.2     (L)    hdf5/1.12.0      (L)    qhull/2020.2       (L)
arpack/3.8.0     (L)    hypre/2.22.0     (L)    qrupdate/1.1.2     (L)
blacs/1.1        (L)    lis/2.0.30              scalapack/2.1.0    (L)
boost/1.76.0     (L)    matio/1.5.21     (L)    scipy/1.7.0        (L)
cgal/5.3                metis/5.1.0      (L)    scotch/6.1.1       (L)
dakota/6.15.0           mumps/5.4.0      (L)    suitesparse/5.10.1 (L)
dealii/9.3.1     (L)    netcdf/4.8.0     (L)    superlu/5.3.0      (L)
eigen/3.3.9      (L)    octave/6.2.0     (L)    tbb/2021.3.0       (L)
fenics/2019.1.0         openblas/0.3.15 (L)    trilinos/13.0.1    (L)
fftw/3.3.9       (L)    p4est/2.3.2      (L)    vtk/9.0.3          (L)
glpk/5.0         (L)    petsc/3.15.1     (L)
```

The module system takes care of the integrity of the definitions and does not permit bad cross-coupling (e.g. libraries with an incompatible `libc`).

To unload all the loaded modules:

```
$ module purge
```

The module system takes care of defining the environment variable `LD_LIBRARY_PATH` so the loader can find newly loaded libraries.
The user is responsible of passing compiler and linker flags when compiling, to do so they must know where libraries and headers are located.
The module system provides variables to make it easy to find headers and libraries of loaded modules

```
$ module load eigen
$ module list
Currently Loaded Modules:
1) gcc-glibc/11.2.0   2) eigen/3.3.9

$ echo ${mkEigenInc}
/u/sw/pkgs/.../eigen/3.3.9/include/eigen3
```

Naming scheme for in our toolchain

General naming scheme:

- Installation path: `"mk" + Modulename + "Prefix"`, *e.g.* `${mkOctavePrefix}`
- Headers: `"mk" + Modulename + "Inc"`, *e.g.* `${mkEigenInc}`
- Libraries: `"mk" + Modulename + "Lib"`, *e.g.* `${mkSuitesparseLib}`

How to use a third-party library

Basic compile/link flags:

```
$ g++ -I${mkLibrarynameInc} -c main.cpp
$ g++ -L${mkLibrarynameLib} -llibraryname main.o -o main
```

Warning: by mistake, one can include headers and link against libraries related to different installations/versions of the same library! The compile, link and loading phase may succeed, but the executable may crash, resulting in a very subtle yet painful error to debug!

Using shared libraries

Shared libraries:

- User point of view
- Developer point of view

Users need to take care to

- **linking** phase during compilation
- **loading** phase at execute time

Developers usually contribute to a

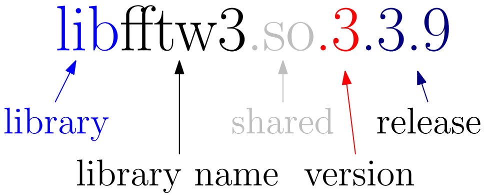
- **development** phase
- **release** phase

Versions and releases

The *version* is a symbol (typically a number) by which we indicate a set of instances of a library with a common public interface and functionality.

Within a version, one may have several *releases*, typically indicated by one or more numbers (major and minor or bug-fix). A new release is issue to fix bugs or improve of a library without changes in its public interface. So a code linked against version 1, release 1 of a library should work (in principle) when you update the library to version 1, release 2.

Normally version and releases are separated by a dot in the library name: `libfftw3.so.3.3.9` is version 3, release 3.9 of the `fftw3` library (The Fastest Fourier Transform in the West).



Naming scheme of shared libraries (Linux/Unix)

We give some nomenclature used when describing a shared library

- **link name**: name used in the linking stage when you use the `-lmylib` option.
- **soname**: *shared object name* looked after by the *loader* stage.
- **full name**: name of the actual file that stores the library.

Example:

`fftw`: is the *link name*.

`libfftw3.so.3`: is the *soname*.

`libfftw3.so.3.3.9`: is the *real name* of the file.

We call a *fully qualified library* a soname that contains the full path to the library.

How does it work?

The command `ldd` lists the shared libraries used by an object file.

Example:

```
$ ldd ${mkOctavePrefix}/lib/octave/6.2.0/liboctave.so
...
libfftw3.so.3 =>
     /u/sw/toolchains/gcc-glibc/11.2.0/pkgs/fftw/3.3.9/lib/libfftw3.so.3
...
```

- `Octave` has been linked to `libfftw3`.

The **loader** searches the occurrence of this library, finding his full qualified name.

Which release?

