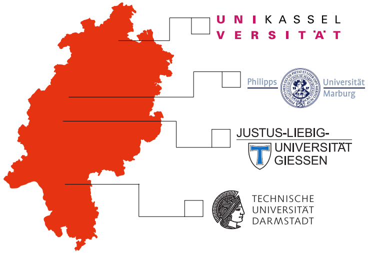
Building Software in Linux

Hessisches Kompetenzzentrum für Hochleistungsrechnen (HKHLR)

Tim Jammer / Dr. Marcel Giar

26.04.2023


UNIKASSEL VERSITÄT
Philipps Universität Marburg
JUSTUS-LIEBIG-UNIVERSITÄT GIESSEN
TECHNISCHE UNIVERSITÄT DARMSTADT

HKHLR is funded by the Hessian Ministry of Sciences and Arts

Motivation

- Not all software is pre-Installed on an HPC system
- How can you install software yourself?
- As Linux software is often open source, compiling it from source code is quite common
- Errors can happen at different stages of compilation
  - ⇒ Basic understanding of the compilation process will help to resolve potential errors quickly

Table of Contents

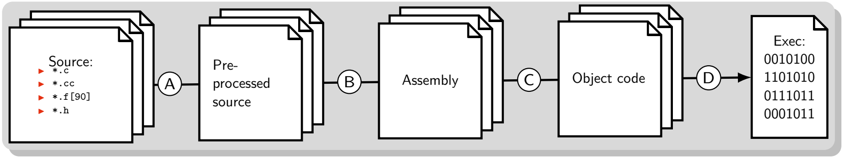
From code to binary: Details

Definition (Compilation)

The transformation of code written in one programming language (source) into another programming language (target).

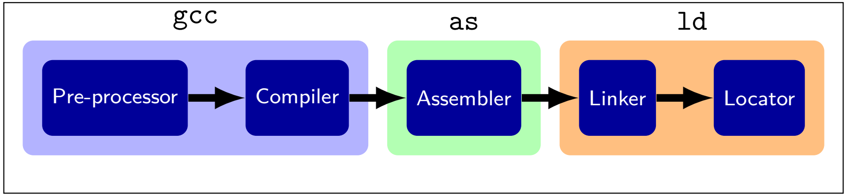
- To generate a binary, all source code must be available, i.e.:
  - All functions (methods, member-functions, etc.) declared
  - All functions defined
- Compilation: A four-step process

(A) Pre-processing
(B) Compilation "proper"
(C) Assembling
(D) Linking



GNU’s tool collection - GCC toolchain

Complete tool-chain can be accessed with `gcc` frontend



- GNU project ships complete set of tools to manage the build process
  - GNU Compiler Collection (GCC)
    - ★ Pre-processor: `cpp`
    - ★ Compiler: `gcc`
  - GNU Binutils (`https://www.gnu.org/software/binutils/`)
    - ★ Assembler: `as`
    - ★ Linker: `ld`

Invoking a compiler from the command line

Like many other UNIX utilities, compilers must be passed options and files as parameters:

```
<CompilerName> <Options> <InputFile> [-o <OutputFile>]
```

An example for generating an object file from a source file:

```
gcc -Wall -O0 -g -c SourceFileName.c -o SourceFileName.o
```

Command	"Value"
`CompilerName`	`gcc`
`Options`	`-Wall -O0 -g -c`
`InputFile`	`SourceFileName.c`
`-o OutputFileName`	`-o SourceFileName.o`

Pre-processing - Includes and defines

- Pre-processing: "Simple" text-manipulation
- Invoke the pre-processor from the command line:
  - Call `cpp`: `cpp <filename>.c -o <filename>.i`
  - Compiler frontend: `gcc -E <filename>.c -o <filename>.i`
- Pre-processor handles source code lines starting with "`#`"

```c
#include <stdio.h>
#include <stdlib.h>
#include "myfft.h"

#ifdef __USE_FFTW
#define FFT_SIZE 1024
#endif

int main() {
  // code
}
```

