CGold Documentation

Release 0.1

Ruslan Baratov

Feb 17, 2019

Contents

Warning: THIS DOCUMENT IS UNDER CONSTRUCTION! WORK IN PROGRESS!

Welcome to **CGold**!

This guide will show you how to use *CMake* and will help you to write elegant, correct and scalable projects. We'll start from the simple cases and add more features one by one. This tutorial covers only part of the CMake capabilities - some topics are skipped intentionally in favor of better modern approaches [1]. Document designed to be a good tutorial for the very beginners but touches some aspects in which advanced developers may be interested too. Look at this document as a skeleton/starting point for further CMake learning.

Enjoy!

[1] See *rejected* section for list with detailed description

CHAPTER 1

Overview

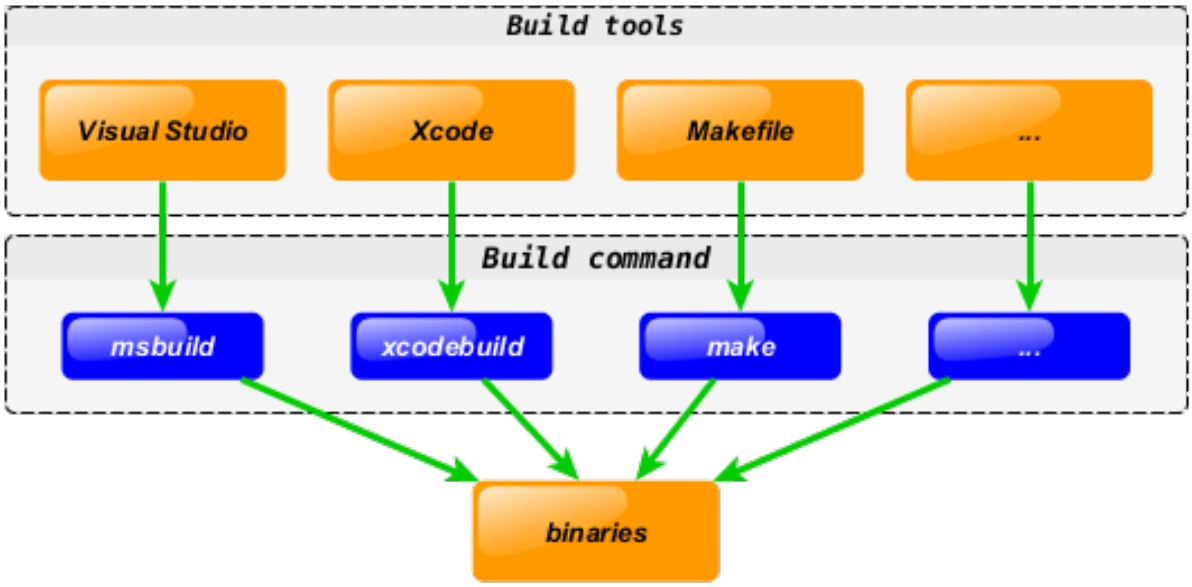
1.1 What CMake can do

CMake is a meta build system. It can generate real *native build tool* files from abstracted text configuration. Usually such code lives in `CMakeLists.txt` files.

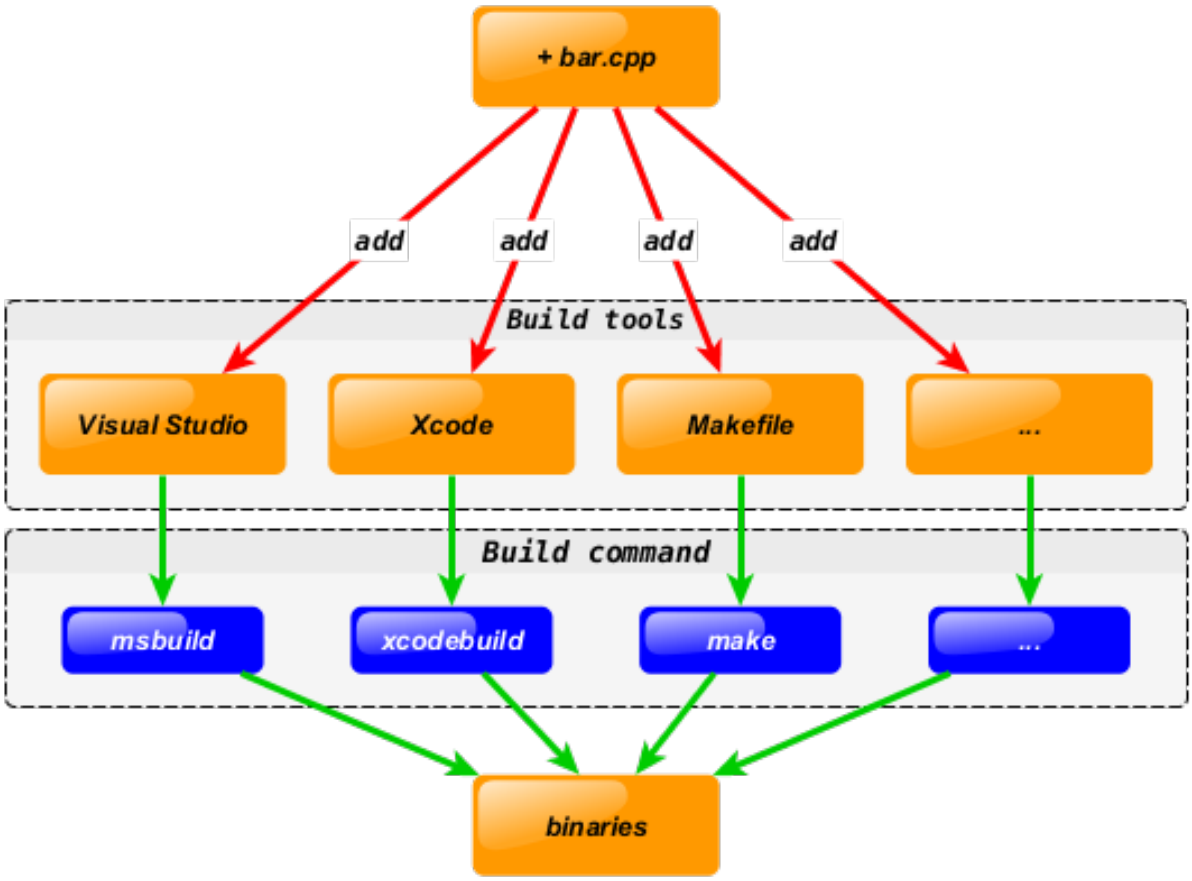
What does it mean and how it can be useful?

1.1.1 Cross-platform development

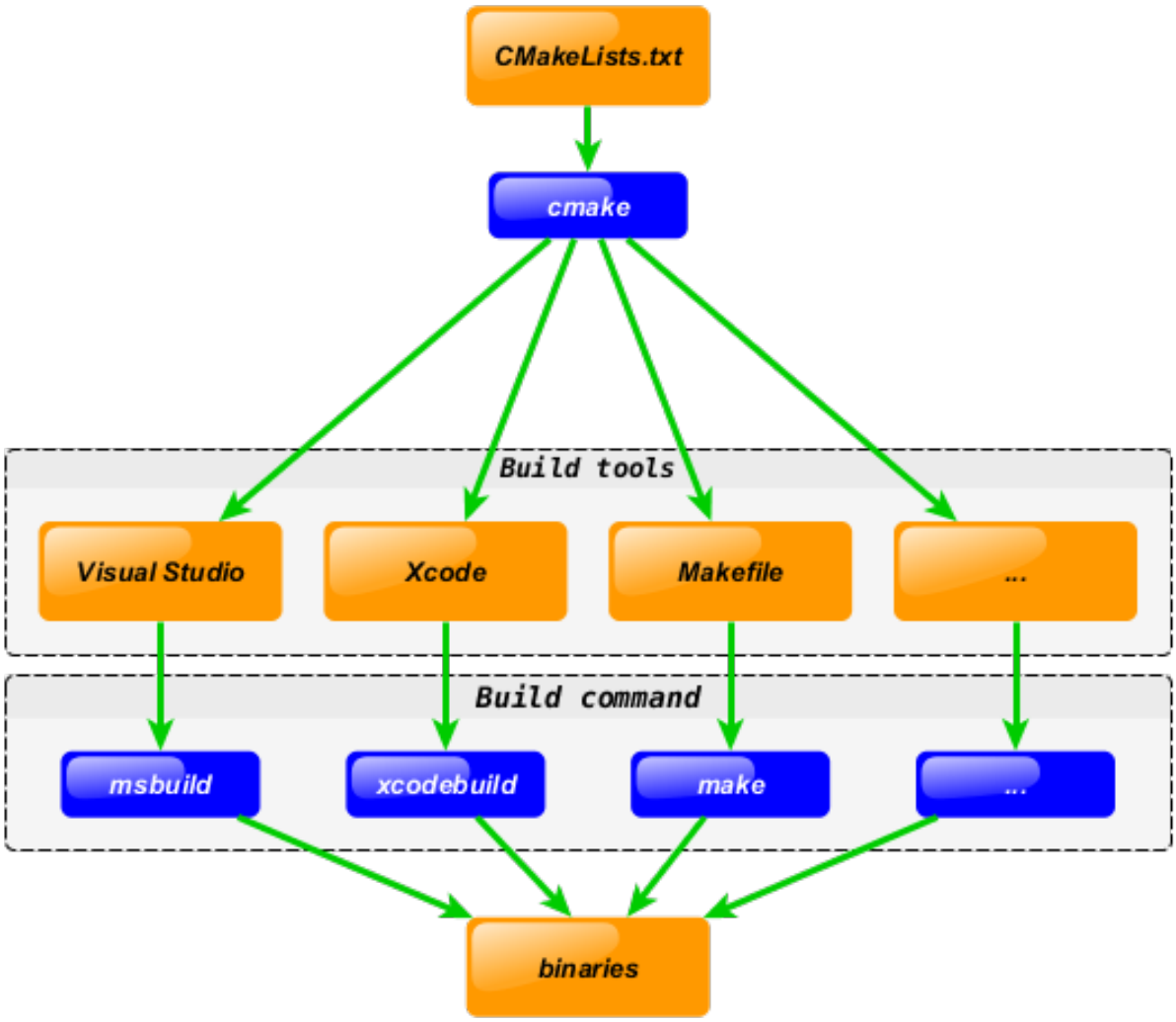
Let's assume you have some cross-platform project with C++ code shared along different platforms/IDEs. Say you use `Visual Studio` on Windows, `Xcode` on OSX and `Makefile` for Linux:



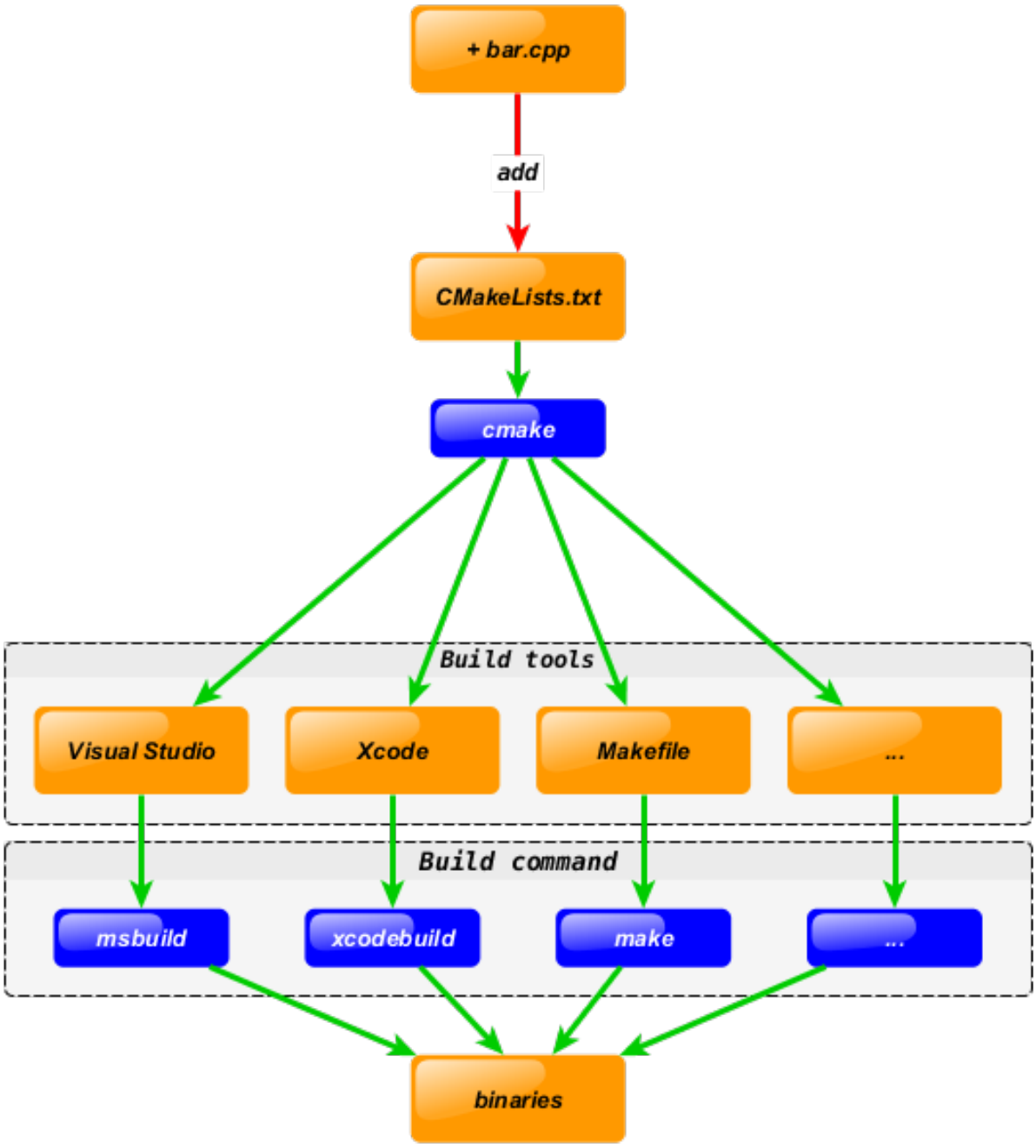
What you will do if you want to add new `bar.cpp` source file? You have to add it to every tool you use:



To keep the environment consistent you have to do the similar update several times. And the most important thing is that you have to do it **manually** (arrow marked with a red color on the diagram in this case). Of course such approach is error prone and not flexible. CMake solve this design flaw by adding extra step to development process. You can describe your project in `CMakeLists.txt` file and use *CMake* to generate tools you currently interested in using cross-platform *CMake* code:



Same action - adding new `bar.cpp` file, will be done in **one step** now:



Note that the bottom part of the diagram **was not changed**. I.e. you still can keep using your favorite tools like `Visual Studio/msbuild`, `Xcode/xcodebuild` and `Makefile/make`!

