Scientific Software
Compilation with terminal commands

Koen Poppe Peter Opsomer

16 february 2010 September 22, 2015

1. **Source code** — Everything starts with the program code: in the case of Fortran, the files typically get an extension *.f*, *.f77*, *.f90*, *.f95* or *.f03*, depending on the standard used. Certain compilers use this extension to figure out which format is used (*fixed* or *free*) and which standard was employed, other compilers expect that this is indicated using compile flags. To enforce full compatibility with one specific standard, one needs to explicitly state this using a specific compilation flag. Remark that using such *extensions* for a specific standard will also be reported.

2. **Compile** — The (Fortran) code is hereby converted to an *object file*. This contains the machine code, but also extra information such as the names of routines and global variables. In this step, the full syntax is of course also checked as well as whether each function and routine used is available and, if possible, whether it contains the correct arguments.

   Besides that, a *module file* is also made. This contains all information about the interfaces of the functions and subroutines that are declared in the modules of the file. This is useful if a program consists of multiple files: like this, one can check whether the right number and type of arguments is used via this module file (object files do not contain such information!). These files are automatically looked up in the current folder; if they are elsewhere, that path has to be given to the compiler.

   Multiple object files can be grouped in a *library*. That way, one avoids having to designate all individual object files when linking. Remark that a library, like an object, does not contain information about the interfaces. Therefore, one has to be careful and check meticulously whether the arguments are correct using the documentation. If not, one can get errors when executing (often a *segmentation fault*).

3. **Link** — After all code is verified and converted to object file(s), one can compose the final program. Hereby, all names of functions and subroutines are replaced by their correct addresses. If a library was forgotten or certain procedures cannot be found, one gets an error message. In that case, it may be necessary to add the path to the directory containing the library (`-L`) or other files (`-I`). However, `-L/usr/lib/lapack` should not be necessary, since it has been installed; documentation can be found on for example `http://www.netlib.org/lapack/single/`. When everything proceeds correctly, one gets a program of which the name corresponds to the `-o` option or `a.out` when not given.

   It is advised to use the same compiler for compiling and linking as much as possible because when compiling, references are appended to specific compiler dependent implementations of intrinsic functions (ex. `print` or `cos`). When linking with the same compiler, the necessary libraries are also added automatically. One can of course use any linker, but then he has to specify these libraries himself.

Using object files avoids having to compile all code at every turn: source files that have not changed do not have to be compiled again if their object files are available. This shortens compilation time for large projects, but for small programs this might not be worth the effort. In that case one can immediately obtain the program by giving all source files to the compiler. This alternative is also shown in the schedule of Figure 1. There, `f95` does use the `gfortran` compiler.

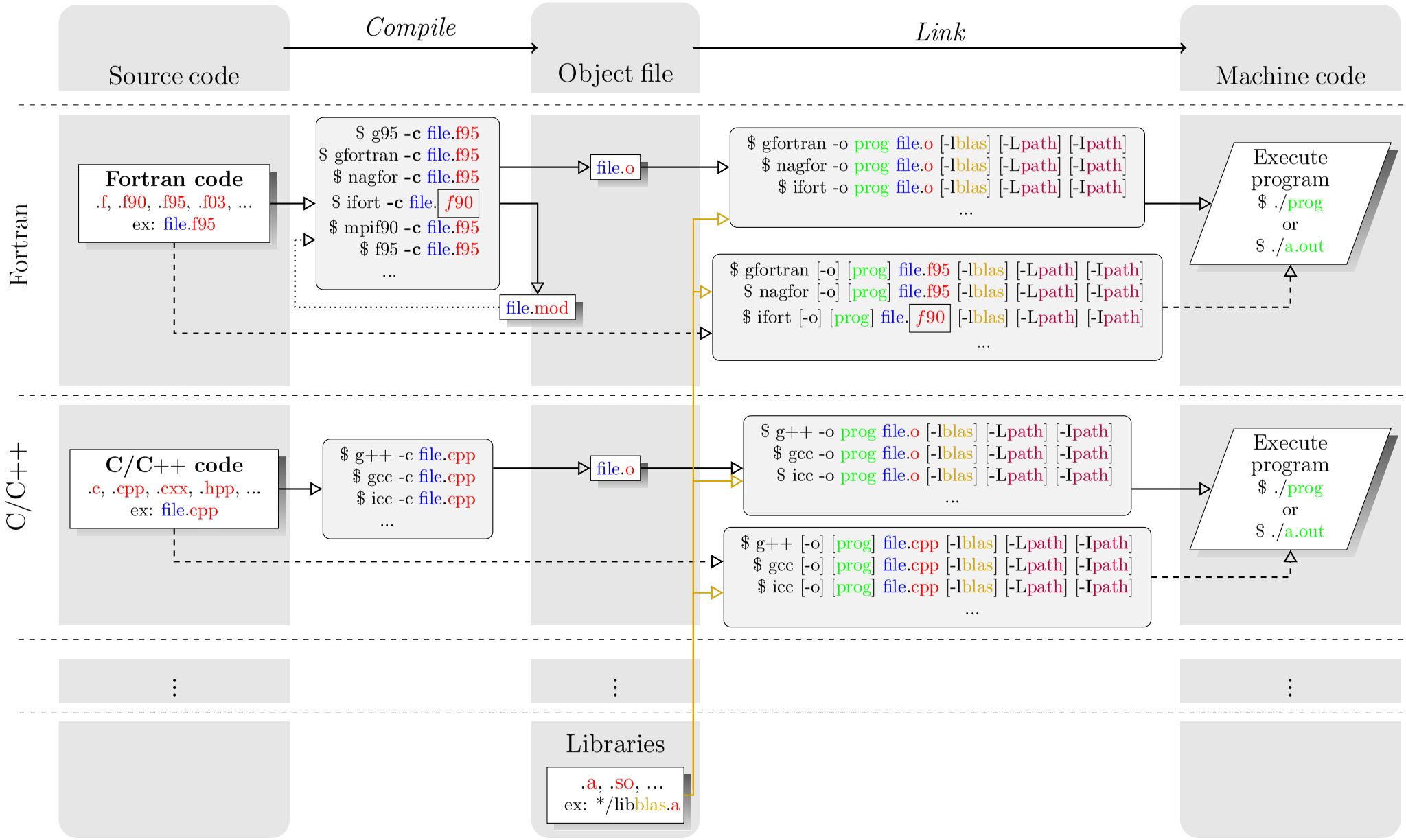

Figure 1: Schematic representation of the compilation process that deviates slightly for Fortran code from what one is used to in C or C++.