- Includes:
  - Compiler argument: `-I@PATH@`
  - `#include <...>` and `#include "..."`
    - ★ These behave differently in where to look for include files (see next slide)
- Defines:
  - Compiler/Pre-processor argument `-D@DEFINE@=@VALUE@`
  - `#ifdef ... #endif`

`cd Demos/01_includes-and-defines`

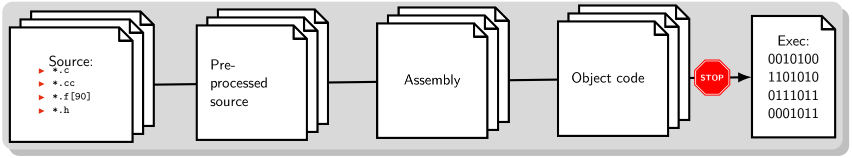
`> make DEMO`

Pre-processing demo - What did we just learn?

- Use pre-processor as `gcc -E` or `cpp`
  - Includes header files into source files (system header + user's headers)
  - Evaluated macro definitions (using argument `-D@DEFINE@=@VALUE@`)
- Include styles: `#include "..."` vs. `#include <...>`
  - `#include <...>`
    - ★ Used for system header files
    - ★ Self-defined paths are searched first; them system paths
  - `#include "..."`
    - ★ Used for *user's own* header files
    - ★ *Current* directory (where the source file is located) is searched first; then system paths
- Augment search paths for includes with `-I@PATH@` option
- Monitor include paths with `gcc -E -v` or `cpp -v`

Object files - What and what for?

Compile to object



- Object files:
  - "Portable" partial compiled output of a compiler
  - Code is transformed but function/method names remain unresolved
  - Enabled by the `-c` option
- Object files are used to:
  - Speed up compilation process
  - Enable parallelization of compilation
  - Enable cross- language compilation
  - **Attention:** Compilers are only **mostly** binary compatible

The linker - Object files, static & dynamic libraries

- Object files must be combined to build a binary
- Linker processes all symbols and resolves addresses
  - Object files (`*.o`), generated by compiling with `-c`
  - Archive files (`*.a`), generated by `ar`
  - Dynamic libraries:
    - ★ Search paths are added via `-L@PATH@`
    - ★ Environment variable `LD_LIBRARY_PATH` contains runtime library paths
    - ★ When given `-l@LIBNAME@`, the linker will look for the file "`lib@LIBNAME@.so`"
    - ★ A library must be present both in the compiler's/linker's search path (`-L`) for compilation, and in the dynamic linker path (`LD_LIBRARY_PATH`) for running
- Compiler runtime environment is **not** portable

```
cd Demos/04_missing-symbols
```

```
> make DEMO
```

Compiler demo - What did we just learn?

```
undefined reference to 'vectorf_from_array'
```

- Linker was not able to resolve reference to `vectorf_from_array`
  - Object file with function definition missing
  - Dynamic or static library with function definition missing
- Possible solution: Provide library to linker ⇒ `-L/path/to/lib -l@LIBNAME@`

```
libarray.so: cannot open shared object file: No such file or directory
```

- *Dynamic linker* failed to load shared libraries for binary

```
> LD_LIBRARY_PATH=/path/to/lib:${LD_LIBRARY_PATH} ./program
> # or
> export LD_LIBRARY_PATH=/path/to/lib:${LD_LIBRARY_PATH}
> ldd ./program # check dynamically linked dependencies
```

GCC compiler flags - Optimization

- Optimizations `-O[0-3]`, `-Ofast`, `-Og`
  - `-O0`: Mostly disable optimizations, reduce compilation time and make debugging produce the expected results
  - `-O2`: Good compromise of optimizations and compile time, should be numerically identical to `-O0`
  - `-O3`: Aggressive optimizations, **slight changes in numerics possible**
  - `-Ofast`: Most aggressive optimizations, **no guarantee for standard IEEE floating point semantics compliance**
  - `-Og`: Minimal optimizations, designed for minimal interference with debugger
- Debugging: `-g` and either `-Og` or `-O0`

Optimization during development

Always use `-O0 -g` during development when checking correctness of your code (during the "edit-compile-debug" cycle).

Rule of thumb

Code should compile equally well with `-O0` and `-O3`.