See also:

- KDE moving from autotools to CMake
- Visual C++ Team Blog: Support for Android CMake projects in Visual Studio
- Android Studio: Add C and C++ code to Your Project

1.1.2 VCS friendly

When you work in team on your code you probably want to share and save the history of changes, that's what usually *VCS* used for. How does storing of IDE files like `*.sln` works on practice? Here is the diff after adding `bar` executable with `bar.cpp` source file to the `Visual Studio` solution:

```
--- /overview/snippets/foo-old.sln
+++ /overview/snippets/foo-new.sln
@@ -4,6 +4,8 @@
 VisualStudioVersion = 14.0.25123.0
 MinimumVisualStudioVersion = 10.0.40219.1
 Project("{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}") = "foo", "foo.vcxproj", "{C8F8C325-
 ↪ACF3-460E-81DF-8515C72B334A}"
+EndProject
+Project("{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}") = "bar", "..\bar\bar.vcxproj", "
 ↪{D14B78EA-1ADA-487F-B1ED-42C2B919C000}"
 EndProject
 Global
         GlobalSection(SolutionConfigurationPlatforms) = preSolution
@@ -21,6 +23,14 @@
                 {C8F8C325-ACF3-460E-81DF-8515C72B334A}.Release|x64.Build.0 =␣
 ↪Release|x64
                 {C8F8C325-ACF3-460E-81DF-8515C72B334A}.Release|x86.ActiveCfg =␣
 ↪Release|Win32
                 {C8F8C325-ACF3-460E-81DF-8515C72B334A}.Release|x86.Build.0 =␣
 ↪Release|Win32
+                {D14B78EA-1ADA-487F-B1ED-42C2B919C000}.Debug|x64.ActiveCfg =␣
 ↪Debug|x64
+                {D14B78EA-1ADA-487F-B1ED-42C2B919C000}.Debug|x64.Build.0 = Debug|x64
+                {D14B78EA-1ADA-487F-B1ED-42C2B919C000}.Debug|x86.ActiveCfg =␣
 ↪Debug|Win32
+                {D14B78EA-1ADA-487F-B1ED-42C2B919C000}.Debug|x86.Build.0 =␣
 ↪Debug|Win32
+                {D14B78EA-1ADA-487F-B1ED-42C2B919C000}.Release|x64.ActiveCfg =␣
 ↪Release|x64
+                {D14B78EA-1ADA-487F-B1ED-42C2B919C000}.Release|x64.Build.0 =␣
 ↪Release|x64
+                {D14B78EA-1ADA-487F-B1ED-42C2B919C000}.Release|x86.ActiveCfg =␣
 ↪Release|Win32
+                {D14B78EA-1ADA-487F-B1ED-42C2B919C000}.Release|x86.Build.0 =␣
 ↪Release|Win32
         EndGlobalSection
         GlobalSection(SolutionProperties) = preSolution
                 HideSolutionNode = FALSE
```

And new `bar.vcxproj` of 150 lines of code. Here are some parts of it:

```
<Project DefaultTargets="Build" ToolsVersion="14.0" xmlns="http://schemas.microsoft.
 ↪com/developer/msbuild/2003">
  <ItemGroup Label="ProjectConfigurations">
    <ProjectConfiguration Include="Debug|Win32">
      <Configuration>Debug</Configuration>
      <Platform>Win32</Platform>
    </ProjectConfiguration>
    <ProjectConfiguration Include="Release|Win32">
      <Configuration>Release</Configuration>
      <Platform>Win32</Platform>
    </ProjectConfiguration>
```

```
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|x64'" Label=
↪"Configuration">
    <ConfigurationType>Application</ConfigurationType>
    <UseDebugLibraries>true</UseDebugLibraries>
    <PlatformToolset>v140</PlatformToolset>
    <CharacterSet>Unicode</CharacterSet>
  </PropertyGroup>
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Release|x64'" Label=
↪"Configuration">
    <ConfigurationType>Application</ConfigurationType>
    <UseDebugLibraries>false</UseDebugLibraries>
    <PlatformToolset>v140</PlatformToolset>
    <WholeProgramOptimization>true</WholeProgramOptimization>
    <CharacterSet>Unicode</CharacterSet>
  </PropertyGroup>
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />
  <ImportGroup Label="ExtensionSettings">
  </ImportGroup>
  <ImportGroup Label="Shared">
  </ImportGroup>
  <ImportGroup Label="PropertySheets" Condition="'$(Configuration)|$(Platform)'==
↪'Debug|Win32'">
    <Import Project="$(UserRootDir)\Microsoft.Cpp.$(Platform).user.props" Condition=
↪"exists('$(UserRootDir)\Microsoft.Cpp.$(Platform).user.props')" Label=
↪"LocalAppDataPlatform" />
  </ImportGroup>
```

```
  <ItemDefinitionGroup Condition="'$(Configuration)|$(Platform)'=='Release|x64'">
    <ClCompile>
      <WarningLevel>Level3</WarningLevel>
      <PrecompiledHeader>
      </PrecompiledHeader>
      <Optimization>MaxSpeed</Optimization>
      <FunctionLevelLinking>true</FunctionLevelLinking>
      <IntrinsicFunctions>true</IntrinsicFunctions>
      <PreprocessorDefinitions>NDEBUG;_CONSOLE;%(PreprocessorDefinitions)</
↪PreprocessorDefinitions>
    </ClCompile>
    <Link>
      <SubSystem>Console</SubSystem>
      <EnableCOMDATFolding>true</EnableCOMDATFolding>
      <OptimizeReferences>true</OptimizeReferences>
      <GenerateDebugInformation>true</GenerateDebugInformation>
    </Link>
```

```
  <ItemGroup>
    <ClCompile Include="bar.cpp" />
  </ItemGroup>
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />
  <ImportGroup Label="ExtensionTargets">
  </ImportGroup>
```

When using `Xcode`:

```
--- /overview/snippets/project-old.pbxproj
+++ /overview/snippets/project-new.pbxproj
```

```
@@ -8,6 +8,7 @@

 /* Begin PBXBuildFile section */
                 0FE79B881D22BAE400E38C27 /* main.cpp in Sources */ = {isa =␣
↪PBXBuildFile; fileRef = 0FE79B871D22BAE400E38C27 /* main.cpp */; };
+                0FE79B951D22BB5E00E38C27 /* bar.cpp in Sources */ = {isa =␣
↪PBXBuildFile; fileRef = 0FE79B941D22BB5E00E38C27 /* bar.cpp */; };
 /* End PBXBuildFile section */

 /* Begin PBXCopyFilesBuildPhase section */
@@ -20,15 +21,33 @@
                         );
                         runOnlyForDeploymentPostprocessing = 1;
                 };
+                0FE79B901D22BB5E00E38C27 /* CopyFiles */ = {
+                        isa = PBXCopyFilesBuildPhase;
+                        buildActionMask = 2147483647;
+                        dstPath = /usr/share/man/man1/;
+                        dstSubfolderSpec = 0;
+                        files = (
+                        );
+                        runOnlyForDeploymentPostprocessing = 1;
+                };
 /* End PBXCopyFilesBuildPhase section */

 /* Begin PBXFileReference section */
                 0FE79B841D22BAE400E38C27 /* foo */ = {isa = PBXFileReference;␣
↪explicitFileType = "compiled.mach-o.executable"; includeInIndex = 0; path = foo;␣
↪sourceTree = BUILT_PRODUCTS_DIR; };
                 0FE79B871D22BAE400E38C27 /* main.cpp */ = {isa = PBXFileReference;␣
↪lastKnownFileType = sourcecode.cpp.cpp; path = main.cpp; sourceTree = "<group>"; };
+                0FE79B921D22BB5E00E38C27 /* bar */ = {isa = PBXFileReference;␣
↪explicitFileType = "compiled.mach-o.executable"; includeInIndex = 0; path = bar;␣
↪sourceTree = BUILT_PRODUCTS_DIR; };
+                0FE79B941D22BB5E00E38C27 /* bar.cpp */ = {isa = PBXFileReference;␣
↪lastKnownFileType = sourcecode.cpp.cpp; path = bar.cpp; sourceTree = "<group>"; };
 /* End PBXFileReference section */

 /* Begin PBXFrameworksBuildPhase section */
                 0FE79B811D22BAE400E38C27 /* Frameworks */ = {
+                        isa = PBXFrameworksBuildPhase;
+                        buildActionMask = 2147483647;
+                        files = (
+                        );
+                        runOnlyForDeploymentPostprocessing = 0;
+                };
+                0FE79B8F1D22BB5E00E38C27 /* Frameworks */ = {
                         isa = PBXFrameworksBuildPhase;
                         buildActionMask = 2147483647;
                         files = (
@@ -42,6 +61,7 @@
                         isa = PBXGroup;
                         children = (
                                 0FE79B861D22BAE400E38C27 /* foo */,
+                                0FE79B931D22BB5E00E38C27 /* bar */,
                                 0FE79B851D22BAE400E38C27 /* Products */,
                         );
                         sourceTree = "<group>";
```

```
@@ -50,6 +70,7 @@
                        isa = PBXGroup;
                        children = (
                                0FE79B841D22BAE400E38C27 /* foo */,
+                               0FE79B921D22BB5E00E38C27 /* bar */,
                        );
                        name = Products;
                        sourceTree = "<group>";
@@ -60,6 +81,14 @@
                                0FE79B871D22BAE400E38C27 /* main.cpp */,
                        );
                        path = foo;
+                       sourceTree = "<group>";
+               };
+               0FE79B931D22BB5E00E38C27 /* bar */ = {
+                       isa = PBXGroup;
+                       children = (
+                               0FE79B941D22BB5E00E38C27 /* bar.cpp */,
+                       );
+                       path = bar;
                        sourceTree = "<group>";
                };
 /* End PBXGroup section */
@@ -80,6 +109,23 @@
                        name = foo;
                        productName = foo;
                        productReference = 0FE79B841D22BAE400E38C27 /* foo */;
+                       productType = "com.apple.product-type.tool";
+               };
+               0FE79B911D22BB5E00E38C27 /* bar */ = {
+                       isa = PBXNativeTarget;
+                       buildConfigurationList = 0FE79B981D22BB5E00E38C27 /* Build␣
↪configuration list for PBXNativeTarget "bar" */;
+                       buildPhases = (
+                               0FE79B8E1D22BB5E00E38C27 /* Sources */,
+                               0FE79B8F1D22BB5E00E38C27 /* Frameworks */,
+                               0FE79B901D22BB5E00E38C27 /* CopyFiles */,
+                       );
+                       buildRules = (
+                       );
+                       dependencies = (
+                       );
+                       name = bar;
+                       productName = bar;
+                       productReference = 0FE79B921D22BB5E00E38C27 /* bar */;
                        productType = "com.apple.product-type.tool";
                };
 /* End PBXNativeTarget section */
@@ -94,6 +140,9 @@
                                        0FE79B831D22BAE400E38C27 = {
                                                CreatedOnToolsVersion = 7.3.1;
                                        };
+                                       0FE79B911D22BB5E00E38C27 = {
+                                               CreatedOnToolsVersion = 7.3.1;
+                                       };
                                };
                        };
                        buildConfigurationList = 0FE79B7F1D22BAE400E38C27 /* Build␣
↪configuration list for PBXProject "foo" */;
```

```
@@ -109,6 +158,7 @@
                        projectRoot = "";
                        targets = (
                                0FE79B831D22BAE400E38C27 /* foo */,
+                               0FE79B911D22BB5E00E38C27 /* bar */,
                        );
                };
 /* End PBXProject section */
@@ -119,6 +169,14 @@
                        buildActionMask = 2147483647;
                        files = (
                                0FE79B881D22BAE400E38C27 /* main.cpp in Sources */,
+                       );
+                       runOnlyForDeploymentPostprocessing = 0;
+               };
+               0FE79B8E1D22BB5E00E38C27 /* Sources */ = {
+                       isa = PBXSourcesBuildPhase;
+                       buildActionMask = 2147483647;
+                       files = (
+                               0FE79B951D22BB5E00E38C27 /* bar.cpp in Sources */,
                        );
                        runOnlyForDeploymentPostprocessing = 0;
                };
@@ -220,6 +278,20 @@
                        };
                        name = Release;
                };
+               0FE79B961D22BB5E00E38C27 /* Debug */ = {
+                       isa = XCBuildConfiguration;
+                       buildSettings = {
+                               PRODUCT_NAME = "$(TARGET_NAME)";
+                       };
+                       name = Debug;
+               };
+               0FE79B971D22BB5E00E38C27 /* Release */ = {
+                       isa = XCBuildConfiguration;
+                       buildSettings = {
+                               PRODUCT_NAME = "$(TARGET_NAME)";
+                       };
+                       name = Release;
+               };
 /* End XCBuildConfiguration section */

 /* Begin XCConfigurationList section */
@@ -239,6 +311,15 @@
                                0FE79B8D1D22BAE400E38C27 /* Release */,
                        );
                        defaultConfigurationIsVisible = 0;
+                       defaultConfigurationName = Release;
+               };
+               0FE79B981D22BB5E00E38C27 /* Build configuration list for␣
↪PBXNativeTarget "bar" */ = {
+                       isa = XCConfigurationList;
+                       buildConfigurations = (
+                               0FE79B961D22BB5E00E38C27 /* Debug */,
+                               0FE79B971D22BB5E00E38C27 /* Release */,
+                       );
+                       defaultConfigurationIsVisible = 0;
```

```
                };
/* End XCConfigurationList section */
        };
```

As you can see there are a lot of magic happens while doing quite simple task like adding new target with one source file. Looking at the diffs above try to answer next questions:

- Are you sure that all XML sections added on purpose and was not the result of accidental clicking?
- Are you sure all this x86/x64/Win32, Debug/Release configurations connected together in right order and you haven't break something while debugging?
- Are you sure all that magic numbers was not read from your environment while you have done non-trivial scripting and is in fact some private key, token or password?
- Do you think it will be easy to resolve conflict in this file?

Luckily we have *CMake* which helps us in a neat way. We haven't touched any *CMake* syntax yet but I'm pretty sure it's quite obvious what's happening here :)

```
--- /overview/snippets/CMakeLists-old.txt
+++ /overview/snippets/CMakeLists-new.txt
@@ -2,3 +2,4 @@
 project(foo)

 add_executable(foo foo.cpp)
+add_executable(bar bar.cpp)
```

What a relief! Having such human-readable form of build system commands actually making *CMake* a convenient tool for development even if you're using only one platform.

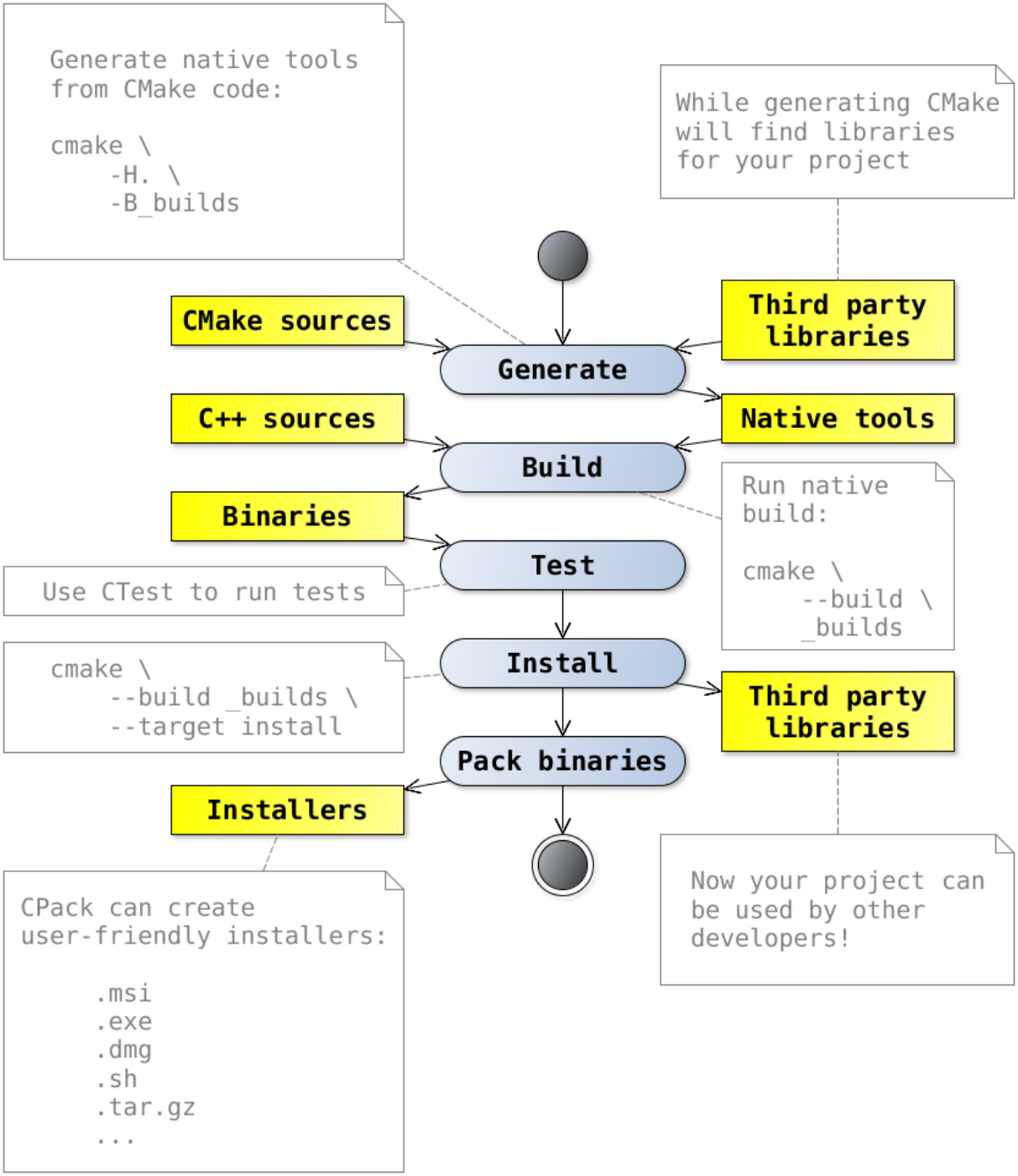
1.1.3 Experimenting

Even if your team has no plans to work with some *native tools* originally this may change in future. E.g. you have worked with `Makefile` and want to try `Ninja`. What you will do? Convert manually? Find the converter? Write converter from scratch? Write new `Ninja` configuration from scratch? With *CMake* you can change `cmake -G 'Unix Makefiles'` to `cmake -G Ninja` - done!

