HPS

trevor-khwam.tabougua@gwdg.de

Trevor Khwam Tabougua

Compilation of applications via, make, CMake, and Autotools

Table of contents

1 Learning objectives

2 Definition

3 Makefile

4 CMake

5 Autotools

Learning objectives

- Get acquainted with software compilation
- Identify a makefile
- Build a simple library with make
- Generate executables with CMake and Autotools

Why compiling?

- Compiling means to create an executable – or a library – from the source code
- GWDG cannot install all software required by users
  (see modules for what is available)
- Scientific software is often only available as source code
- Compiling on the target system often yields better performance
- Prepackaged software typically requires administrator (root) privileges ...

Example

Small programs → single file → manual compilation

`gcc -o hello main.c`

Problems:

- **Difficulty in Maintaining Dependencies:** It can be tough to ensure that all dependencies are correctly compiled and updated.
- **Scaling Issues:** As the codebase grows, manual compilation becomes increasingly inefficient.
- **Reproducibility:** Without a structured build process, it may be difficult to maintain consistency across different build environments and ensure that the code will compile and run as expected.

Solution: **Makefile**

Makefile

It contains a set of rules that check:

- If certain preconditions are met
  (files exist or have been updated).
- Runs specific commands as needed if the dependencies changed.

Format

```
<target>: {<sources>}
    <commands>
```

⚠ Note that the white-space is a tabulator.

Example:

```
hello: main.c
    gcc main.c -o hello
```

A **clean** rule can also be added

```
hello: main.c
    gcc main.c -o hello
clean:
    rm -f hello
```

Execution

In the terminal:

- The command **`make`** will create the file `hello`
- If there's a cleaning rule, the command `make clean` will remove all the output objects

Build more than one executable

An **all** rule that depends on all other objects is used.

```
1 all: <target1> <target2>
2 <target1>: {<sources1>}
3     <commands1>
4 <target2>: {<sources2>}
5     <commands2>
```

Exercise 1

- Clone the git repository `https://github.com/KTTrev/Exercises.git`
- cd into Exercises/test_make, identify the makefile, and execute it
- Check if the compilation worked properly by running the obtained executables, and clean them afterwards

Time: 10min

Challenges

Libraries often come with several challenges, including:

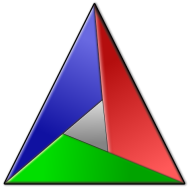
- **Complex Programs:** Libraries tend to have intricate structures, making it difficult to manage multiple files and functions.
- **Makefile Management:** Writing and maintaining Makefiles manually for large projects is error-prone and time-consuming.
- **Platform Variability:** Libraries may behave differently across operating systems, requiring conditional Makefile configurations.
- **Updating Dependencies:** Keeping dependencies up to date without breaking compatibility with older versions can be challenging.

Solution: Makefile Generators

CMake

CMake is a cross-platform, open-source tool for managing the build process in a compiler-independent way.

It generates Makefiles using a configuration file called `CMakeLists.txt`, placed in the project directory.



Generating a Makefile with CMake

Steps: Assuming you're in the project folder with `CMakeLists.txt`:

1. `mkdir build`
2. `cd build`
3. `cmake ..`

Now, the Makefile is generated. Use `make` to build the executables.

Exercise 2

In the previous git repository, if you cd into Exercises/test_cmake, you will find the three C scripts that were used in the previous exercise, with an additional file: CMakeFiles.txt

- Load cmake/3.27.7 with the following command:
  `module load cmake/3.27.7`
- Use the steps provided in the previous slide to generate the executables

Time: 5 min

Autotools

Like CMake, **Autotools** helps manage and package source code, making it easier for users to compile and install software.

Primary input files:

- `configure.ac` (**Autoconf**): Defines configuration scripts for system detection and build configuration.
- `Makefile.am` (**Automake**): Specifies how the software should be built and installed, generating the ‘Makefile‘.

Using Autotools

The basic steps to build Autotools-based software are:

1. **Configuration**
   `./configure --prefix=DIR`
   Checks the build environment, verifies dependencies, and generates Makefile(s) and a `config.h`.
2. **Compilation**
   `make`
   Builds the software using the generated Makefiles.
3. **Installation**
   `make install`
   Installs the built components (moves binaries to the specified PATH).

About "--prefix"

- The "--prefix" option specifies the base installation directory for your software.
- Use `./configure --prefix=DIR` to install directly in DIR.
- **Example:**
  - `./configure --prefix=$HOME/software/<name-version>`

Example with Nano

- Download the source code using `wget`:
  - `wget https://www.nano-editor.org/dist/v7/nano-7.2.tar.gz`
- Unpack the source code with `tar`:
  - `tar -xvf nano-7.2.tar.xz`
- Build and install Nano:
  - `cd nano-7.2/`
  - `mkdir build`
  - `../configure --prefix=/usr/users/YOUR_USERNAME/bin/`
  - `make`
  - `make install`

Exercise 3

Task: Download and install the source code for `ncurses-6.3`.

- Visit: `https://ftp.gnu.org/gnu/ncurses/`
- Follow the installation steps from the previous example.

Estimated Time: 10 minutes

References and Further Reading

- **Make**
  - Official Manual
  - Make Tutorial
- **CMake**
  - Official Manual
- **Autotools**
  - Official Manual