GCC compiler flags - Warnings, errors, and architecture

- Demand ISO C/C++ standards: `-pedantic` (and `-pedantic-errors`); see `https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html`
  - Issue warnings demanded by strict ISO C/C++
  - ISO C: Follows value of `-std=<value>` flag
  - Rejects GNU extensions
- Warnings: `-Wall` and `-Wextra`, e.g.:
  - `-Wunused-function`
  - `-Wunused-variable`
- Turn all warnings into errors: `-Werror` (**aborts compilation!**)
- Architecture-specific optimizations (e.g. AVX, AVX2 or AVX512): See `https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html`
  - `-march=native -mtune=native` or `-mavx -mavx2 -mavx512f`
- Pointer aliasing: `-fstrict-aliasing` and `-Wstrict-aliasing`

```
cd Demos/05_compiler-flags
```

```
> make DEMO
```

Compiler demo - What did we just learn?

- Enabling optimization (e.g. `-O3` vs. `-O0`) can yield **much** faster code
  - Obtainable speedup depends on problem!
- Compiler applies source code transformations while optimizing
  - Assembler code differs between `-O0` and `-O3`
- Using `-g`: Additional symbols in assembler code
  - Needed by debugger
  - Should always be used during development, irrespective of optimization level!
- Architecture optimizations can yield performance gain
  - Oftentimes library routines (e.g. from Eigen) are written correspondingly
  - **Attention:** In case the architecture on the compilation host differs from the architecture of the execution host (e.g. if the cluster headnode has a different CPU type than the compute nodes), `-march=native -mtune=native` might be less optimal or even incompatible

GNU Make

- Common choice for build system in the Linux command line environment: GNU `make`
- *Not* part of the GCC toolchain ⇒ Can be used with any compiler!
- The `make` utility helps us compile and link a software in order to use it on Linux (e.g. on a HPC cluster)
- `make` will take care of all steps of compilation:

  (A) Pre-Processing
  (B) Compilation proper
  (C) Assembling
  (D) Linking

- `make` offers an extra level of abstraction on top of the frontents provided by compiler toolchains (e.g. GCC)
- We can configure the build process by using *Makefiles*

Toy software project - Automatic build

File: ./Demos/00_toy–project/makefile.simple

```
 1 # SPDX-FileCopyrightText: 2021 Competence Center for High Performance Computing in He
 2 # SPDX-License-Identifier: MIT
 3
 4 all: program.exe
 5
 6 program.exe: array.c helper.c linalg.c main.c
 7   gcc -c -O0 -Wall -Wextra -pedantic array.c -o array.o
 8   gcc -c -O0 -Wall -Wextra -pedantic helper.c -o helper.o
 9   gcc -c -O0 -Wall -Wextra -pedantic linalg.c -o linalg.o
10   gcc -c -O0 -Wall -Wextra -pedantic main.c -o main.o
11   gcc -o program.exe array.o helper.o linalg.o main.o
```

- Line 3: Executable depends on the source files (a rule)
- Lines 4-7: Generate object files from C sources (a recipie)
- Line 8: Link the object files into an executable (a recipie)

Too much repetitive content in this file!

Makefiles - What is contained inside?

`make` configuration

To configure the `make` build system, we must add things to our Makefile. These ingredients comprise a set of instructions specifying how to generate an executable from our source files.

- Variables:
  We must specify ...
  - Source files
  - Compiler and compiler flags
  - Libraries to link and headers to include
- Rules:
  We must specify *what* to build
- Recipes:
  We must specify *how* to build it

Toy software project - Makefile refactored

File: Demos/00_toy–project/makefile

```
.SUFFIXES:
.SUFFIXES: .c .h

all: cprogram
sync_files:
  bash ./setup_demo.sh --prepare

CSources = array.c helper.c linalg.c main.c
CObjects = $(CSources:%.c=%.o)
ExeFile  = program.exe

CC      = gcc
CFLAGS = -Wall -Wextra -pedantic -O0 -g

vpath %.c ./
vpath %.h ./

cprogram: $(ExeFile)

$(ExeFile): $(CObjects)
  $(CC) $(CFLAGS) -o $@ $^

%.o : %.c
  $(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
  @rm -fv *.o $(ExeFile)
  @bash ./setup_demo.sh --clean
```

- A simple text file ⇒ Can track changes in a version control system (e.g. git)
- More compact but harder to understand
- Variables: Set compiler and flags
- Rules and recipes: Customize the build process
- Contains all commands to compile applications

Important notice

The `make` utility processes Makefiles in a nonlinear fashion!