This helps developers of new IDEs also. Instead of putting your IDE users into situations when they have to decide should they use your `SuperDuperIDE` instead of their favorite one and probably writing endless number of `SuperDuperIDE` <-> `Xcode`, `SuperDuperIDE` <-> `Visual Studio`, etc. converters, all you have to do is to add new generator `-G SuperDuperIDE` to *CMake*.

1.1.4 Family of tools

CMake is a family of tools that can help you during all stages of `sources for developers` -> `quality control` -> `installers for users` stack. Next activity diagram shows CMake, CTest and CPack connections:



Note:

- All stages will be described fully in *Tutorials*.

See also:

- *CMake Workflow*

Hunter

- Hunter in CMake environment

1.1.5 Summary

- Human-readable configuration
- Single configuration for all tools
- Cross-platform/cross-tools friendly development
- Doesn't force you to change your favorite build tool/IDE
- *VCS* friendly development
- Easy experimenting
- Easy development of new IDEs

1.2 What can't be done with CMake

> *Good judgement comes from experience.*
> *Experience comes from bad judgement.*
> *– Mulla Nasrudin* (?)

CMake has it's strengths and weaknesses. Most of the drawbacks mentioned here can be worked around by using approaches that may differ from your normal workflow, yet still reach the end goal. Try to look at them from another angle; think of the picture as a whole and remember that the advantages definitely outweigh the disadvantages.

1.2.1 Language/syntax

This is probably the first thing you will be hit with. The *CMake* language is not something you can compare with what you have likely used before. There are no classes, no maps, no virtual functions or lambdas. Even such tasks like "parse the input arguments of a function" and "return result from a function" are quite tricky for the beginners. *CMake* is definitely not a language you want to try to experiment with implementation of red-black tree or processing JSON responses from a server. **But it does** handle regular development very efficiently and you probably will find it more attractive than XML files, autotools configs or JSON-like syntax.

Think about it in this way: if you want to do some nasty non-standard thing then probably you should stop. If you think it is something important and useful, then it might be quite useful for other *CMake* users too. In this case you need to think about implementing new feature **in CMake itself**. *CMake* is open-source project written in C++, and additional features are always being introduced. You can also discuss any problems in the CMake mailing-list to see how you can help with improving the current state.

CMake mailing list

- Wrapping CMake functionality with another language

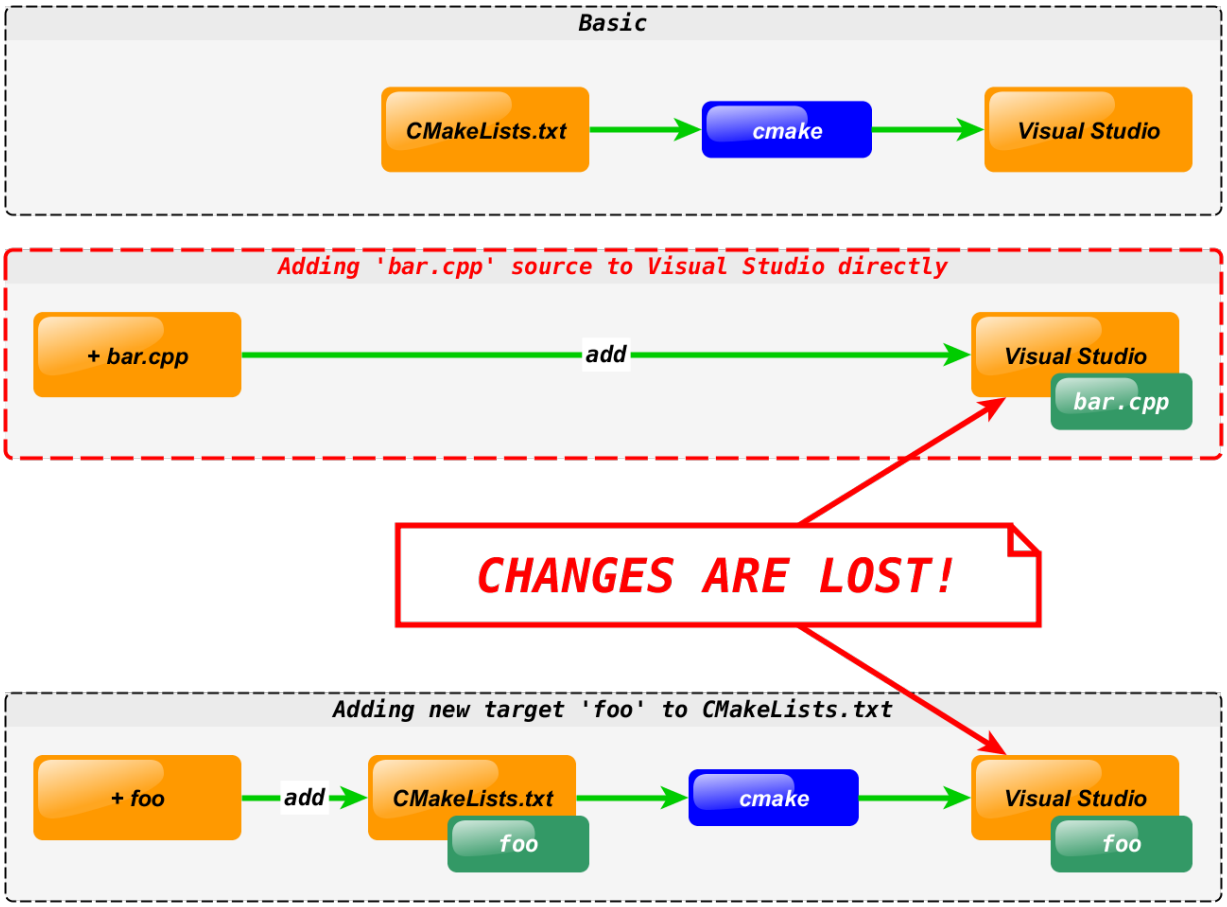
1.2.2 Affecting workflow

This might sound contradictory to the statement that you can *keep using your favorite tools*, but it's not. You still can work with your favorite IDE, but you must remember that *CMake* is now "in charge".

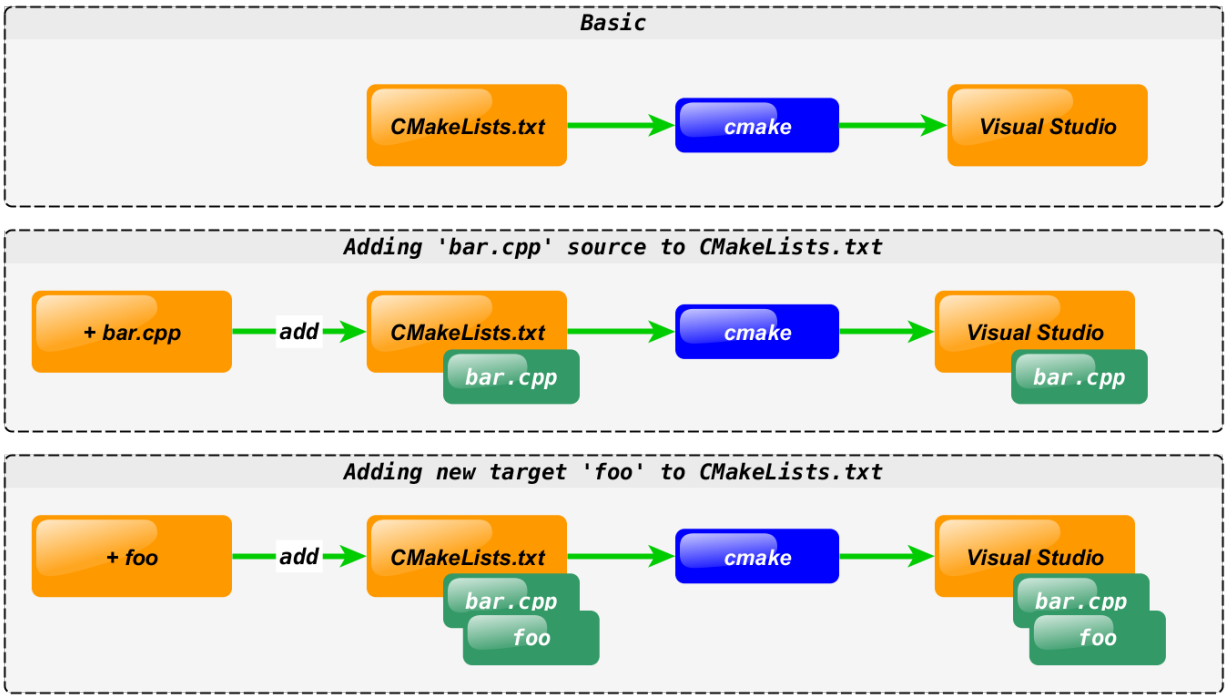
Imagine you have C++ header `version.h` generated automatically by some script from template `version.h.in`. You see `version.h` file in your IDE, you can update it and run build and new variables from `version.h` will be used in binary, but you **should never** do it since you know that source is actually `version.h.in`.

Similarly, when you use *CMake* - you **should never** update your build configuration directly in the IDE. Instead, you have to remember that any target files generated from `CMakeLists.txt` and all your project additions made directly in the IDE will be lost next time you run *CMake*.

Wrong workflow:



Correct workflow:



It's not enough to know that if you want to add a new library to your `Visual Studio` solution you can do:

- *Add → New Project ... → Visual C++ → Static Library*

You have to know that this must instead be done by adding a new `add_library` command to `CMakeLists.txt`.

1.2.3 Incomplete functionality coverage

There are some missing features in *CMake*. Mapping of CMake functionality <-> *native build tool* functionality is not always bijective. Often this can be worked around by generating different native tool files from the same CMake code. For example it's possible using autotools create two versions of library (*shared + static*) by one run. However this may affect performance, or be outright impossible for other platforms (e.g. on Windows). With *CMake* you can generate two versions of project from one `CMakeLists.txt` file: one each for shared and static variants, effectively running generate/build twice.

With `Visual Studio` you can have two variants, x86 and x64, in one solution file. With *CMake* you have to generate project twice: once with `Visual Studio` generator and one more time with `Visual Studio Win64` generator.

Similarly with `Xcode`. In general *CMake* can't mix two different toolchains (at least for now) so it's not possible to generate an `Xcode` project with `iOS` and `OSX` targets—again, just generate code for each platform independently.

Note:

- *Building universal iOS libraries*

1.2.4 Unrelocatable projects

Internally, *CMake* saves the full paths to each of the sources, so it's not possible to generate a project then share it between several developers. In other words, you can't be "the CMake person" who will generate separate projects

for those who use Xcode and those who use Visual Studio. All developers in the team should be aware of how to generate projects using CMake. In practice it means they have to know which CMake arguments to use, some basic examples being `cmake -H. -B_builds -GXcode` and `cmake -H. -B_builds "-GVisual Studio 12 2013"` for Xcode and Visual Studio, respectively. Additionally, they must understand the *changes they must make in their workflow*. As a general rule, developers should make an effort to learn the tools used in making the code they wish to utilize. Only when providing an end product to users is it your responsibility to generate user-friendly installers like `*.msi` instead of simply providing the project files.

CMake documentation

- CMAKE_USE_RELATIVE_PATHS removed since CMake 3.4

Even if support for relative paths will be re-implemented in the future each developer in the team should have *CMake* installed, as there are other tasks which *CMake* automatically takes care of that may be done incorrectly if done manually. A few examples are:

- The automatic detection of changes to `CMakeLists.txt` and subsequent regeneration of the source tree.
- The inclusion of custom build steps with the built-in scripting mode.
- For doing internal stuff like searching for installed dependent packages

---

TODO

Link to relocatable packages

CHAPTER 2

First step

Okay, time to run some code! Now we will check tools we need, create project with one executable, will build it and run. Try to follow instructions **accurately**. The goal of this section is to run simplest configuration with commonly/widely used tools. After you've checked that everything fine and feel comfortable you can find more options in: *Platforms*, *Generators* and *Compilers*. Each command's usage/pitfalls will be described in depth further in *Tutorials*.

2.1 CMake Installation

That's it, ground.
I wonder if it will be friends with me?
Hello, ground!
– Whale

Obviously to use some tool you need to install it first. CMake can be installed using default system package manager or by getting binaries from Download page.

2.1.1 Ubuntu

CMake can be installed by `apt-get`:

```
> sudo apt-get -y install cmake
> which cmake
/usr/bin/cmake
> cmake --version
cmake version 2.8.12.2
```

Installing CMake GUI is similar:

```
> sudo apt-get -y install cmake-qt-gui
> which cmake-gui
/usr/bin/cmake-gui
> cmake-gui --version
cmake version 2.8.12.2
```

Binaries can be downloaded and unpacked manually to any location:

```
> wget https://cmake.org/files/v3.4/cmake-3.4.1-Linux-x86_64.tar.gz
> tar xf cmake-3.4.1-Linux-x86_64.tar.gz
> export PATH="`pwd`/cmake-3.4.1-Linux-x86_64/bin:$PATH" # save it in .bashrc if␣
↪needed
> which cmake
/.../cmake-3.4.1-Linux-x86_64/bin/cmake
> which cmake-gui
/.../cmake-3.4.1-Linux-x86_64/bin/cmake-gui
```

Version:

```
> cmake --version
cmake version 3.4.1

CMake suite maintained and supported by Kitware (kitware.com/cmake).
> cmake-gui --version
cmake version 3.4.1

CMake suite maintained and supported by Kitware (kitware.com/cmake)
```

2.1.2 OS X

CMake can be installed on Mac using brew:

```
> brew install cmake
> which cmake
/usr/local/bin/cmake
> cmake --version
cmake version 3.4.1

CMake suite maintained and supported by Kitware (kitware.com/cmake)
```

Binaries can be downloaded and unpacked manually to any location:

```
> wget https://cmake.org/files/v3.4/cmake-3.4.1-Darwin-x86_64.tar.gz
> tar xf cmake-3.4.1-Darwin-x86_64.tar.gz
> export PATH="`pwd`/cmake-3.4.1-Darwin-x86_64/CMake.app/Contents/bin:$PATH"
> which cmake
/.../cmake-3.4.1-Darwin-x86_64/CMake.app/Contents/bin/cmake
> which cmake-gui
/.../cmake-3.4.1-Darwin-x86_64/CMake.app/Contents/bin/cmake-gui
```