```
$ ls -l ${mkFftwLib}/libfftw3.so.3
/full/path/to/libfftw3.so.3 -> libfftw3.so.3.3.9
```

I am in fact using release `3.9` of version `3`.

Got it?

The executable (`octave`) contains the information on which shared library to load, including version information (its `soname`). This part has been taken care by the developers of `Octave`.

When I launch the program the loader looks in special directories, among which `/usr/lib` for a file that matches the `soname`. This file is typically a symbolic link to the real file containing the library.

If I have a new release of `fftw3` version 3, let's say 3.4.1, I just need to place the corresponding shared library file, reset the symbolic links and automagically `octave` will use the new release (this is what `apt` does when installing a new update in a Debian/Ubuntu system, for example).

No need to recompile anything!

Another nice thing about shared libraries

A shared library may depend on another shared library. This information may be encoded when creating the library (just as for an executable, we will see it later on).
For instance

```
$ ldd /usr/lib/x86_64-linux-gnu/libumfpack.so.5
...
libblas.so.3 => /lib/x86_64-linux-gnu/libblas.so.3
...
```

The UMFPACK library is linked against version 3 of the BLAS library.

This prevents using incorrect version of libraries. Moreover, when creating an executable that needs UMFPACK I have to indicate only `-lumfpack`!
Note: This is not true for static libraries: you have to list all dependencies.

How to link against a shared library

It is now sufficient to proceed as usual

```
g++ -I${mkFFtwInc} -c main.cpp
g++ -L${mkFFtwLib} -lfftw3 main.o -o main
```

The linker finds `libfftw3.so`, controls the symbols it provides and verifies if the library contains a `soname` (if not the link name is assumed to be also the soname).
Indeed `libfftw3.so` provides a `soname`. If we wish we can check it:

```
$ objdump libx.so.1.3 -p | grep SONAME
SONAME   libfftw3.so.3
```

(of course this has been taken care by the library developers).

Being `libfftw3.so` a shared library the linker does not resolve the symbols by integrating the corresponding code in the executable. Instead, it inserts the information about the `soname` of the library:

```
$ ldd main
libfftw3.so.3 => /full/path/to/libfftw3.so.3
```

The loader can then do its job now!

In conclusion, linking with a shared library is not more complicated than linking with a static one.

Remember: By default if the linker finds both the static and shared version of a library it gives precedence to the shared one. If you want to by sure to link with the static version you need to use the `-static` linker option.

Folders where the loader searches the shared libraries

- `/lib*` or `/usr/lib*` .
- Additional folders can specified by `/etc/ld.conf` and files inside the folder `/etc/ld.conf.d/`.

The command `ldconfig` rebuilds the data base of the shared libraries and should be called every time one adds a new library (of course `apt` does it for you, and moreover `ldconfig` is launched at every boot of the computer).

Note: all this operations require you act as superuser, for instance with the `sudo` command.

Alternative ways of directing the loader

- Setting the environment variable `LD_LIBRARY_PATH`. If it contains a comma-separated list of directory names the loader will first look for libraries on these directories (analogous to `PATH` for executables):

  ```
  export LD_LIBRARY_PATH+=:dir1:dir2
  ```
- With the special flag `-Wl,-rpath=directory` during the compilation of the executable, for instance

  ```
  g++ main.cpp -o main -Wl,-rpath=/opt/lib  -L. -lsmall
  ```

  Here the loader will look in `/opt/lib` before the standard directories. You can use also relative paths.
- Launching the command `sudo ldconfig -n directory` which adds `directory` to the loader search path (superuser privileges are required). This addition remains valid until the next reboot of the computer. **Note**: prefer the other alternatives!

How to build a shared library?

We will dedicate another lecture to this issue, where we will also show how to handle shared libraries and symbols dynamically. For the moment we need to know only the following:

- When building a shared library we need to pass the option `-shared` to the linker
- Object code used in a shared library must be *position independent* (compiler option `-fPIC`)

Basic build of a shared library starting from an object file:

```
$ g++ -shared -Wl,-soname,libutility.so utility.o -o libutility.so
```

Code used for the exercise

We employ the code in `LinearAlgebra/IML_Eigen` example.
The code is available here.

Exercise 1: compiler and linker options

- Compile `Utilities.cpp` and make it a shared library named `liblinearalgebra.so`
- Compile `test.cpp` and make an executable binary linked against `liblinearalgebra.so`

Exercise 2: a simple `Bash` script to automate repetitive operations

Create a simple `Bash` script to build the library and executable of Exercise 1 without typing the same commands every time.

Make and makefiles

Alberto Artoni

Controlling the compilation process

The compilation process requires to assemble 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 helps to automatize 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 user defined, or predefine,) 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 a well defined rule.

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

The basic layout of a makefile

Let's start with the simplest of Makefiles:

```
hello:
<TAB> echo "Hello, World"
```

A Makefile consists of a set of rules. A rule generally looks like this:

```
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 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 importing the value of 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` statement 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 =O0)
`-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 synamic 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)

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`

```
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.

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`.

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
```

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

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 inbuilt macros.

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: special 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 fist and last rule are not necessary. `make` already knows them (implicit rules)!. We will talk them soon.

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.

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 the manual in http://sourceware.org/autobook/.

An alternative to autotools is cmake.

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

Exercise 3: a Makefile

Create a Makefile that works as the `Bash` script in Exercise 3.