Variables in Makefiles

Variables are defined in a Makefile by assigning *values* to them.

- There are some obvious advantages:
  - Enhance legibility of Makefiles → More self-explanatory
  - Allows quick adaption of compile options
  - Increase portability of whole build system
- Without using variables, commands become cluttered, e.g.:

```
program.exe: array.o helper.o array_ops.o main.o
  gcc -O3 -mtune=native -march=native -Wall -Wextra -g -o program.exe \
  array.o helper.o array_ops.o main.o -lm
```

- Using variables can make the same operation look compact and concise:

```
$(EXE): $(OBJS)
  $(CC) $(CFLAGS_OPT) $(CFLAGS) -o $@ $^ $(LIBS)
```

Variables - Definition

We can assign values to variables ourselves

```
variable_name = string
```

- Setting compilers:

```
# Fortran compiler
FC = gfortran
# C compiler
CC = gcc
```

- Defining object files:

```
# Object files
OBJS = utils.o mathfunc.o main.o
```

- Setting names of executables (choose suffix as you like):

```
# Name of compiled executable
EXE = program.x
```

Variables - Access

Accessing the value of a variable

Use `$(variable_name)` to access the content of variables

For example (details of this syntax will be explained later):

```
$(EXE) : $(OBJS)
  $(CC) -o $(EXE) $(OBJS)
```

It can be used to define additional variables:

```
CFLAGS_BASE = -Wall -std=c11 -O2
CFLAGS_DEBUG = -g
CFLAGS = $(CFLAGS_BASE) $(CFLAGS_DEBUG)
```

Case sensitivity

`$(variable_name)` is **not** the same as `$(VARIABLE_NAME)`!

Rules and recipes - What to build and how to build it

Explicit rules[5]

A "rule" within a Makefile connects *targets* and *prerequisites* by specifying a *recipe* of how to (re-)create the target files from the prerequisite files (e.g. compilation, linking, building an archive).

General rule syntax:

```
targets: prerequisite1 prerequisite2 prerequisite3 ...
	command1 # this is a <tab> indentation!
	command2
	command3
	...
```

- Targets: List of file names separated by spaces (usually only one per rule)
- Prerequisites: List of file names separated by spaces (oftentimes more than one per rule)
- Recipe: List of commands indented with a `<tab>` character, prescription for how to make the target (interpreted by the `shell`)

Recipes of Makefiles

Recipes

A recipe of a corresponding rule consists of `shell` commands that are executed one after the other.

In Makefiles, we really have two types of syntax combined:

- `make` syntax used for rules and defining variables
- `shell` syntax for recipes (default: interpreted by `/bin/sh`)

How to distinguish recipes from the rest of a Makefile

Recipes must *always* start with a `<tab>` character. Anything indented with a `<tab>` in a Makefile will be considered part of a recipe and therefore is passed to the `shell`.

Pattern rules in Malefiles

Pattern rules

These rules contain a % in the target; the target is a file name matching the pattern.

General syntax: Object files from sources (SUFFIX $\rightarrow$ .o):

```
%.o: %$(SUFFIX)
    command1
    command2
    ...
```

- % in a prerequisite *must* match the same substring as in the target's pattern (e.g. `main.o : main.c`)

Availability of prerequisites

In order to be applicable, the prerequisites must either exist or there must be rules for making them.

Rules and recipes in Makefiles - Examples

- Compile sources and link to executable:

```
sources := $(wildcard $(SRCPATH)/*.c)
objects := $(subst $(SRCPATH)/,,$(sources:%.c=%.o))

$(executable): $(objects) # make executable from objects
  $(CC) $(CFLAGS) -o $(executable) $(objects)

%.o : %.c # make objects from C sources (pattern rule)
  $(CC) $(CFLAGS) $(INCS) -c $< -o $@
```

Rules - How do we execute a rule?