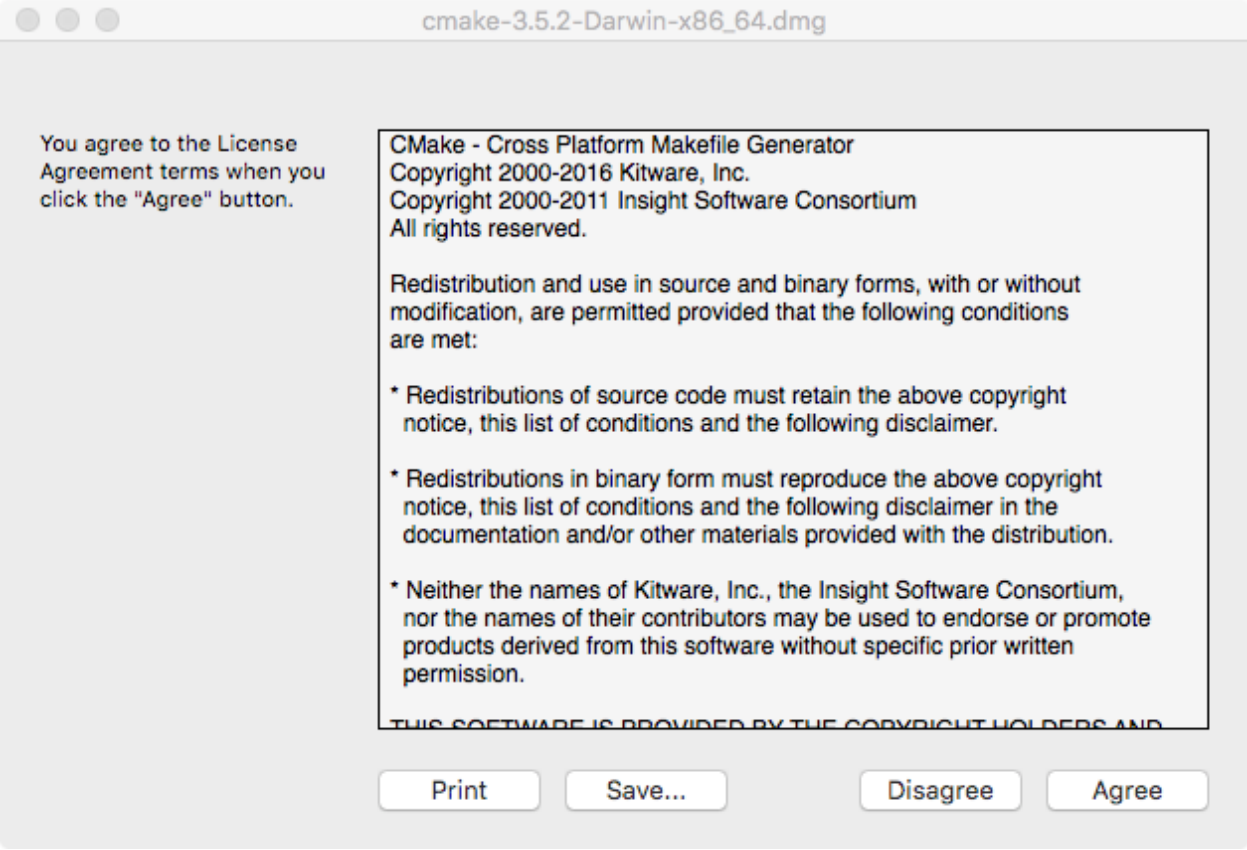
Version:

```
> cmake --version
cmake version 3.4.1

CMake suite maintained and supported by Kitware (kitware.com/cmake).
> cmake-gui --version
cmake version 3.4.1

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

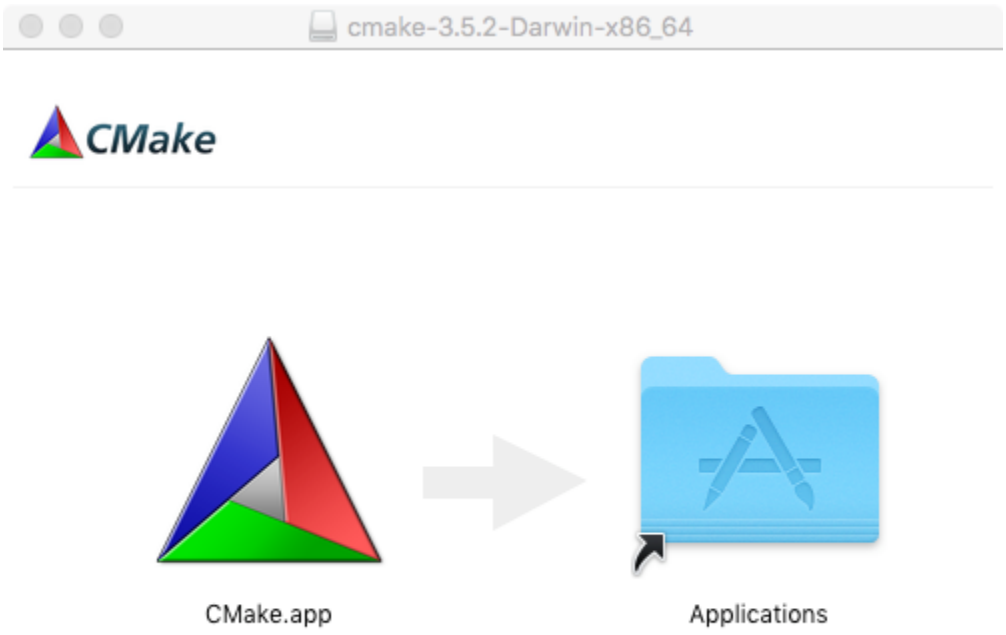
DMG installer

Download `cmake-*.dmg` installer from Download page and run it.

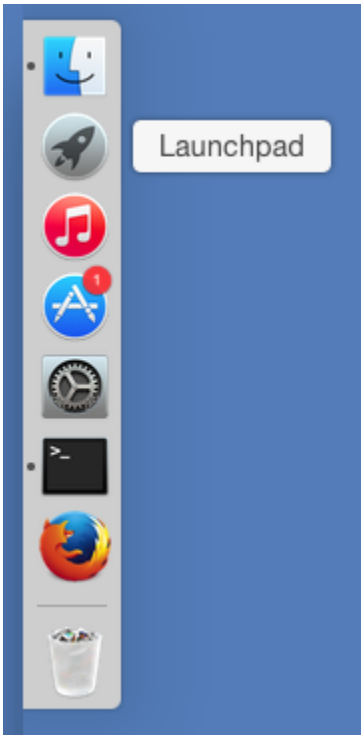
Click `Agree`:



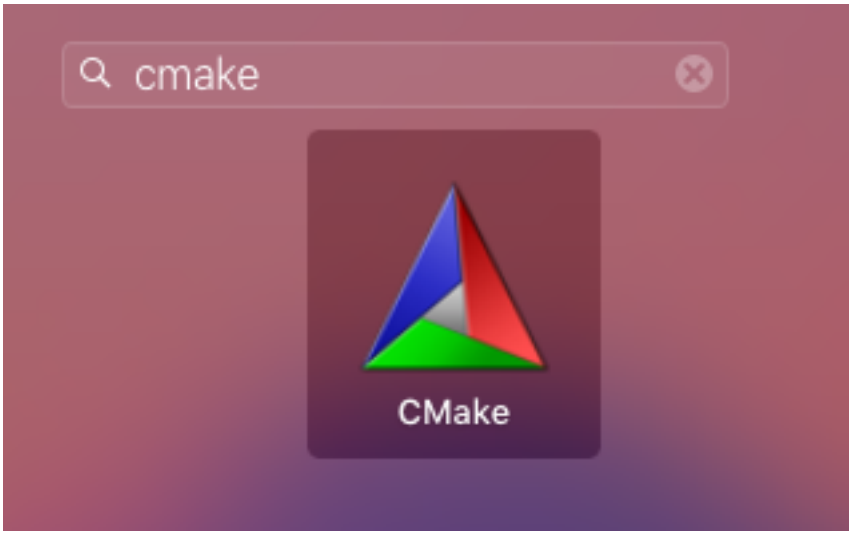
Drag `CMake.app` to `Applications` folder (or any other location):



Start `Launchpad`:



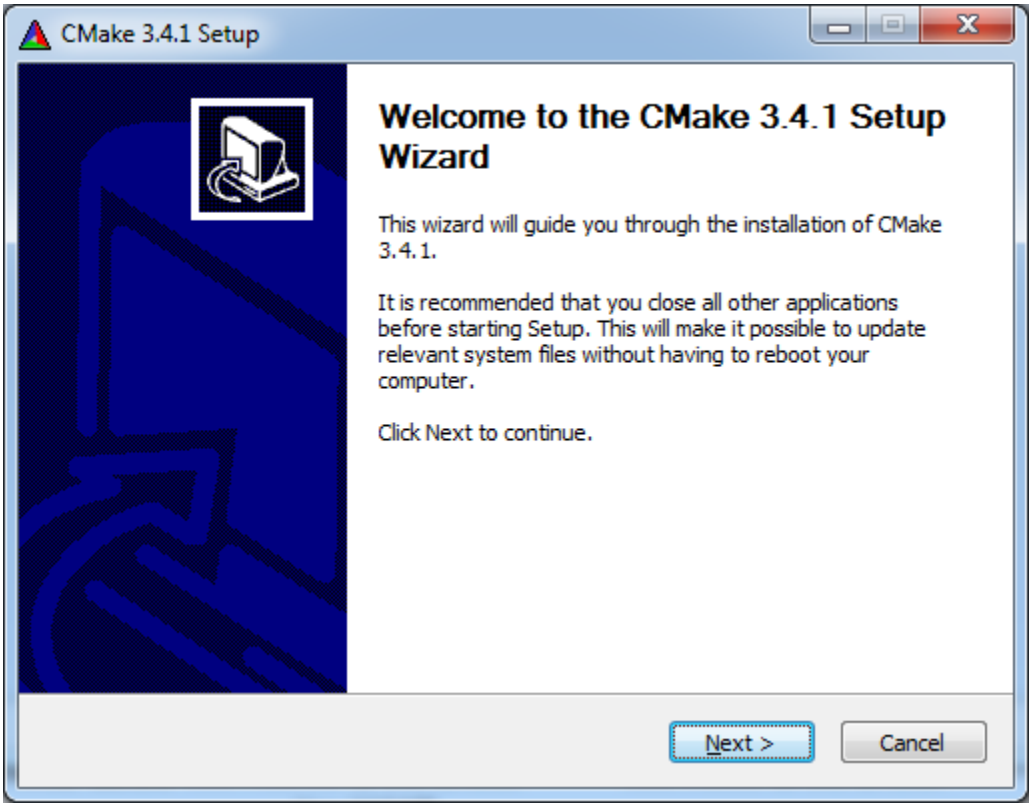
Find `CMake` and launch it:



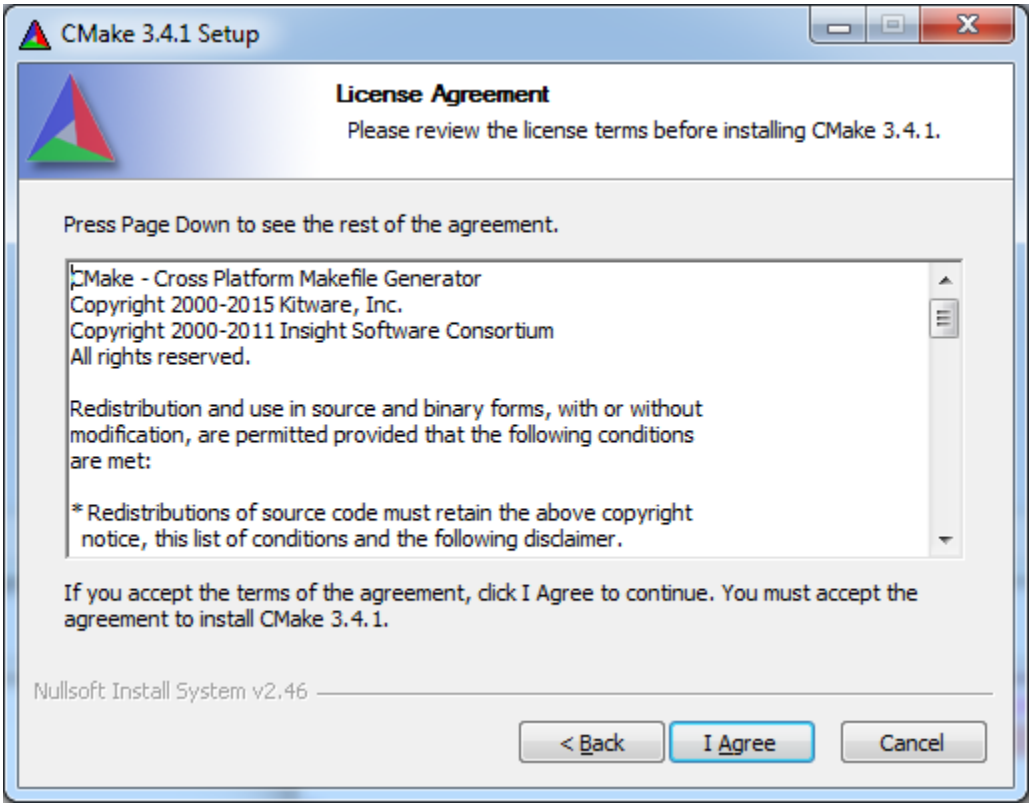
2.1.3 Windows

Download `cmake-*.exe` installer from Download page and run it.

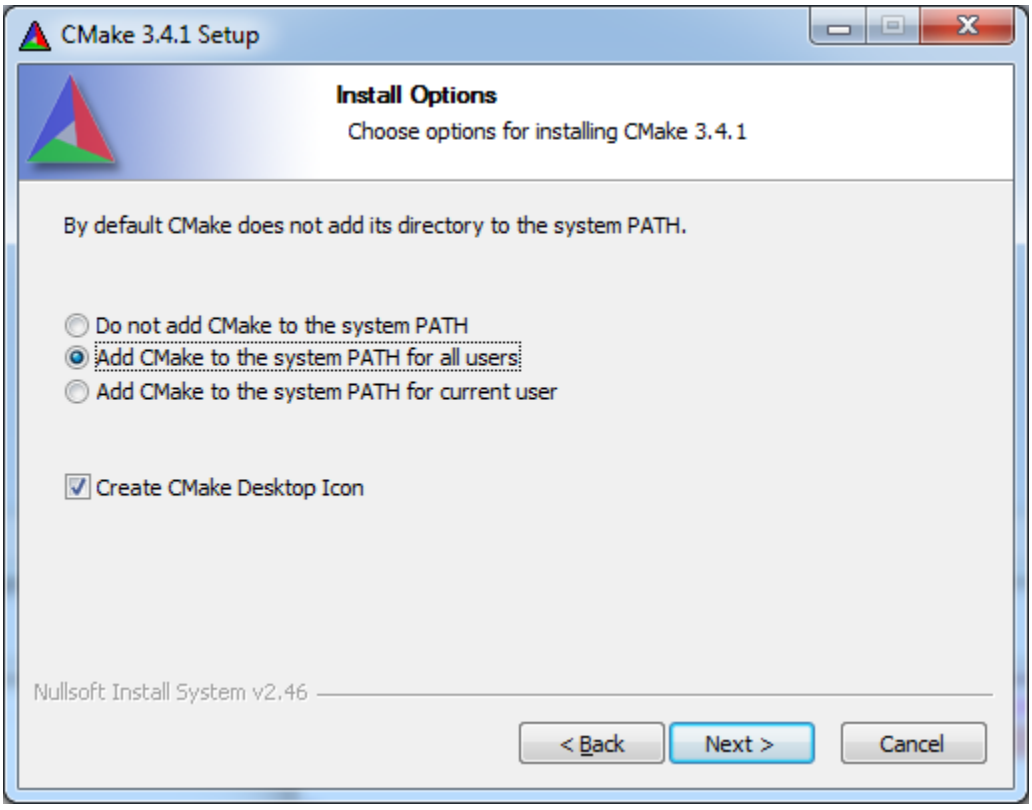
Click `Next`:



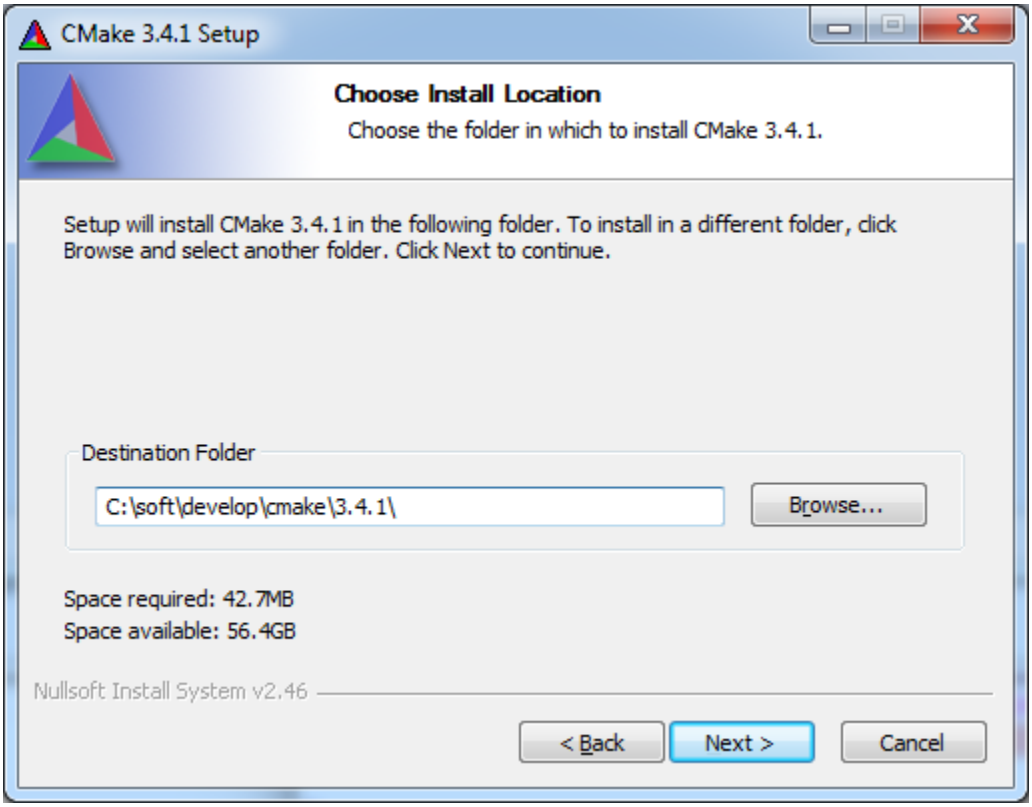
Click `I agree`:



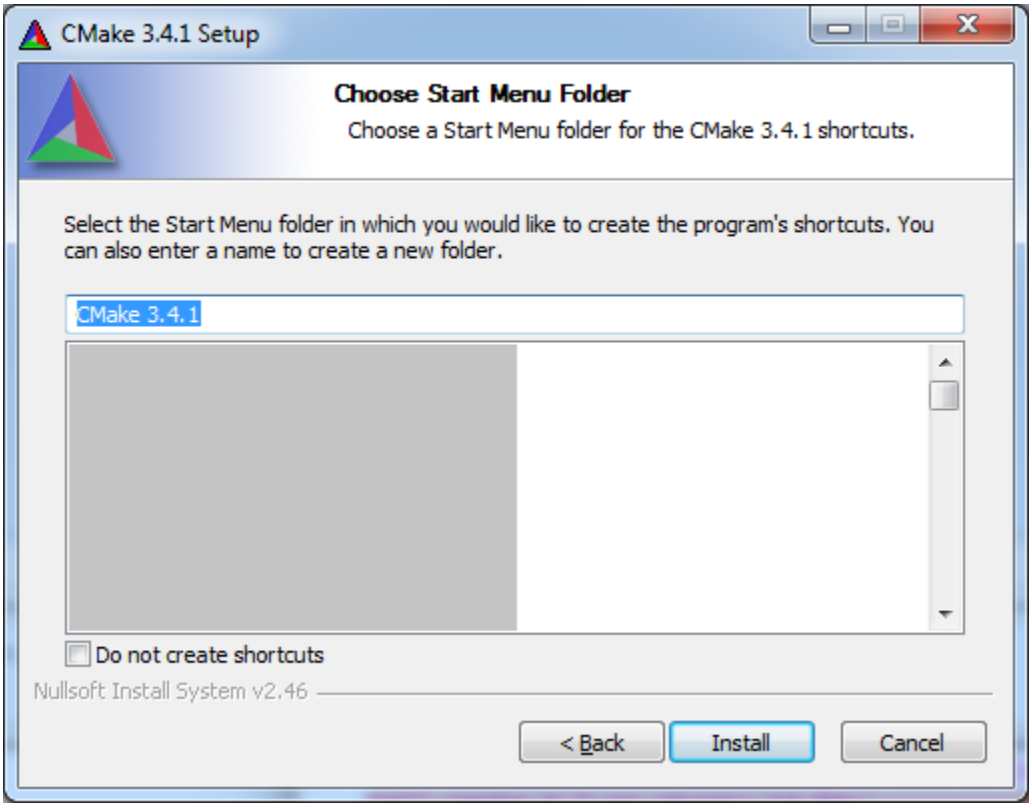
Check one of the `Add CMake to the system PATH ...` if you want to have CMake in `PATH`. Check `Create CMake Desktop Icon` to create icon on desktop:



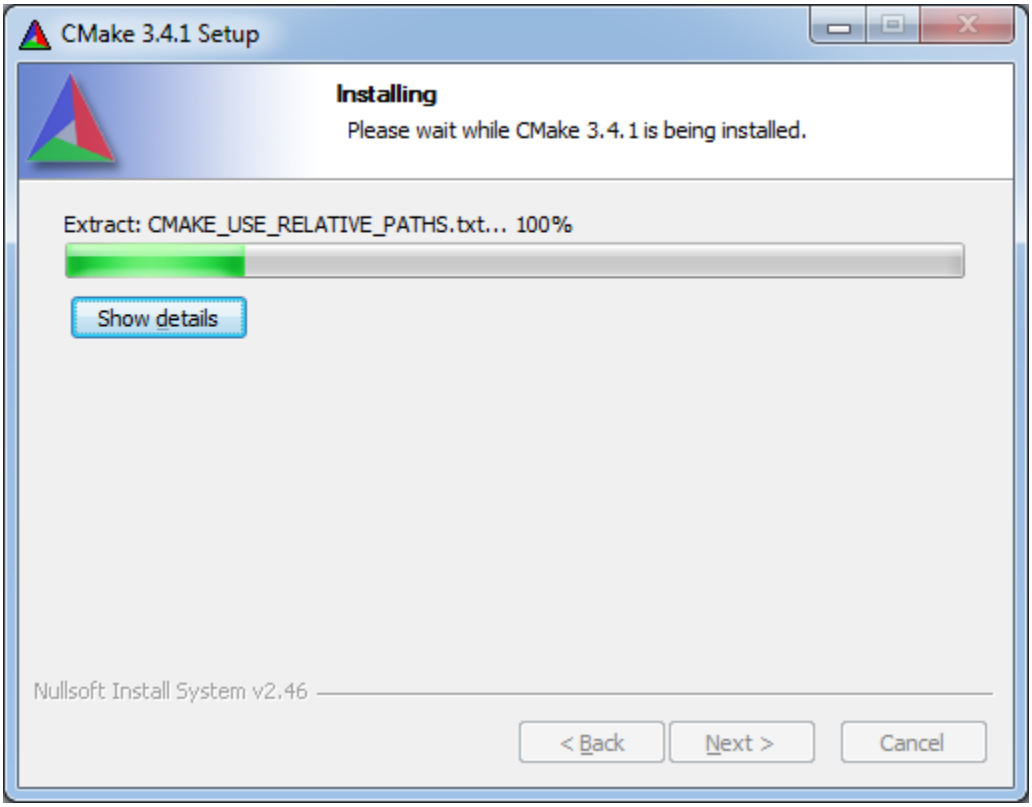
Choose installation path. Add suffix with version in case you want to have several versions installed simultaneously:



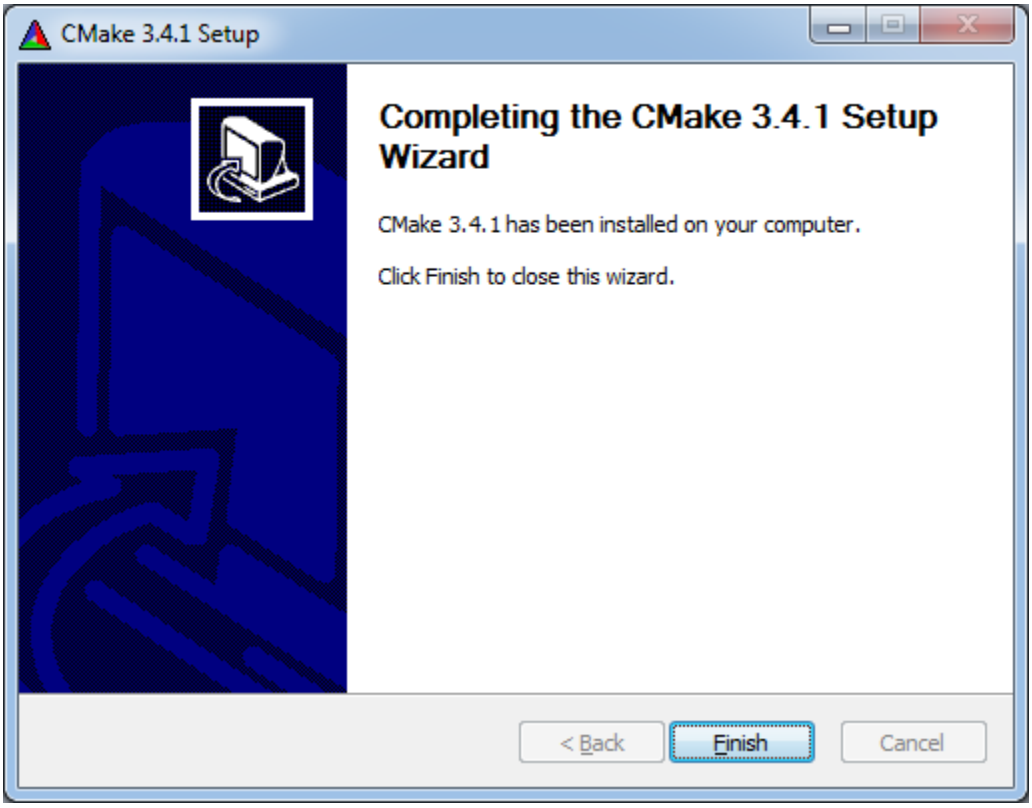
Shortcut in Start Menu folder:



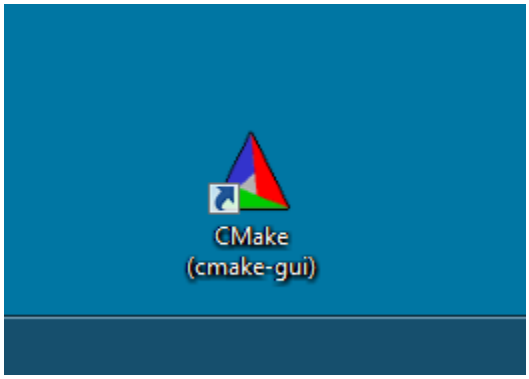
Installing...



Click Finish:



Desktop icon created:



If you set `Add CMake to the system PATH ...` checkbox then CMake can be accessed via terminal (otherwise you need to add `...\bin` to PATH environment variable):

