TRAINING-CMAKE-BASICS

Speaker: Nicolò Genesio(@Nicogene)

OUTLINE

1. Compile an executable from sources
2. Build without CMake
3. About CMake
4. CMake the Basics
5. Hands-on

COMPILE AN EXECUTABLE FROM SOURCES

Compile an executable from sources means take some human readable code(c, c++, c#, java) and give it to the compiler.

The compiler is a program that translate it in instructions for the processor(way less human readable).

Focusing on c++, what you need is a c++ compiler, and a .cpp file containing your `main()` function.

COMPILE AN EXECUTABLE FROM SOURCES

Your program can work standalone or it may need some, let’s say, external help.

This help is provided by the libraries that define the external dependencies. This allow to reuse some logic already implemented by someone else, and hopefully already tested by thousand of people in order to “not-re-invent-the-wheel”.

Without going in details, the libraries has an interface (headers) that tell to the user how to use them, and a runtime component, the actual processor instructions implemented by the library.

An executable has to include headers and link the library runtime component.

BUILD WITHOUT CMAKE

There are several ways to compile an executable without cmake:

1. Command line:

```
$ g++ -Wall -o my_exe –I /my/include/dir/ –l /my/library/location/lib.so main.cpp
```

(or arlternatively embed this to a bash script)

- Drawbacks: non-portability! it works only on a Unix systems, with gcc, and the include and library have to be in those specific location. In few words: “It works on my machine”.

BUILD WITHOUT CMAKE

2. Write your Makefile:

The Makefiles are special files in which you can write in a more (but still complicated) readable way the command we saw before. This file will be used by the make command, that translate it in the instructions for compiling our program.

- Drawbacks: non-portability, again! It contains hardcoded path for the libraries and the include files, and it doesn’t take count of the differences between the OSes.

ABOUT CMAKE

CMake is a build-system, since it consists on a system that allow to build you executable or library from sources.

It is cross-platform, it supports all of the most popular OSes and compilers.

Its entry-point is a text file called CMakeLists.txt that it is written in its own scripting language.

This file is used by the command `cmake` to generate the makefiles abstracting over the OS, compiler, and libraries installation forlders.

ABOUT CMAKE

But “not all that glitters is gold” :

- Uses its own language, reinventing the wheel.
- It does not follow any well known standard or guidelines.
- Documentation is not so good and we often rely on non official webpages.
- NO target uninstall

“CMake is **** but it is the best build system we have” [cit @drdanz]

CMAKE THE BASICS

The cmake-language is a 'scripting language':

1. You can define variables: `set(var 10)`
2. Print on the terminal that var: `message("This is my var: ${var}")`
3. It has control statements: `foreach, if, while, function`
4. It has some data structures: `list, string`
5. It supports modularity trough CMakeModules files.
6. System inspection: `find library , find program, find package`, etc

CMAKE THE BASICS

It is possible to pass arguments(called options) user-defined or cmake-native to the `cmake` command through the gui (`ccmake`) or through the command line:

```
$ cmake -D<my_option>=ON <path-where-is-the-CMakeLists.txt>
```

An example of cmake-native option is CMAKE_INSTALL_PREFIX, that allows to define where your project(library or executable) will be installed on your PC.

CMAKE THE BASICS

CMake it is a scripting language that allows you to do several things but let's focus on how to use it in order to create an executable.

Here a list of basic cmake directive:

- **`cmake_minimum_required`**: sets the minimum required version of cmake for a project.
- **`project`**: sets the name of the project, and stores it in the variable PROJECT_NAME. When called from the top-level CMakeLists.txt also stores the project name in the variable CMAKE_PROJECT_NAME.
- **`add_subdirectory`**: adds a subdirectory to the build. This subdirectory must contain a CMakeLists.txt.
- **`add_library`**: defines a library with a specific name, it may be STATIC or SHARED.

CMAKE THE BASICS

- **`add_executable`**: defines an executable with a specific name.
- **`target_include_directories`**: this is equivalent to "-I…", this adds a directory to the list of preprocessor include file search directories for a specific target, that it has to be previously defined by a `add_executable()` and `add_library()` command.
- **`target_link_libraries`**: specifies libraries to use when linking a given target and/or its dependents.
- **`target_compile_definitions`**: specifies compile definitions to use when compiling a given target that must have been created by a command such as `add_executable()` or `add_library()`

HANDS-ON

Now we should be prepared in order to start the hands-on.

Please follow the instructions that you can find here:

https://github.com/icub-tech-iit/training-cmake-basics