Assuming we have defined the following rules:

```
program: array.o linalg.o main.o
	$(CC) $(CFLAGS) -o $@ $^
%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@
```

How can we tell `make` to build the target `program`?

```
> make program # Pass target to make
```

- If all `*.o` files exist, the rule `$(CC) $(CFLAGS) -o $@ $^` will be executed
- If `*.o` files are missing: Build object targets for all `*.c` files first, e.g.:

```
array.o: array.c
	$(CC) $(CFLAGS) -c array.c -o array.o
```

Automatic variables in pattern rules

In the previous examples, we have made use of so-called *automatic variables*[1].

Automatic variables

Variables whose values are computed anew whenever the rule involving them is executed; they are based on the target or the prerequisites used in the rule. **They are only valid within the scope of the corresponding recipe.**

Automatic variable	**Meaning**
`$@`	File name of the target in a rule.
`$<`	The name of the first prerequisite.
`$*`	The substring with which the pattern rule matches.
`$^`	The name of *all* prerequisites, separated by spaces.

Searching for prerequisites - VPATH variable

The VPATH variable[2]

VPATH specifies *all* directories in which make shall search for the files needed in the prerequisites used in the rules defined in the Makefile. The order in which directories are listed is the order used by make in its search.

Generic syntax for defining VPATH:

```
VPATH = directory_1:directory_2:...
```

Example: Source tree that has different subdirectories

```
VPATH = src/tools:src/parser:src/math

%.o : %.c # with VPATH: Equivalent to %.o : src/tools/%.c:...
  $(CC) $(CFLAGS) -c $< -o $@
```

⇒ make knows that it must search *.c-files in the directories specified in the VPATH variable.

Fine-grained search for prerequisites

The `vpath` variable[2]

A more fine-grained configuration of search paths for prerequisites can be obtained with the `vpath` variable (lower case!) by specifying the patterns for file names.

Generic syntax for defining `vpath`:

```
vpath <pattern> <directories> # dir_1:dir_2:...
```

Example: Tell `make` to look for `*.c` and `*.h` files in these directories:

```
vpath %.c src/c
vpath %.h src/headers
```

⇒ Can be useful if different source types are placed in different directories in the source tree.

Goal when using .PHONY targets

Force certain rules to be executed *every time* they are explicitly invoked.

Example[4]: `make clean` (shall *always* execute recipe for `clean`!)

```
clean:
  rm -vf *.o
  rm -vf *.exe
```

⇒ `make clean` will not work if a file named "`clean`" exists

⇒ `clean` would always be considered up to date since it has no prerequisites that change over time

```
.PHONY: clean # make it a prerequisite of built-in .PHONY target
clean:
  rm -vf *.o
  rm -vf *.exe
```

Conventions - Some more things to adhere to in Makefiles[3]

- Program name variables should also have a flags variable:

```
CC = ...         # C compiler
CXX = ...        # C++ compiler
FC = ...         # Fortran compiler
CFLAGS = ...     # Flags for C compiler
FFLAGS = ...     # Flags for Fortran compiler
CXXFLAGS = ... # Flags for C++ compiler
```

- Use extra variables for compiler options that are mandatory for properly building the application: `CFLAGS_BASE = ...`
- Place `CFLAGS` variable as last compilation command (usually `CFLAGS` is user-defined; override other variables):

```
%.o : %.c
  $(CC) $(CFLAGS_BASE) $(CFLAGS) -c $< -o $@
```

`CFLAGS` should occur in *every* invocation of `$(CC)` (compilation *and* linking!)

Running `make` - Some examples

- `make` by default searches for files named (in order!):

  (1) `GNUmakefile` (2) `makefile` (3) `Makefile`

Some useful command line arguments are:

- Manually specify the Makefile to use:

  ```
  > make -f|--file <filename>
  ```

  ⇒ For the default names, no command line argument is needed!