```
> where cmake
C:\soft\develop\cmake\3.4.1\bin\cmake.exe

> where cmake-gui
C:\soft\develop\cmake\3.4.1\bin\cmake-gui.exe

> cmake --version
cmake version 3.4.1

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

See also:

- Installing CMake
- How to install cmake 3.2 on ubuntu 14.04?

2.2 Native build tool

As already mentioned CMake is **not designed** to do the build itself - it *generates files* which can be used by real *native build tool*, hence you need to choose such tool(s) and install it if needed. Option -G <generator-name> can be used to specify what type of generator will be run. If no such option present CMake will use default generator (e.g. `Unix Makefiles` on *nix platforms).

List of available generators depends on the host OS (e.g. `Visual Studio` family generators not available on `Linux`). You can get this list by running `cmake --help`:

```
> cmake --help
...
Generators

The following generators are available on this platform:
  Unix Makefiles              = Generates standard UNIX makefiles.
  Ninja                       = Generates build.ninja files (experimental).
  Watcom WMake                = Generates Watcom WMake makefiles.
  CodeBlocks - Ninja          = Generates CodeBlocks project files.
  ...
```

2.2.1 Visual Studio

`Visual Studio` is an IDE created by `Microsoft`. Here are the links to the community versions:

- Visual Studio Community 2017
- Visual Studio Community 2015
- Visual Studio Community 2013

See also:

