



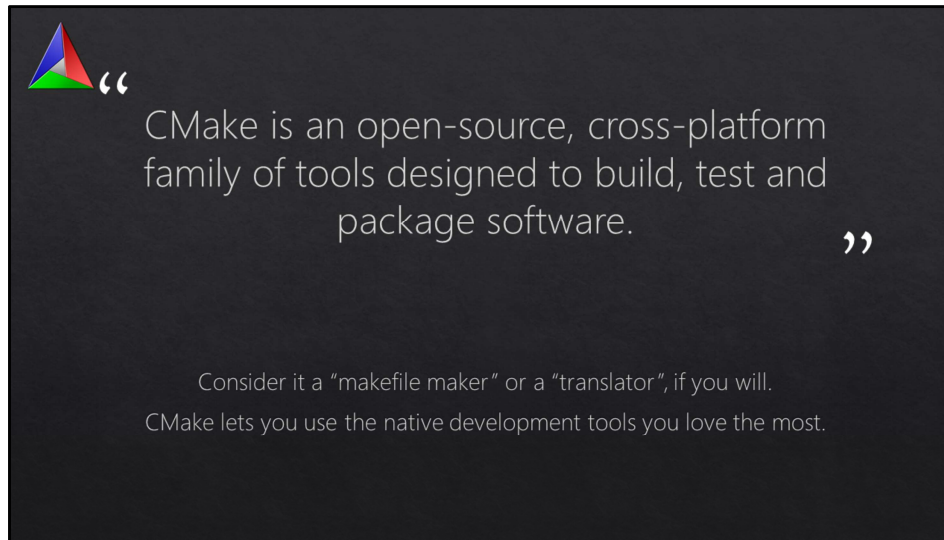
CMake

An introduction

Karstein Brynestad (许云深) 2016400428
Department of Industrial Engineering, Tsinghua University
Industrial Economics and Technology Management, NTNU



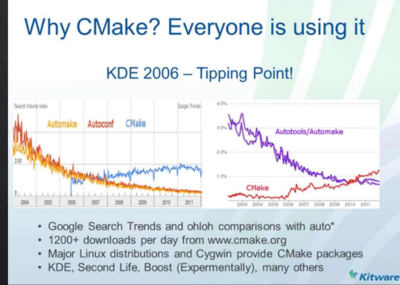
Now from its name, CMake obviously sounds like a tool you can use for “making C-based programs”. And that’s not too far from the truth. However, there is one significant difference between CMake and your regular compiler as the GCC we use for this course.



The key word in this definition is “cross-platform”. CMake is designed to allow for simple project or makefile generation from source code so that you and your colleague/study buddy don’t have to compromise on which compiler or tool you use. I will go into detail on the generation process later.



Initiated early 2000



Why CMake? Everyone is using it

KDE 2006 – Tipping Point!

Search Trends Index: Autotools Autocord CMake

Usage Trends: Autotools/Automake CMake

- Google Search Trends and ohloh comparisons with auto*
- 1200+ downloads per day from www.cmake.org
- Major Linux distributions and Cygwin provide CMake packages
- KDE, Second Life, Boost (Experimentally), many others

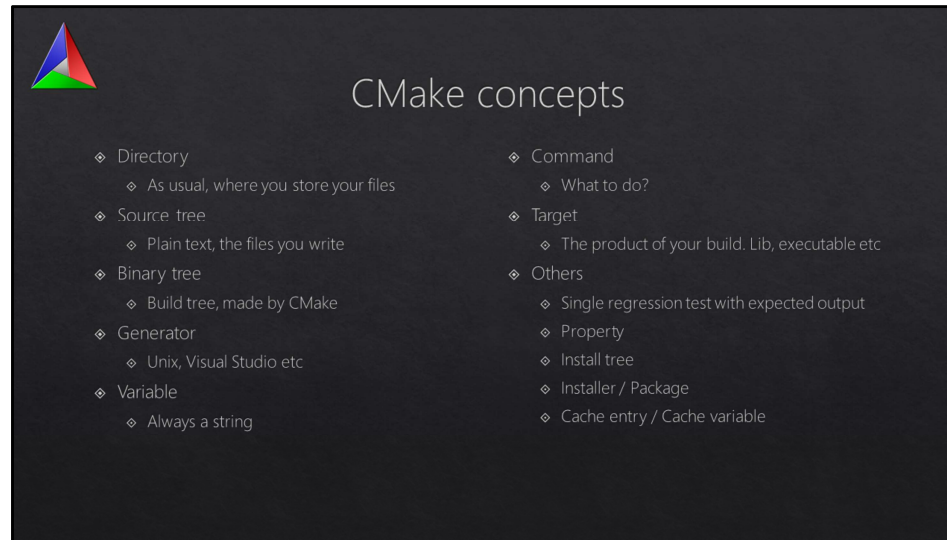
Source: "Introduction to CMake" video, <https://vimeo.com/32212195>

Kitware

CMake is influenced by an early system called pcmaker, created by Ken Martin. It was the guy in the picture, Bill Hoffman of Kitware, who transformed this into the CMake language late 2000 – early 2001. By 2010 it became the most used makefile/project generator for C. Bill Hoffman and his team are probably quite happy about the evolution of what initially was made to ease the development of a larger project in medical technology.



So essentially, with Cmake you start off with your source files as usual. However, instead of writing a makefile for your compiler you write a makefile for CMake. Depending on the complexity of your program and your level of programming this CMake might be easier to write than the original makefile as CMake is quite high-level. CMake even has a GUI you can use to look into the CMake process. However, when you run your CMake command it does not build your program. What it does is make a makefile or project file for the compiler/project tool of your choice, in the location where you want it. This is properly linked to your source file directory. This makefile then has to be run by your compiler / project software. CMake does not validate your program – this is still done by the compiler. Essentially, you will not need to run CMake more than once per project. CMake should by itself identify when it needs to be re-run and do this by itself.