- Make a "dry run": Do not actually run the recipes, just print the commands that would be executed:

  ```
  > make -n|--dry-run
  ```

- Change the directory to `DIRECTORY` before doing anything:

  ```
  > make -C|--directory DIRECTORY
  ```

Running `make` for specific targets

Use `make` to build certain targets

By passing the name of specific *targets* to `make` we can execute the *recipes* needed to build them: `make <target>`

Example:

```
all: kkrlin kkrpara # first target is compiling everything
kkrlin:
  <recipe_for_kkrlin>
kkrpara:
  <recipe_for_kkrpara>
```

- `make kkrlin`: Executes the recipe for rule `kkrlin`
- `make kkrpara`: Executes the recipe for rule `kkrpara`

Standard targets in Makefiles

Commonly used targets in Makefiles:

- `all`: Compile the entire program. Should be the first target in the Makefile. As a default, `make` should compile with the `-g` option to be able to debug the program if it crashes.
- `clean`: Delete all files that were created by executing `make` (with some additional options).
- `distclean`: Delete all files created by configuring or building the program. This option becomes particularly interesting if the program to be built comes with a `configure` file.
- `install`: Install the program to some destination directory after building.
- `check`: Run self-tests (if there are any). The program must be built first before executing the recipe for this target.

Variables from the command line - Overriding defaults

Overriding defined variables

Variables defined in Makefiles can also be passed or changed from the command line when calling `make`. *These values take precedence* over the variables defined inside the Makefile.

The typical syntax is as following:

```
> make var_1=value_1 var_2=value_2 ...
```

Example:

```
CC = gcc
CFLAGS = -O3 -march=native -mtune=native
```

Override values from the command line when calling `make`:

```
> make CC=clang CFLAGS=-O0
```

Priorities for defining and overriding variables

The following ranking shows the priorities of variable definitions in the scope of `make` (first entry overrides later entries):

1. Variables passed via the command line
2. Self-defined variables inside the Makefile
3. Environment variables (using `export <var>=<value>`)
4. Pre-defined variables (`make` internal defaults)

How to interchange ranks 2 and 3:

```
> make -e|--environment-overrides
```

Important

- Pre-defined variables can (and should!) *always* be overridden

Autotools - Motivation

- Software packages oftentimes need to be buildable on different platforms
  $\Rightarrow$ Makefile needs to be adjusted
  $\Rightarrow$ Cumbersome and inefficient if done manually!
- Solution: Use a script to adjust Makefile to current needs
- GNU project: `configure` script
  - Standardized process of customizing the build options
  - Simplest scenario: `./configure && make && make install`
- Autotools:
  - Create the whole build system for current software package
  - Autoconf: `configure` script
  - Automake: Makefile
- For detailed introduction see: `https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html`

Working with GNU configure Scripts

The `configure` script

A GNU `configure` script aims to ensure a working compilation on a specific target system with a specific software environment, without the user having to manually edit the Makefile.

- Most `configure` scripts offer lots of options for customization of the build process (e.g. which location to install into, etc.)
- There are some "quasi-standards" (see next slide) w.r.t. parameter names and available options that are adhered to by **most** developers
- `configure --help` displays an exhaustive list of possible parameters and environment variables with explanations on how to use them

GNU configure Scripts - Common Parameters

When using a GNU `configure` script that honors a few common standards, the following is expected to work:

- `configure --prefix=<install_location>` sets the install location (i.e. where compiled files get copied to when typing `make install`)

> Make sure that you have write permissions in the prefix path!

- `--enable-<feature>` and `--disable-<feature>` include or exclude optional components in the compilation
- `--with-<library>[=<library_path>]` and `--without-<library>` include and set paths to external dependencies, or try to build without them if possible
- Common GNU Makefile environment (e.g. `CC`, `CFLAGS`, `LDFLAGS`) are taken into account by `configure`

GNU configure Scripts - Typical Workflow

It is recommended to build **outside of the source tree**, i.e. creating a new empty directory to contain files produced by the build process (e.g. object files).