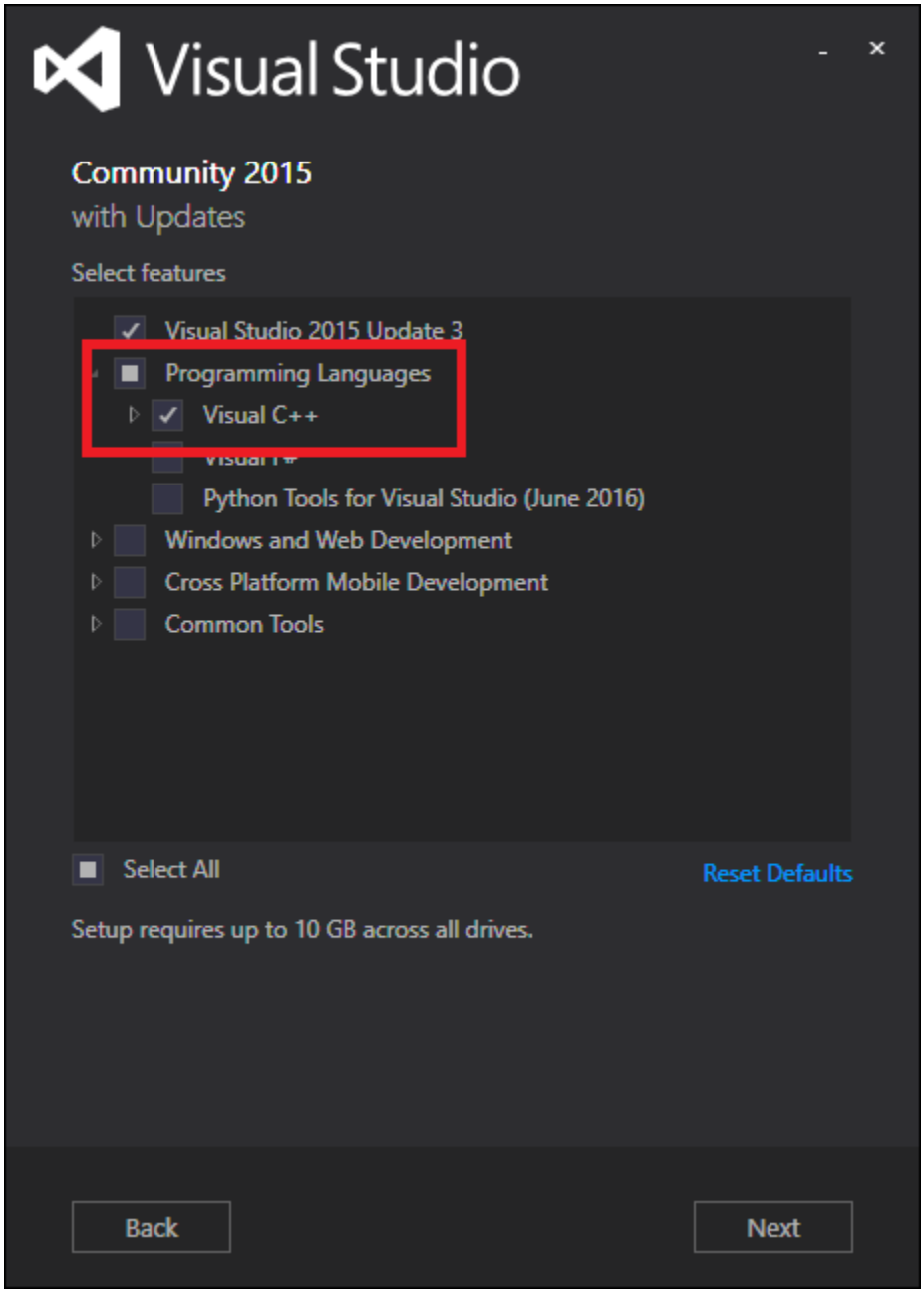
- Official site

Wikipedia

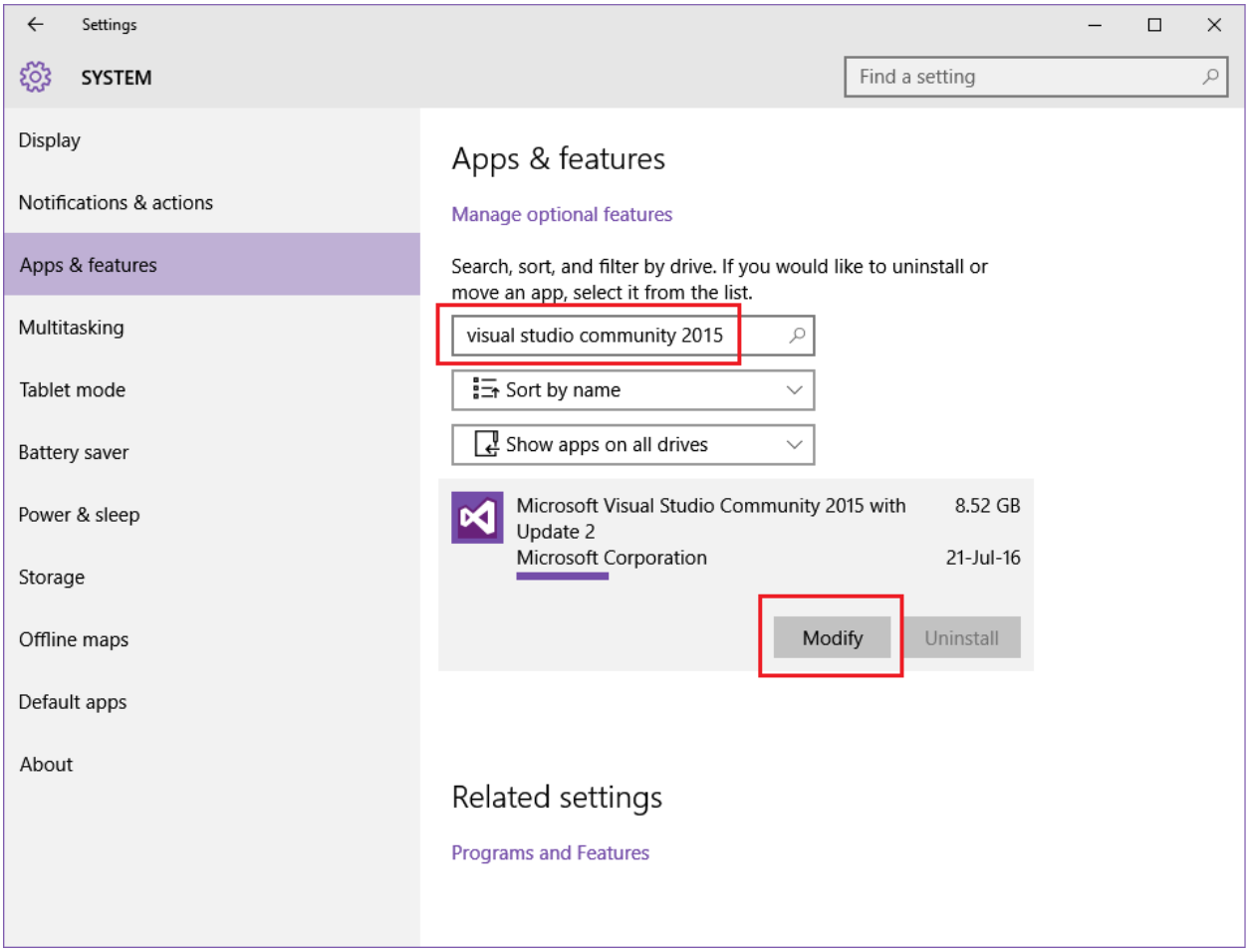
- Visual Studio

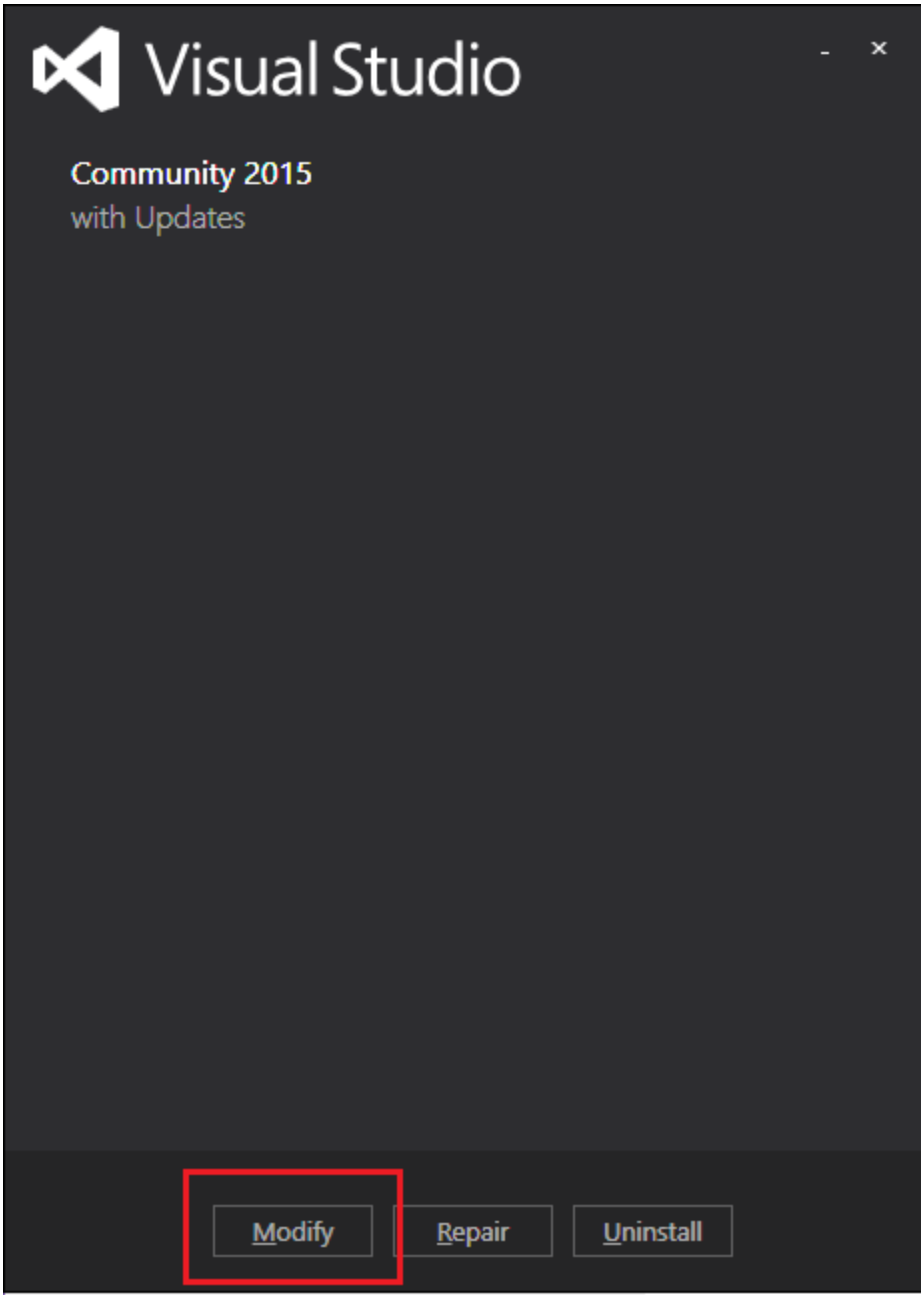
Manage features

Installer will offer you the menu to manage features you need. Don't forget to add *Programming Languages* → *Visual C++*:



If you already have Visual Studio installed you can go to *System* → *Apps & features* → *Modify*:


Settings
SYSTEM
Find a setting
Display
Notifications & actions
Apps & features
Multitasking
Tablet mode
Battery saver
Power & sleep
Storage
Offline maps
Default apps
About
Apps & features
Manage optional features
Search, sort, and filter by drive. If you would like to uninstall or move an app, select it from the list.
visual studio community 2015
Sort by name
Show apps on all drives
Microsoft Visual Studio Community 2015 with Update 2
8.52 GB
Microsoft Corporation
21-Jul-16
Modify
Uninstall
Related settings
Programs and Features



See also:

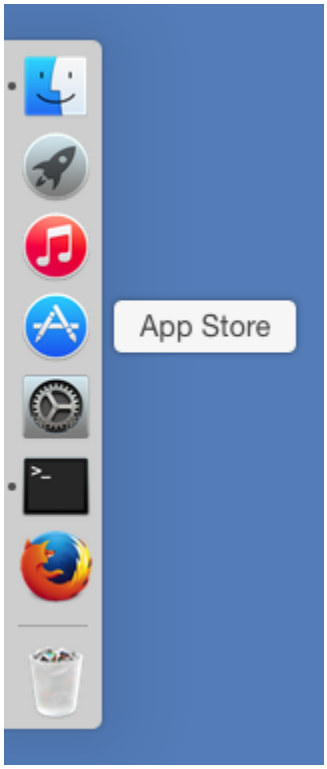
- CMake Tools for Visual Studio
- VsVim
- Editor Guidelines
- *Developer Command Prompt*

2.2.2 Xcode

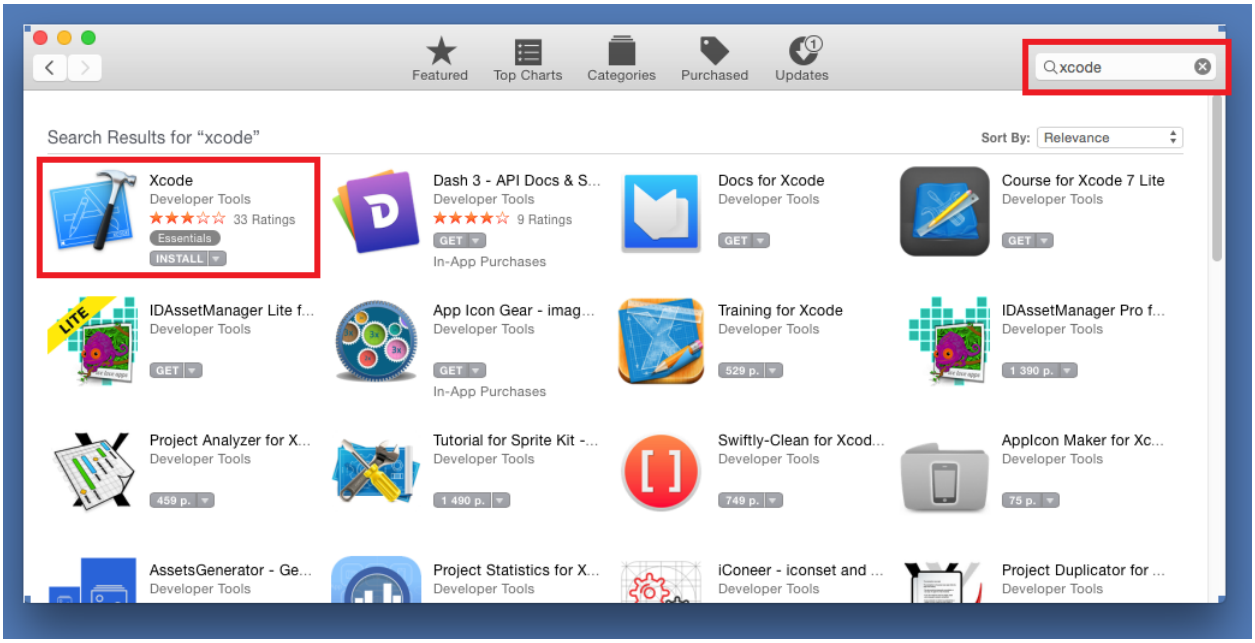
`Xcode` is an IDE for OSX/iOS development (Wikipedia).

Default install with App Store

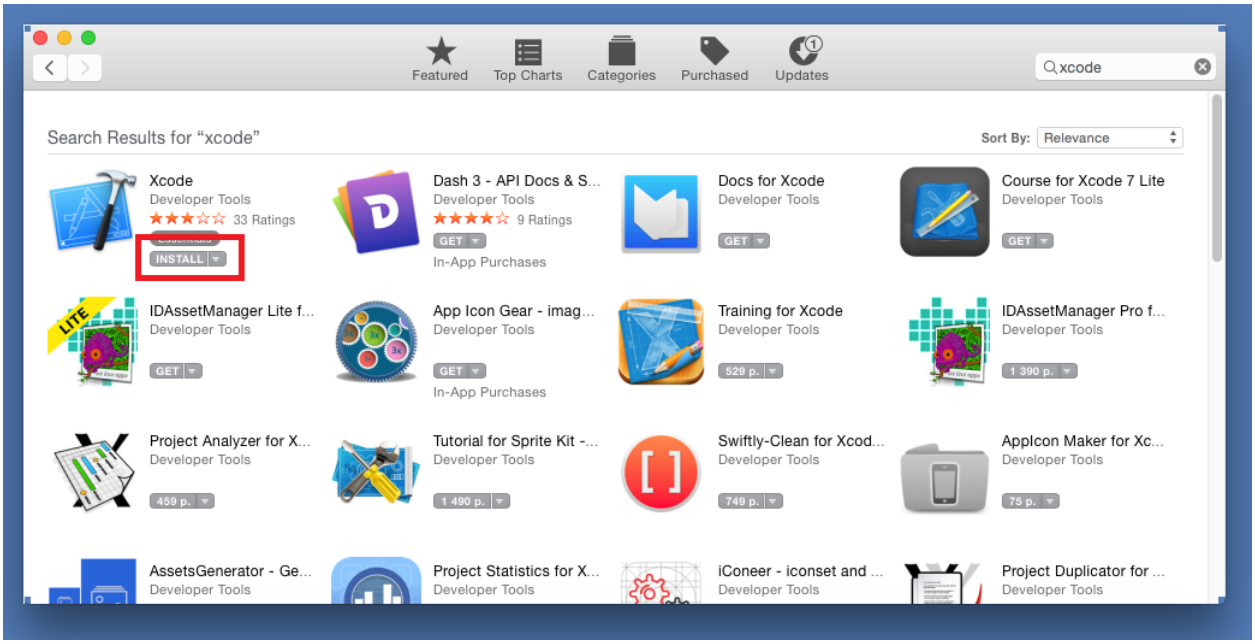
Go to `App Store`:



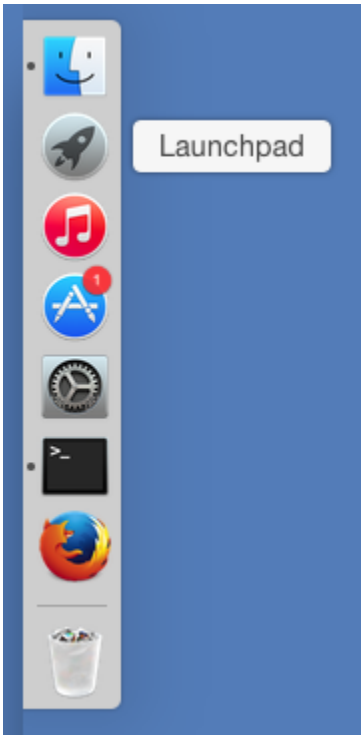
Search for `Xcode` application:



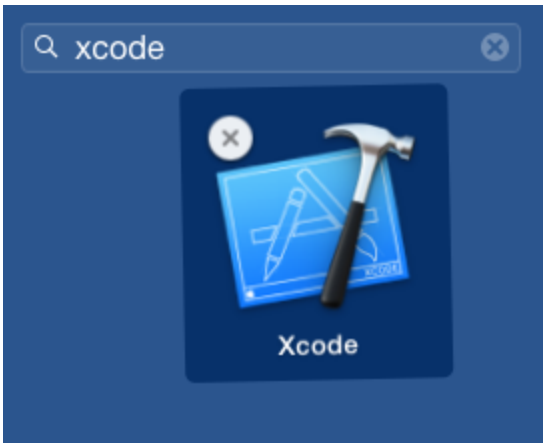
Run install:



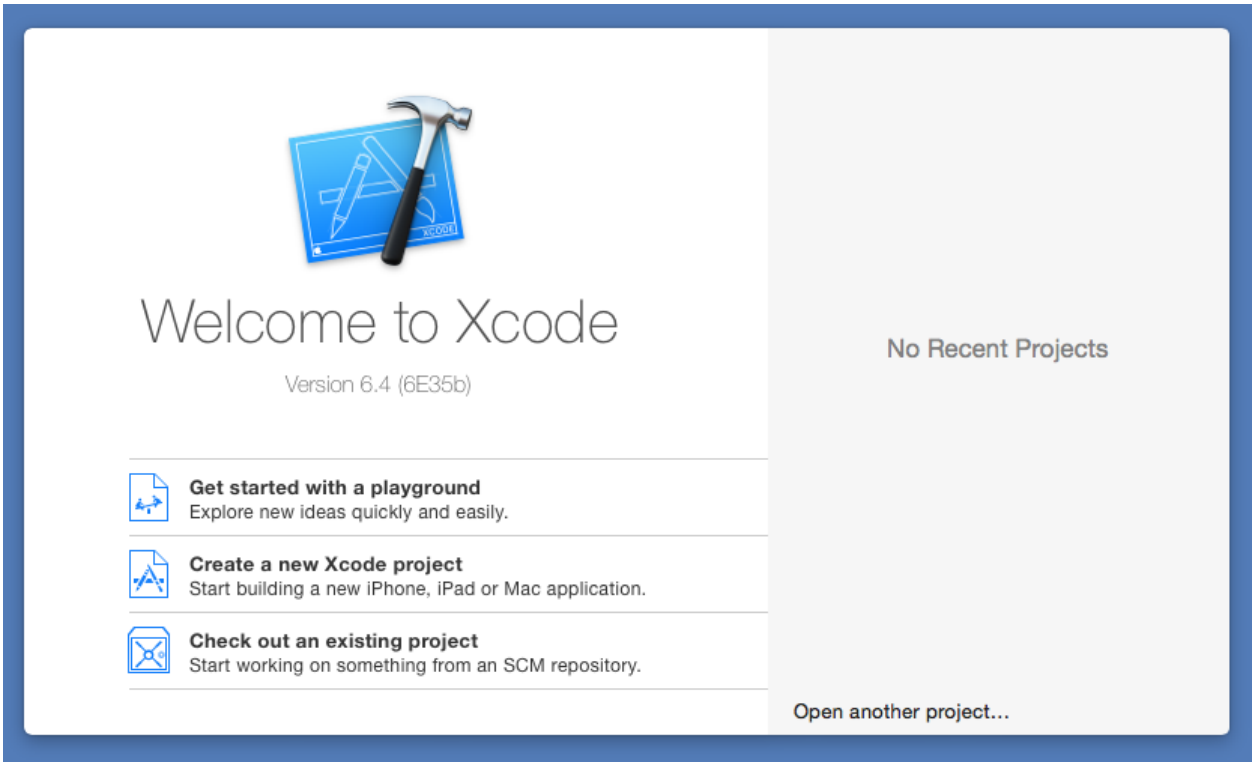
After successful install run `Launchpad`:



Search for `Xcode` and launch it:



Success!



Note: Other developer tools is *installed now too*.

Several/custom Xcode versions

If you want to have several `Xcode` versions simultaneously for testing purposes or you want the exact version of `Xcode` you can download/install it manually from Apple Developers site.

For example:

```
> ls /Applications/develop/ide/xcode
4.6.3/
5.0.2/
```

```
6.1/
6.4/
7.2/
7.2.1/
7.3.1/
```

Default directory and version can be checked by `xcode-select`/`xcodebuild` tools:

```
> xcode-select --print-path
/Applications/develop/ide/xcode/7.3.1/Xcode.app/Contents/Developer

> xcodebuild -version
Xcode 7.3.1
Build version 7D1014
```

Default version can be changed by `xcode-select -switch`:

```
> sudo xcode-select -switch /Applications/develop/ide/xcode/7.2/Xcode.app/Contents/
 ↪Developer

> xcodebuild -version
Xcode 7.2
Build version 7C68
```

Or using environment variable `DEVELOPER_DIR`:

```
> export DEVELOPER_DIR=/Applications/develop/ide/xcode/7.3.1/Xcode.app/Contents/
 ↪Developer
> xcodebuild -version
Xcode 7.3.1
Build version 7D1014

> export DEVELOPER_DIR=/Applications/develop/ide/xcode/7.2/Xcode.app/Contents/
 ↪Developer
> xcodebuild -version
Xcode 7.2
Build version 7C68
```

See also:

- Polly iOS toolchains

2.2.3 Unix Makefiles

- CMake option: `-G "Unix Makefiles"`

CMake documentation

- Unix Makefiles

Wikipedia

- Make

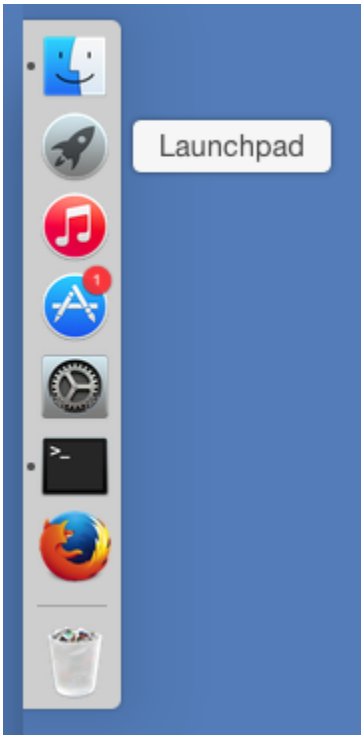
Ubuntu Installation

```
> sudo apt-get -y install make
> make -v
GNU Make 3.81
...
```

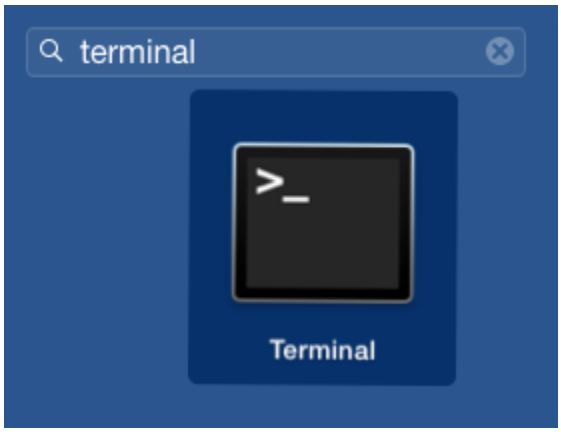
OSX Installation

If you're planning to install *Xcode* then install it first. `make` and other tools go with `Xcode`. Otherwise `make` can be installed with `Command line tools` only.

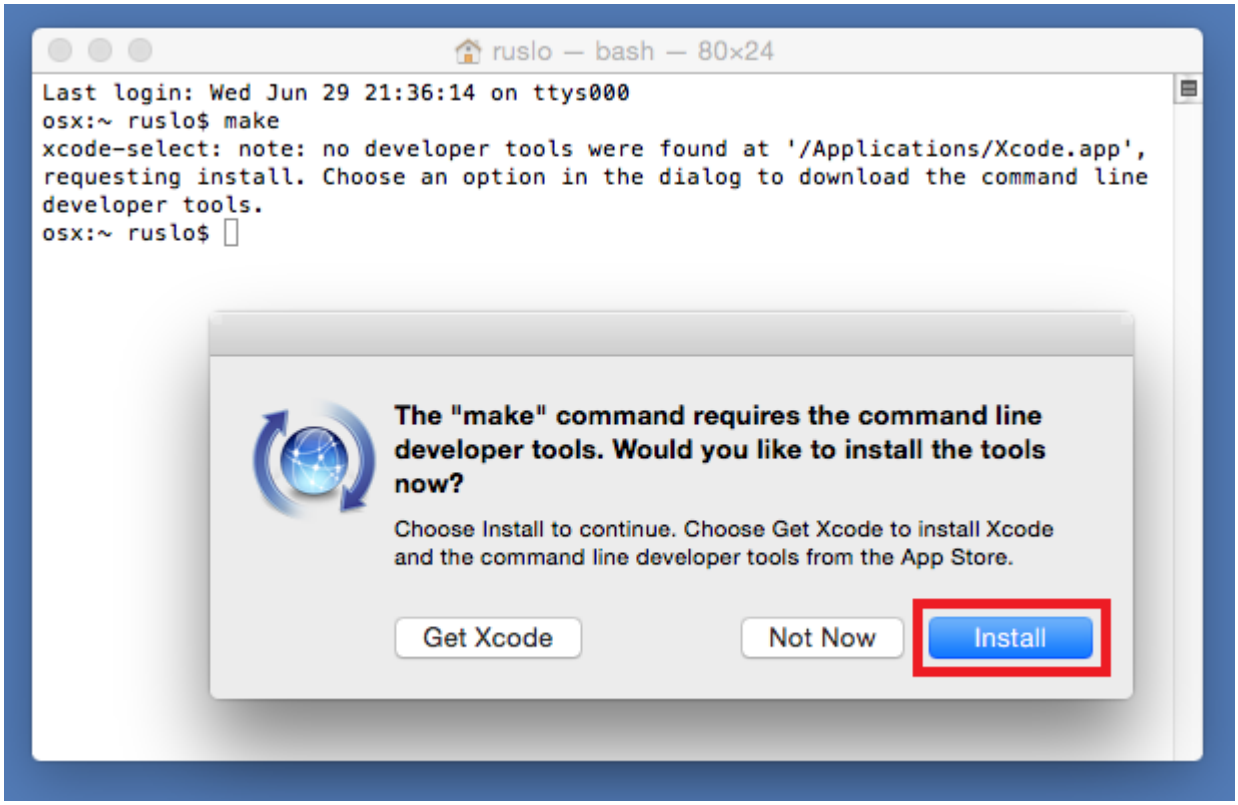
Run `Launchpad`:



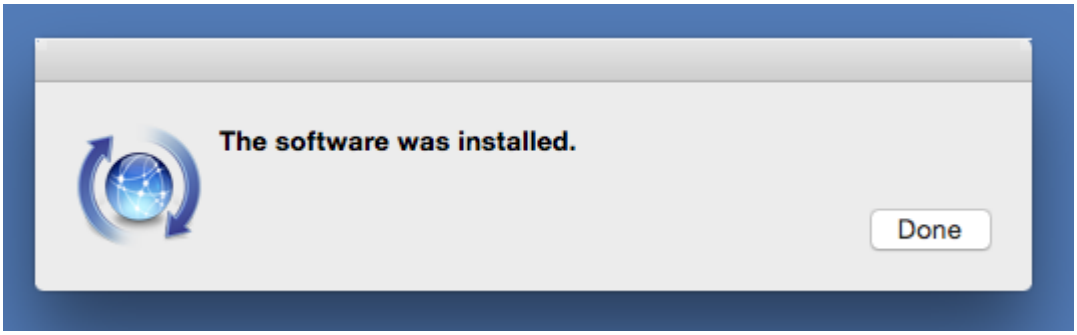
Find `Terminal` and launch it:



Try to execute `make` (or any other commands for development like GCC, git, clang, etc.). Popup dialog window will appear:



Click `Install` wait until it finished with successful message:



Check `make` location and version:

```
> which make
/usr/bin/make

> make --version
GNU Make 3.81
Copyright (C) 2006  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

This program built for i386-apple-darwin11.3.0
```

`Clang` will be installed too:

```
> which clang
/usr/bin/clang

> clang --version
Apple LLVM version 7.0.2 (clang-700.1.81)
Target: x86_64-apple-darwin14.0.0
Thread model: posix
```

As well as `GCC`:

```
> which gcc
/usr/bin/gcc

> gcc --version
Configured with: --prefix=/Library/Developer/CommandLineTools/usr --with-gxx-include-
↪dir=/usr/include/c++/4.2.1
Apple LLVM version 7.0.2 (clang-700.1.81)
Target: x86_64-apple-darwin14.0.0
Thread model: posix
```

CMake documentation

- CMake Generators

2.3 Compiler

Native build tool will only orchestrate our builds but we need to have the compiler which will actually create binaries from our C++ sources.

2.3.1 Visual Studio

Visual Studio compiler (aka `cl.exe`) will be *installed with IDE*, no additional steps needed.

2.3.2 Ubuntu GCC

`GCC` compiler usually used on `Linux OS`. To install it on `Ubuntu` run:

```
> sudo apt-get install -y gcc
```

Check location and version

```
> which gcc
/usr/bin/gcc

> gcc --version
gcc (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4
Copyright (C) 2013 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

2.3.3 OSX Clang

`Clang` compiler will be installed with *Xcode* or while installing *make*.

2.4 Minimal example

Create empty directory and put `foo.cpp` and `CMakeLists.txt` files into it.

Examples on GitHub

- Repository
- Latest ZIP

`foo.cpp` is a C++ source of our executable:

```
// foo.cpp

#include <iostream> // std::cout

int main() {
  std::cout << "Hello from CGold!" << std::endl;
}
```

`CMakeLists.txt` is a project configuration, i.e. source for *CMake*:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_executable(foo foo.cpp)
```

2.4.1 Description

foo.cpp

Explanation of the `foo.cpp` content is out of scope of this document so will be skipped.

CMakeLists.txt

First line of `CMakeLists.txt` is a comment and will be ignored:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_executable(foo foo.cpp)
```

Next line tell us about `CMake` version for which this file is written:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_executable(foo foo.cpp)
```

`2.8` means we can use this configuration with `CMake` versions like `2.8`, `2.8.7`, `3.0`, `3.5.1`, etc. but not with `2.6.0` or `2.4.2`.