Some things to keep in mind:

The source tree is where you keep your code. This can be organized in the way you like thanks to CMake doing the proper linkage for you in the makefile it generates.

The Binary tree, or Build tree, is where the generated files will be stored. Usually (especially for project files like Visual Studio) it is best to keep your source tree and build tree separate to avoid a total mess in your files.

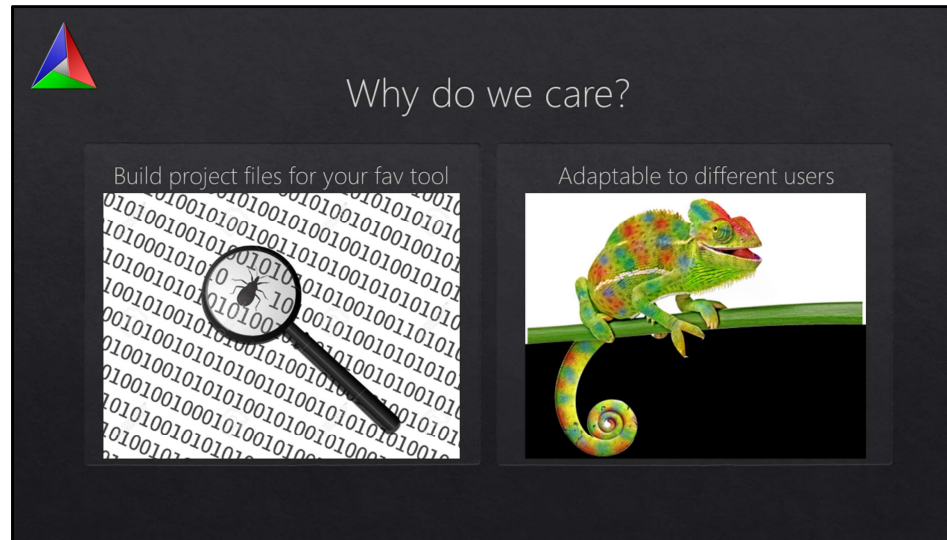
The generator is chosen based on what you tell CMake when running it. Requires you to have the compiler / project software installed.

There are variables in CMake, which always are treated as strings by the program. They can actually be passed on to header files etc!

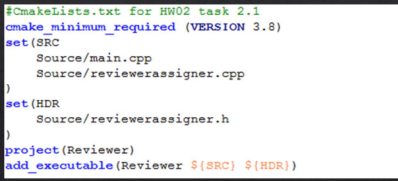
Command: In your CMake file, what you want the generator to do. E.g. `add_executable`

Target: What to make. Library, executable and the like.

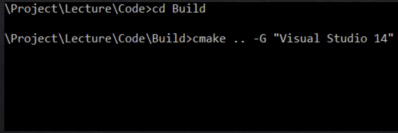
Some others that are quite complicated. Look into it if you want to. Actually you can also use CMake to create installer packages when your project is to be shipped.



Personally, I consider the Visual Studio IDE graphical debugger to be quite useful. However, in this course it has been too complicated to implement right away as we use CodeBlocks and GCC. With CMake, you can easily generate a Visual Studio project file from your code and debug it in VS. Also, when working with other people that prefer different IDEs / compilers, CMake can allow for easy interoperability.

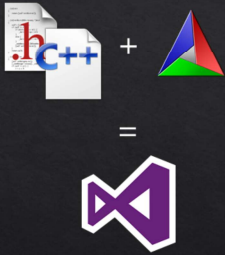


```
#CmakeLists.txt for HW02 task 2.1
cmake_minimum_required (VERSION 3.8)
set(SRC
    Source/main.cpp
    Source/reviewerassigner.cpp
)
set(HDR
    Source/reviewerassigner.h
)
project(Reviewer)
add_executable(Reviewer ${SRC} ${HDR})
```



```
\Project\Lecture\Code>cd Build
\Project\Lecture\Code\Build>cmake .. -G "Visual Studio 14"
```

A simple example



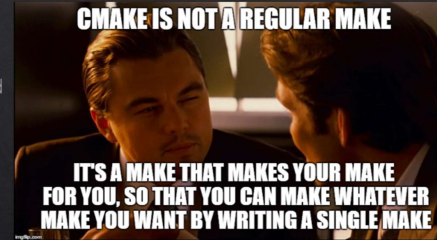
The diagram illustrates the process of building a C++ project. It shows a C++ source file (.h and .cpp) and a CMakeLists.txt file (represented by a triangle icon) being combined (indicated by a plus sign) to create a Visual Studio solution (indicated by an equals sign and the Visual Studio logo).

Codes can be found in the “Code” folder. README explains how to make it work.



What to remember

- ❖ CMake doesn't "make" or "build" the program for you – it generates the build or makefile of your choice
- ❖ Separate your source tree from your build tree
- ❖ Behaves differently for different generators
- ❖ Make sure the tool/compiler you are generating for is already installed!



Sources:

"CMake overview": <https://cmake.org/overview/>

"CMake tutorial": <https://cmake.org/cmake-tutorial/>

"Introduction to CMake" video: <https://vimeo.com/32212195> (VPN Required)

At our level of coding, CMake might seem like adding an extra layer that doesn't provide any benefit. However, as projects get big and complicated this could be a tool that ensures that programmers can use their favorite tools for max efficiency. Takes some time to be fully proficient in CMake though and some weird bugs might be made in generation so it's of course a trade-off. However, as some of the projects in our code reading implements CMake you can tell that it can be nice to have in some instances. If you want to learn more about it, check out the links provided.