P6 – Scientific Programming

Marcus Mohr
Jens Oeser

Geophysics Section
Department of Earth and Environmental Sciences
Ludwig-Maximilians-Universität München

SoSe 2021

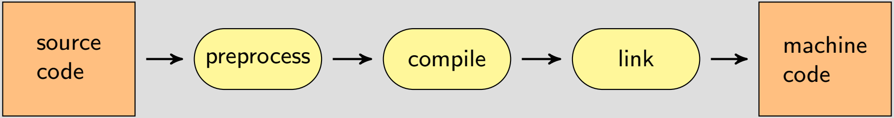
Part #7

Editing & Compiling C Source Code

'under Linux (with GCC and Emacs)'

Steps of Compilation

- To execute our C program on a computer it must be converted into machine code. Collectively refered to as “compilation”.
- The C standard defines eight stages for this translation, which can be grouped into the following:



- We will focus on compiling and linking for the moment.

Compiling Code

Compilation

A compiler is a program that transforms a file containing source code into an (object) file containing machine code.

Linking

A linker takes one (or more) object files and maybe also additional library files and combines them into an executable.

- typically the same program/frontend acts as compiler and linker
- compiling and linking can be performed in one step/program call

C Compilers

The following table gives an overview on the compilers currently available on the Geophysics system

compiler	belongs to
gcc	GNU compiler collection (GCC)
clang	Clang C, C++, and Objective-C compiler (LLVM)
icc	commercial compiler by Intel
pgcc	commercial compiler by PGI

- GCC and Clang are open-source projects/software.
- C Compilers by GCC, Clang and Intel are binary compatible and have mostly identical command-line options.

Remarks on GCC

GCC stands for GNU Compiler Collection. This is a suite of compilers. Implementation-wise the suite uses a common middle- and back-end for compilation, optimisation and linking and provides front-ends for supporting different languages

front-end	language
g77	FORTRAN 77 (phased out)
gfortran	Fortran{77, 90, 95, 2003, 2008}
gcc	C
g++	C++
gcj	Java
gnat	Ada

Compiler Switches

- the table below lists those compiler switches (= command line options) of gcc most important for beginners
- for complete list see e.g. `man gcc`

switch	meaning
`-c`	prevents linking; only object (.o) file compiled
`-o <outfile>`	name output file `outfile` instead of default (e.g. `a.out` for executables)
`-Wall`	enables commonly used warning options pertaining to code structures that are better avoided
`-std=<std>`	conform to the specified standard; allowed values for `<std>` are e.g. c90, c99 or gnu11 (default)
`-g`	adds symbolic debug information in the object file
`-O<level>`	optimise program performance, `<level>` $\in \{0,1,2,3\}$ controls aggressiveness

Some Naming Conventions

Source Files

compilers under Linux/Unix typically assume that a file contains C language source code, if it ends with the postfix `.c`, e.g. `HelloWorld.c`

Object Files

the names of object files generated by the compiler typically end with the postfix `.o`, e.g.

`gcc -c HelloWorld.c` ⟶ `HelloWorld.o`

Executables

there is no convention on the names of executables under Linux; by default most compilers will name the executable `a.out`

Examples

command line	resulting file	type
`gcc HelloWorld.c`	`a.out`	executable
`gcc -c HelloWorld.c`	`HelloWorld.o`	object file
`gcc HelloWorld.o`	`a.out`	executable
`gcc -o HelloWorld HelloWorld.c`	`HelloWorld`	executable
`gcc -o myObj.o -c HelloWorld.c`	`myObj.o`	object file
`gcc -o myObj -c HelloWorld.c`	`myObj`	object file

A closer look

We can use Linux' `file` command to inspect the file type

filename	output of `file <filename>`
HelloWorld.c	C source, ASCII text
HelloWorld.o	ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
a.out	ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/l, for GNU/Linux 3.2.0, BuildID[sha1]=c488f2b446d8a5a1026da151b192268602a3eb2c, not stripped

A closer look (cont.)

the nm command lists the symbols contained in a file

```
nm HelloWorld.o
         U _gfortran_set_std
         U _gfortran_st_write
         U _gfortran_st_write_done
         U _gfortran_transfer_character
00000000 T MAIN__
```

T: symbol is defined in the file; MAIN__ is our HELLO program

U: symbol is undefined; these are e.g. functions implementing the machine code behind our PRINT statement ⟶ run-time environment

⟶ linker takes care of handling these dependencies

Types of Linking (1/2)

static

static linking means that all undefine/imported symbols (like external functions & variables) are inserted into the executable

- makes executable transferable and independent of installed libraries
- size of executable may grow considerably

dynamic

dynamically linked executables only contain references to undefined / imported symbols and some information where to find them at execution time

Types of Linking (2/2)

- mixed approach is possible
- today's default is to perform dynamic linking (at least w.r.t. standard libraries)
- `ldd` command can be used to print shared library dependencies

```
=> ldd HelloWorld
   linux-vdso.so.1 (0x00007ffc7159e000)
   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd03e52e000)
   /lib64/ld-linux-x86-64.so.2 (0x00007fd03eb21000)
```

Editing Source Code

basic requirements of a text editor for programming

- must save files as plain text files (not `.doc`, `.odt`, `.rtf`, `...`)
- should provide fixed size font

ideally it should also support helpful features like

- syntax highlighting
- formatting helps, like e.g. auto-indentation

Emacs

The two most commonly used editors for programming under Linux/Unix are vi/vim and Emacs.

The suggested editor for this course is Emacs. It supports many useful features like, e.g.

- syntax high-lighting
- formatting helps, such as
  - correct indentation
  - automatic line-breaking
  - a fortran ruler
- compilation from within the editor
- interfacing with version control software
- and much more

IDEs

- Even more features (i.e. support for the programmer) are provided by full Integrated Development Environments (IDE).
- An IDE could e.g. support to easily rename a variable/class used in multiple source files.
- For more details and comparisons see Wikipedia: Integrated Development Environment
- A prominent free-software example is eclipse