1. Create a build directory and change into it, e.g.:
   `mkdir $HOME/build/<some_software> && cd $HOME/build/<some_software>`
2. Check configure options: `<path_to_source>/configure --help`
3. Execute configure with appropriate options: `<path_to_source>/configure <parameters>`
4. Build: `make`
5. If available, build and execute self-checks: `make check` or `make test`
6. Install into the directory set by `--prefix`: `make install`

Building a program from a source tarball: Gnuplot

The source tarball for the open source interactive plotting software Gnuplot can be found on https://sourceforge.net/projects/gnuplot/files/gnuplot/5.2.6/

Demos/configure–Gnuplot/build.sh

```
# SPDX-FileCopyrightText: 2021 Competence Center for High Performance Computing in He
# SPDX-License-Identifier: MIT

#!/bin/bash

tar xvf gnuplot-5.2.6.tar.gz # unpack and untar the sources
cd gnuplot-5.2.6
# module purge && module load gcc # default gcc module (on a cluster)
mkdir build && cd build # create and change into the build directory
mkdir -vp ../install/ # create an install directory
# ../configure --help # check the options for configure script
../configure CC=gcc CXX=g++ CPP='gcc -E' CXXCPP='g++ -E' \
    --prefix=$(pwd)/../install --without-latex
# less config.log # inspect configure output
make -j4 # compile
make install # install
```

Environment Variables

- The Modulesystem works with Environment Variables
- Compilers will automatically use this variables to determine where to search for files
- Some Important Environment Variables are:
  - `$PATH`: Location of Executables
  - `$CPATH` Location of header files
  - `$LIBRARY_PATH` Location of Libraries (compile time)
  - `$LD_LIBRARY_PATH` Location of Libraries (run time)
- Each Variable is a colon separated list of directories
- Append it by using `export PATH=$PATH:/your/new/path`

Module Files

- You can even write your own modules
- name the directory after the module and the file itself after the version number
- Use them by appending the `$MODULEPATH` environment variable

libxml2/2.9.7.lua

```lua
whatis("library for xml parsing")
whatis("http://www.xmlsoft.org/")
whatis("Full module name: libxml2/2.9.7")
whatis("Module File: /home/tj75qeje/modules/modulefiles/libxml2/2.9.7.lua")
whatis("Application Root: /home/tj75qeje/modules/software/libxml2/2.9.7")
-- Environment
prepend_path("PATH","/home/tj75qeje/modules/software/libxml2/2.9.7/bin")
prepend_path("CPATH","/home/tj75qeje/modules/software/libxml2/2.9.7/include")
prepend_path("LIBRARY_PATH","/home/tj75qeje/modules/software/libxml2/2.9.7/lib")
prepend_path("LD_LIBRARY_PATH","/home/tj75qeje/modules/software/libxml2/2.9.7/lib")
pushenv("LIBXML2_ROOT","/home/tj75qeje/modules/software/libxml2/2.9.7")

-- Messages
LmodMessage("Lmod: ", mode().."ing", myModuleName(), myModuleVersion())
```

Exercise

- Now try it yourself!
- build the NetCDF library from source
  - `https://downloads.unidata.ucar.edu/netcdf/`
  - The HDF5 libraryis needed as a prerequisite.
    Therefore you have to load this module!
- build an example program and link it against your NetCDF library
  - `https://www.unidata.ucar.edu/software/netcdf/examples/programs/simple_xy_wr.c`
- Can you build the NetCDF library with support for Parallel IO?
- Can you also write a modulefile for your own build of the NetCDF library?

- NetCDF can also be built with cmake (not explained in this course)

References I

Automatic Variables. URL: https://www.gnu.org/software/make/manual/make.html#Automatic-Variables (visited on 03/07/2019) (cit. on p. 27).

Directory-Search. URL: https://www.gnu.org/software/make/manual/make.html#Directory-Search (visited on 02/14/2019) (cit. on pp. 28, 29).

Makefile Conventions. URL: https://www.gnu.org/software/make/manual/make.html#Makefile-Conventions (visited on 03/12/2019) (cit. on p. 31).

Phony Targets. URL: https://www.gnu.org/software/make/manual/make.html#Phony-Targets (visited on 02/14/2019) (cit. on p. 30).

References II

Writing Rules. URL: https://www.gnu.org/software/make/manual/make.html#Rules (visited on 02/14/2019) (cit. on p. 22).