CMake introduction

Alessandro Di Federico

ale@clearmind.me

Algorithms and Parallel Computing
Mathematical Engineering
Politecnico di Milano

November 6, 2015

What's a build system?

- Saves you from typing:

```
g++ hello.cpp -o hello
```

- A build system
  - manages multiple source files
  - find and manages linked libraries
  - selectively compiles what's changed
  - can be configured in different ways

CMake

- A modern, simple and portable build system
- An alternative to plain Makefiles and autoconf
- It generates directives on how to compile the project:
  - Makefiles (scripts for compiling from the terminal)
  - Code::Blocks projects
  - Visual Studio projects
  - ...

Out-of-tree builds

- Source and compiled files used to be in the same directory
- e.g. `hello.c` and `hello.o`
- Separating them has some advantages:
  - You can easily discard the build directory
  - You can have multiple builds with different configurations
  - The build directory can be on another storage device
- CMake makes it very easy to have out-of-tree builds

My first CMakeLists.txt

- CMake is configured through a `CMakeLists.txt` file
- Here's a minimal one:

```
cmake_minimum_required(VERSION 2.7)
project(hello CXX)
add_executable(hello hello.cpp)
```

Configure the build

```
# From the source/ directory
$ cd ..
$ mkdir build
$ cd build
$ cmake ../source/
-- The CXX compiler identification is GNU 4.8.5
-- Check for working CXX compiler: g++
-- Check for working CXX compiler: g++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: [...]/build
$
```

What did CMake do?

- It autodetects the C++ compiler and its features
- It caches the configuration in `CMakeCache.txt`
- It generates the `Makefile` (i.e. the build instructions)

Launch the build

```
# From the build/ directory
$ make
Scanning dependencies of target hello
[ 50%] Building CXX object [...]/hello.cpp.o
[100%] Linking CXX executable hello
[100%] Built target hello
$
```

What happened?

- `make` reads the instructions in `Makefile`
- It builds each source file
- It links them together in the final executable (`hello`)
- At this point we can launch it

```
$ ./hello
Hello world!
$
```

Clean the build directory

- If we launch make again it will do nothing
- That's good, nothing changed
- We can force recompilation with `make clean`

```
$ make
[100%] Built target hello
$ make clean
$ make
[ 50%] Building CXX object [...]/hello.cpp.o
[100%] Linking CXX executable hello
[100%] Built target hello
$
```

