Fortran for Scientists

Helge Todt

Astrophysics
Institute of Physics and Astronomy
University of Potsdam

WiSe 2025/26, 27.11.2025

Libraries

Excursus: Libraries I

→ collection of functions, variables, operators

Including libraries in Fortran:

- at compile time:
  in contrast to C/C++: **no** header files (declaration of functions) needed
  (but: interfaces, modules)

  → no check for correct arguments in function/subroutine call
- at link time:
  look for the matching library that provides the function/subroutine translate the names (symbols)[†] used in the library to (relative) memory addresses;

  static linking: include the necessary library symbols in the program

[†] the list of symbol names of the compiled code can be printed out with `nm` *`file.o`*

Excursus: Libraries II

Dynamic libraries

The Unix command `ldd` lists the dynamically linked-in libraries for a given program (or object file/library), e.g., `ldd -v rcalc`:

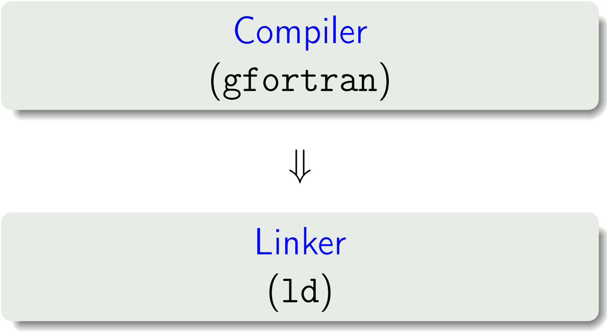
```
linux-vdso.so.1 (0x00007fff72bff000) [†]
libX11.so.6 => /usr/lib64/libX11.so.6 (0x00007f5eb8b22000)
```

The path to the library and the memory address is printed.

- at runtime:
  dynamic linking: loading program and libraries to memory (RAM)
  advantage (over static linking): library is loaded only once and can be used by other programs

[†] vdso = virtual dynamic shared object

Excursus: Libraries III



Compiler

(`gfortran`)

⇓

Linker

(`ld`)

Excursus: Libraries IV

Overview: Unix commands for developers

- `gfortran`: Fortran compiler
- `ld`: link editor (usually called by the compiler)
- `ldd`: lists the used libraries of an object file (also program or library)
- `nm`: lists the *symbols* of an object file (etc.)

Symbols

In a program `main` (= actual program, main function in C/C++) belongs to the symbols labeled with letter `T`. I.e., it is a symbol from the text (code) section of the file.

Linking and libraries I

- sometimes necessary for using some specific libraries: explicit specification (name) of the library at link time
- specification of a library `libpthread.so` via lower case `l`:

  ```
  -lpthread
  ```

  when calling the compiler for creation of the executables

  Example: `gfortran -o programm program.f -lX11`
- specification of the path to the library via upper case `L`:

  ```
  -L/usr/lib/ -lpthread
  ```

  when calling the compiler for creation of the executables
  Heads up: The path must be given before the library!
- Important: the corresponding header file must be in a standard path, the current directory, or the path is specified via `-I`*`path-to-include-file`*

Linking and libraries II

- dynamic libraries must be located in a default system path (e.g., `/lib`) or the the path must be added to the environment variable

```
LD_LIBRARY_PATH
```

E.g. for the `bash` via

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:.
```

and for the `csh` respectively

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:.
```

→ extending the path to dynamic libraries for the current working directory

Static linking I

- static libraries (file extension `.a`) are *archives* of object files
- these objects files are fixed included in the binary output during the procedure of static linking → can lead to large program files
- possible advantage: compact binaries with lean libraries (e.g., diet libc)

Sequence for static linking

If a library/program `libA` needs symbols from the library `libB`, the name of `libA` must be given before that of `libB` at link time for static linking: `-lA -lB`

- (complete) static linking isn't supported anymore by modern OSs (e.g. MacOS) at normal developer level
- but against some libraries (e.g., `libgfortran`, MKL) it can be selectively statically linked

make

make I

Purpose of `make`:

- automatic determination of the program parts (usually source files) that must be re-compiled via
    - a given definition of the dependencies / prerequisites (implicit, explicit)
    - comparison of time stamps (file system)
- calling the required commands for re-compilation:

typical use: `./configure ; make ; make install`
useful especially for large programs with many source files

make II

Main idea of make is the *rule*:

```
Target :  Dependencies
<TAB> command for creation of the target
```

e.g.,

```
myprogram :  myprogram.o
<TAB> gfortran -o $@ $?
```

Note

- *explicit rules* are defined via an ASCII file, the so-called *makefile*
- every command belonging to a rule must start with a `<TAB>`!
- the macros `$@` and `$?` are called *automatic variables*, i.e., they are replaced by `make`:
  `$@` is replaced by the target,
  `$?` by the dependencies that are *newer* than the target
  `$^` → all dependencies (separated by blanks)

make III

Implicit rules:

- some rules for compilation are re-occurring, e.g., for Fortran `.o` files are always created from `.f` files
- `make` has therefore a number of implicit rules, hence `make` can also be used without a `makefile`

Example

```
echo '      END' > myprog.f
make myprog
```

executes `f77 myprog.f -o myprog`[1]

- `make` uses implicit rules if no explicit rule for creation of the target has been found

[1] `make` invokes `f77` automatically, or the Fortran compiler that is specified in the environment variable `FC`, e.g., `export FC=gfortran`

make IV

Explicit rules

- an explicit rule is usually specified in a text file that has one of the following default names: `makefile`, `Makefile`
- every rule must define at least one target
- it is possible to define several dependencies for one target
- a rule can contain an arbitrary number of commands

Moreover, explicit rules overwrite implicit rules:

```
.c.o :
<TAB> $FC -c $?

$(PROJECT) : $(OBJECTS)
<TAB> $(FC) $(FFLAGS) -o $(@) $(OBJECTS)
```

make V

Usual run of a `make` call:

1. after calling `make` the `makefile` is parsed (read)
2. read and substitute variables (see below) and determination of the highest target(s) (given in the beginning), evaluation of the dependencies
3. creation of a *tree* of dependencies
4. determination of the time stamps for all dependencies of the corresponding files and comparison with those of the next step in the tree
5. targets whose dependencies are newer than the targets are re-compiled

make VI

Variables

- during processing of the rules make uses automatic variables, e.g., $@ and $? (see above)
- variables can also be defined explicitly before the first rule, syntax is shell-like:

```
FC = gfortran
FFLAGS = -3
PROJECT = galaxy
```

- variables can, as in the shell, be held together with help of curly braces ${OBJECTFILES}, or alternatively with help of round parentheses $(FFLAGS)

make VII

Usual pseudo targets → Call via `make` *pseudo target*

- don't create a file, or don't have dependencies, e.g.
- `clean`, for `make clean`, defines explicitly how the intermediate and final products (targets) of the compilation shall be removed
- `all` creates all project files
- `install` if the targets (programs, libraries) shall be copied to a specific directory (or similar), it should be stated in `install`

Pseudo targets (e.g., `clean`) can only be used if defined in the `makefile`.

make VIII

Example of a makefile

```
FC = gfortran -O3
FFLAGS = -Wall
LIBRARIES = -lX11

OBJECTS = componentA.o componentB.o
PROJECT = myprogram

$(PROJECT) : $(OBJECTS)
        $(FC) $(FFLAGS) $(OBJECTS) -o $@ ${LIBRARIES}

.f.o :
        $(FC) -c $(FFLAGS) $?

clean :
        rm -f $(OBJECTS)
```

make IX

`Makefile` uses a shell-like syntax:

- comments are started with a #:

  ```
  # a comment
  ```

- one command per line, multiple commands via ; and line continuation via \

  ```
  $FC $?  ; ldconfig
  ```

- every command corresponds to a shell command, and is printed before execution:

  ```
  .f.o :
        echo "Hello ${USER}"
  ```

  the print-out of commands can be suppressed with @ before the command

  ```
        @echo "Hi ${DATE}"
  ```

make X

- variables are set without $ and used/referenced with a $

```
progname = opdat
PROJECT = $(progname).exe
```

Variable names that contain multiple characters should always be held together with parentheses () or curly braces {}.

make XI

Special targets:

- problem: pseudo target `clean` is not executed, if a *file* with that name exists (why?)
- solution: pseudo targets can be marked as such via the *special target* `.PHONY`:

  ```
  .PHONY: clean install
  ```

- special targets start with a `.`

make XII

Some more special targets:

- `.INTERMEDIATE` : dependencies are only created if another dependency before the target is newer, or if a dependency of an intermediate file is newer than the actual target. The intermediate target is deleted after the target was created:

  ```
  .INTERMEDIATE : colortable.o

  xapple.exe : xapple.f colortable.o
          $(FC) -o xapple.exe xapple.f colortable.o

  colortable.o : colortable.f
          $(FC) -c colortable.f
  ```

  Here, `colortable.o` is only created if `xapple.f` or if `colortable.f` are newer than `xapple.exe`. After the creation of `xapple.exe` the target `colortable.o` will be removed.

make XIII

- `.SECONDARY` : like `.INTERMEDIATE`, but the dependencies are not removed automatically
- `.IGNORE` : errors during creation of the specified dependencies will not lead to an abort of the make procedure

Hint

The tool `make` is not bound to programming languages, but can also be used for, e.g., automatic compilation of `.tex` files etc.

Advantage of using `make`

A `Makefile`

- can save compilation time
- documents the compiler options and necessary files of the project