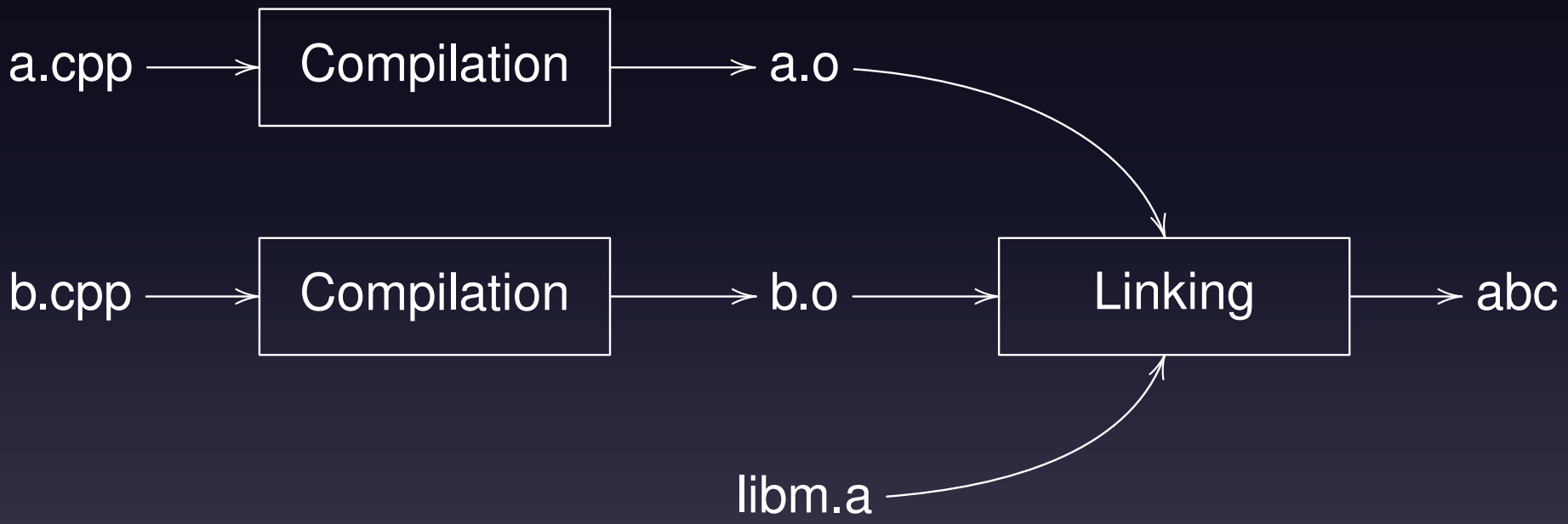
Compiler flags

- Suppose we want to pass some flags to the compiler
- CMake has a set of predefined variables[1]
- `CMAKE_CXX_FLAGS` contains the flags for the C++ compiler

```
set(CMAKE_CXX_FLAGS
    "${CMAKE_CXX_FLAGS} -fopenmp")
```

[1] Here are the most useful ones:
http://www.cmake.org/Wiki/CMake_Useful_Variables

The building process



Linking other libraries

- We saw `add_executable`
- What if we want to use a library?
- We can use `target_link_libraries`

```
target_link_libraries(hello m)
```

Build types

- CMake supports different build configurations:
  - Debug (with debug information, no optimizations)
  - Release (no debug information, aggressive optimizations)
- You can configure them through the `CMAKE_BUILD_TYPE`:
  - Either in CMakeLists:
    ```
    set(CMAKE_BUILD_TYPE Debug)
    ```
  - Or upon CMake invocation:
    ```
    cmake ../source/ -DCMAKE_BUILD_TYPE=Debug
    ```

Custom *cached* variables

- We can also define custom cached variables with `set`[2]

```
set(ENABLE_OMP On CACHE BOOL "Enable OpenMP")

if(ENABLE_OMP)
  set(CMAKE_CXX_FLAGS
      "${CMAKE_CXX_FLAGS} -fopenmp")
endif()
```

- By default OpenMP will be enabled
- To disable it:
  `cmake ../source/ -DENABLE_OMP=Off`

[2] Checkout the set doc:
https://cmake.org/cmake/help/v3.0/command/set.html

Generate Code::Blocks project

- CMake can generate various types of build instructions
- To generate a Code::Blocks project:
  `cmake ../source/ -G "CodeBlocks - Unix Makefiles"`
- It will generate a `hello.cbp` file
- You can just open it with Code::Blocks

CMake packages

- Some projects provide a package to be used with CMake
- For instance MPI, which we are going to use
- A package it's just a CMake file which you can import

```
find_package(MPI REQUIRED)
set(CMAKE_CXX_COMPILE_FLAGS
    ${CMAKE_CXX_COMPILE_FLAGS}
    ${MPI_COMPILE_FLAGS})
set(CMAKE_CXX_LINK_FLAGS
    ${CMAKE_CXX_LINK_FLAGS}
    ${MPI_LINK_FLAGS})
include_directories(MPI_INCLUDE_PATH)
```

CTest

- CTest[3] is a CMake “extension” for testing
- CMake easily integrates with CTest
- You can define a test command to launch
- Depending on the exit result the test fails or succeeds
- CTest keeps track of execution timing and results

[3] https://cmake.org/Wiki/CMake/Testing_With_CTest

CTest example

In CMakeLists.txt:

```
enable_testing()
add_test(NAME check_hello COMMAND ./hello)
```

To run the tests:

```
$ make test
Running tests...
Test project [...]/build
    Start 1: check_hello
1/1 Test #1: check_hello ...   Passed    0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.01 sec
$
```

Further references

- The CMake doc, very well done: https://cmake.org/cmake/help/v3.0/genindex.html
- The CMake tutorial: https://cmake.org/cmake-tutorial/
- The CMake Wiki: https://cmake.org/Wiki/Main_Page

License