Declaration of the project `foo`, e.g. `Visual Studio` solution will have name `foo.sln`, `Xcode` project name will be `foo.xcodeproj`:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_executable(foo foo.cpp)
```

Adding executable `foo` with source `foo.cpp`:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_executable(foo foo.cpp)
```

CMake has some predefined settings so it will figure out next things:

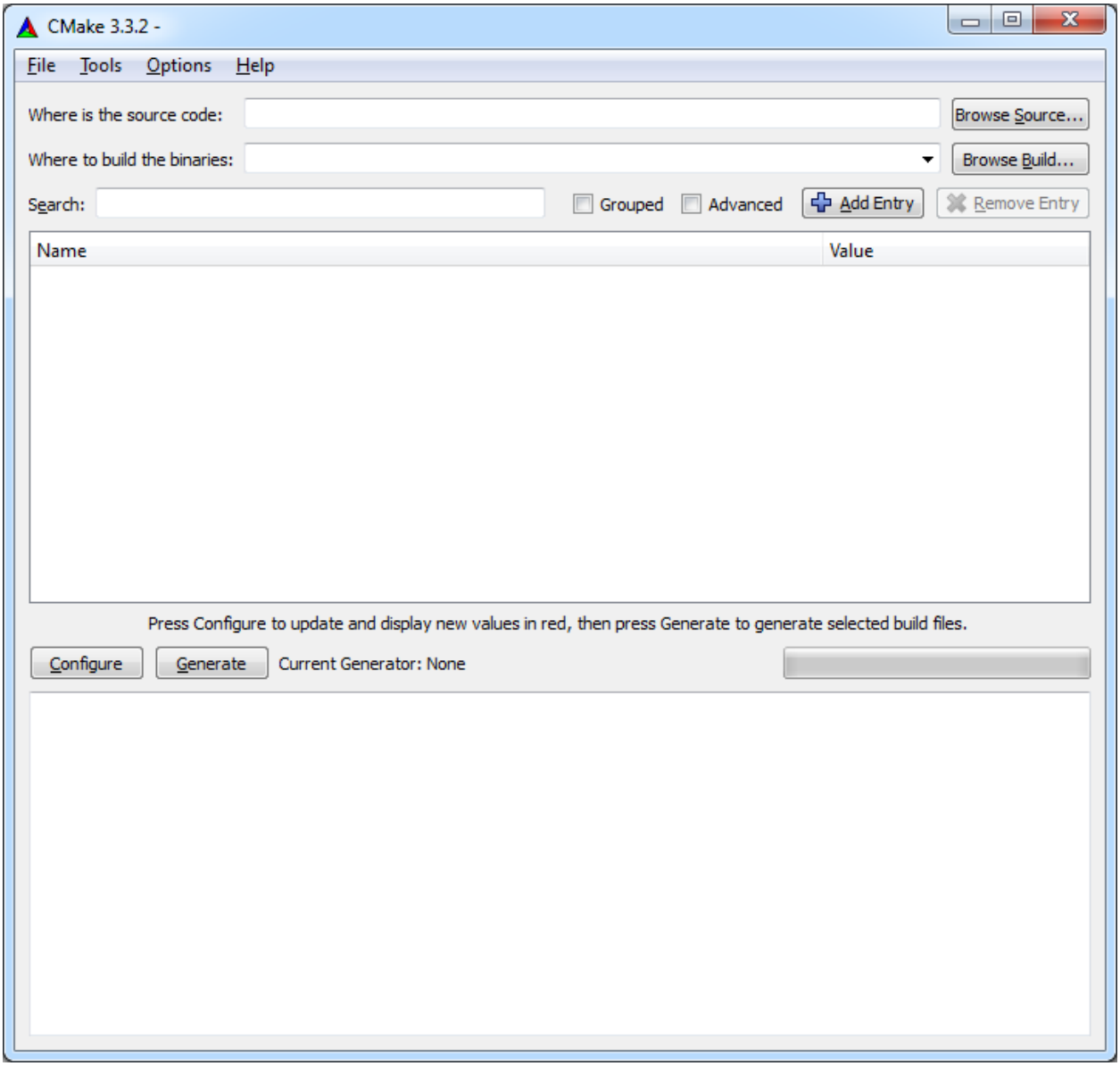
- `*.cpp` extension is for the C++ sources, so target `foo` will be build with C++ compiler
- on Windows executables usually have suffix `.exe` so result binary will have name `foo.exe`
- on Unix platforms like OSX or Linux executables usually have no suffixes so result binary will have name `foo`

2.5 Generate native tool files

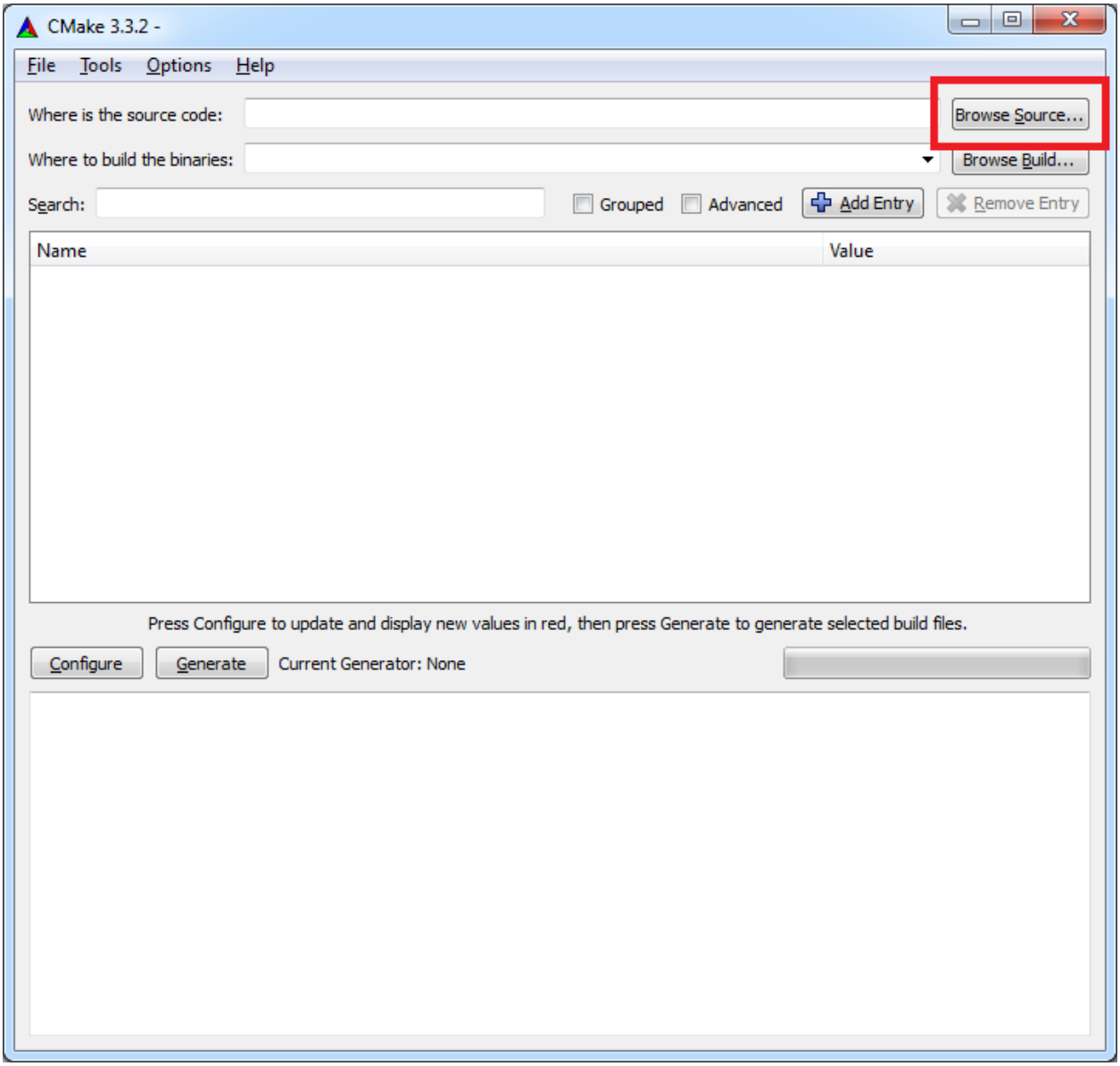
You can use GUI or command-line version of CMake to generate native files.

2.5.1 GUI: Visual Studio

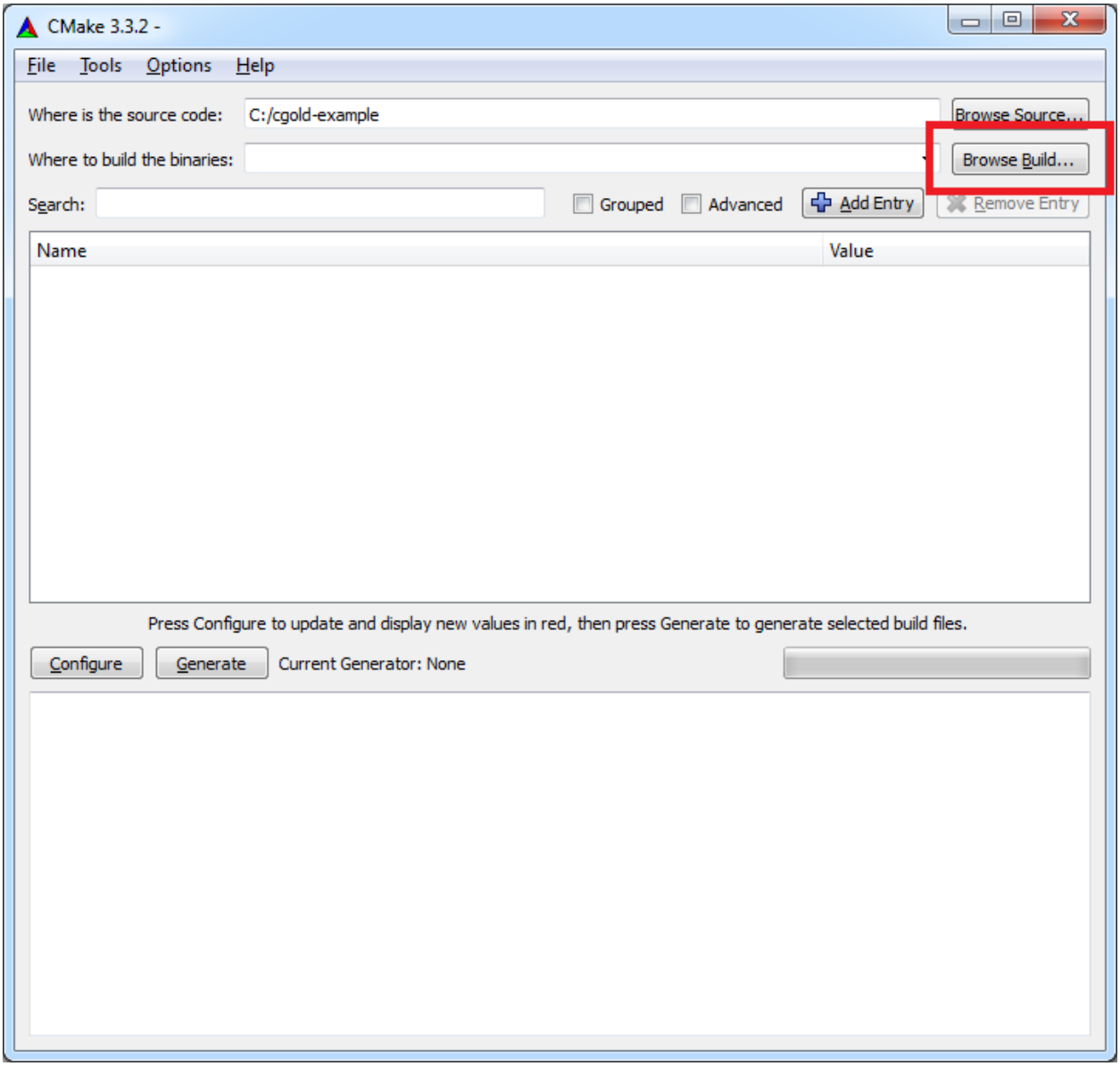
Open CMake GUI:



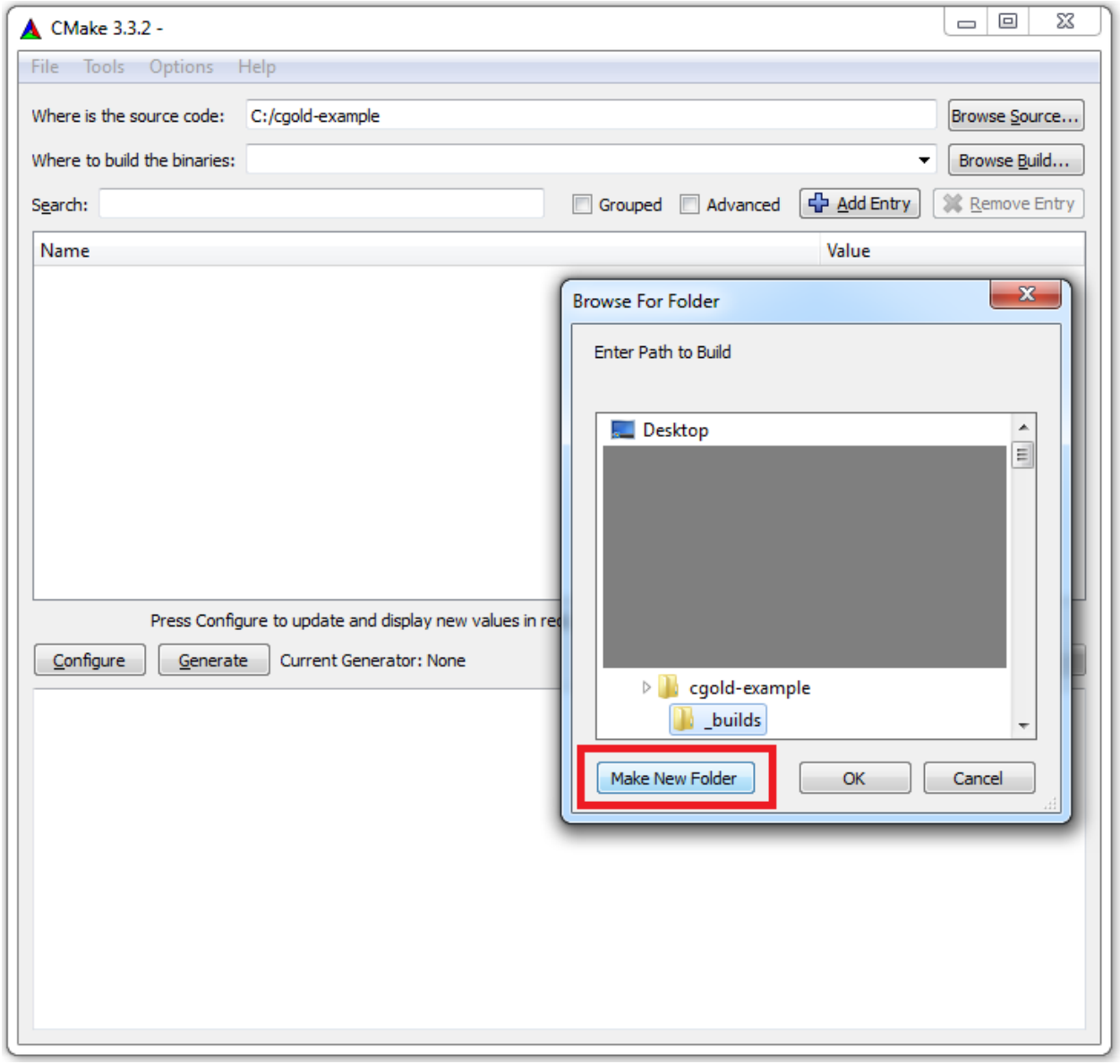
Click `Browse Source...` and find directory with `CMakeLists.txt` and `foo.cpp`:



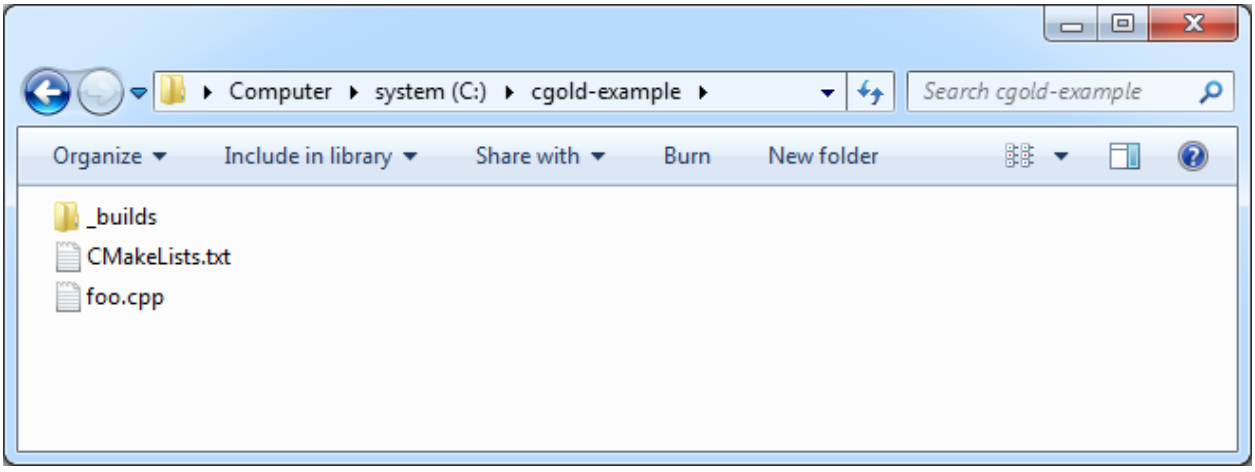
Now we need to choose directory where to put all temporary files. Let's create separate directory so we can keep our original directory clean. Click `Browse Build..`:



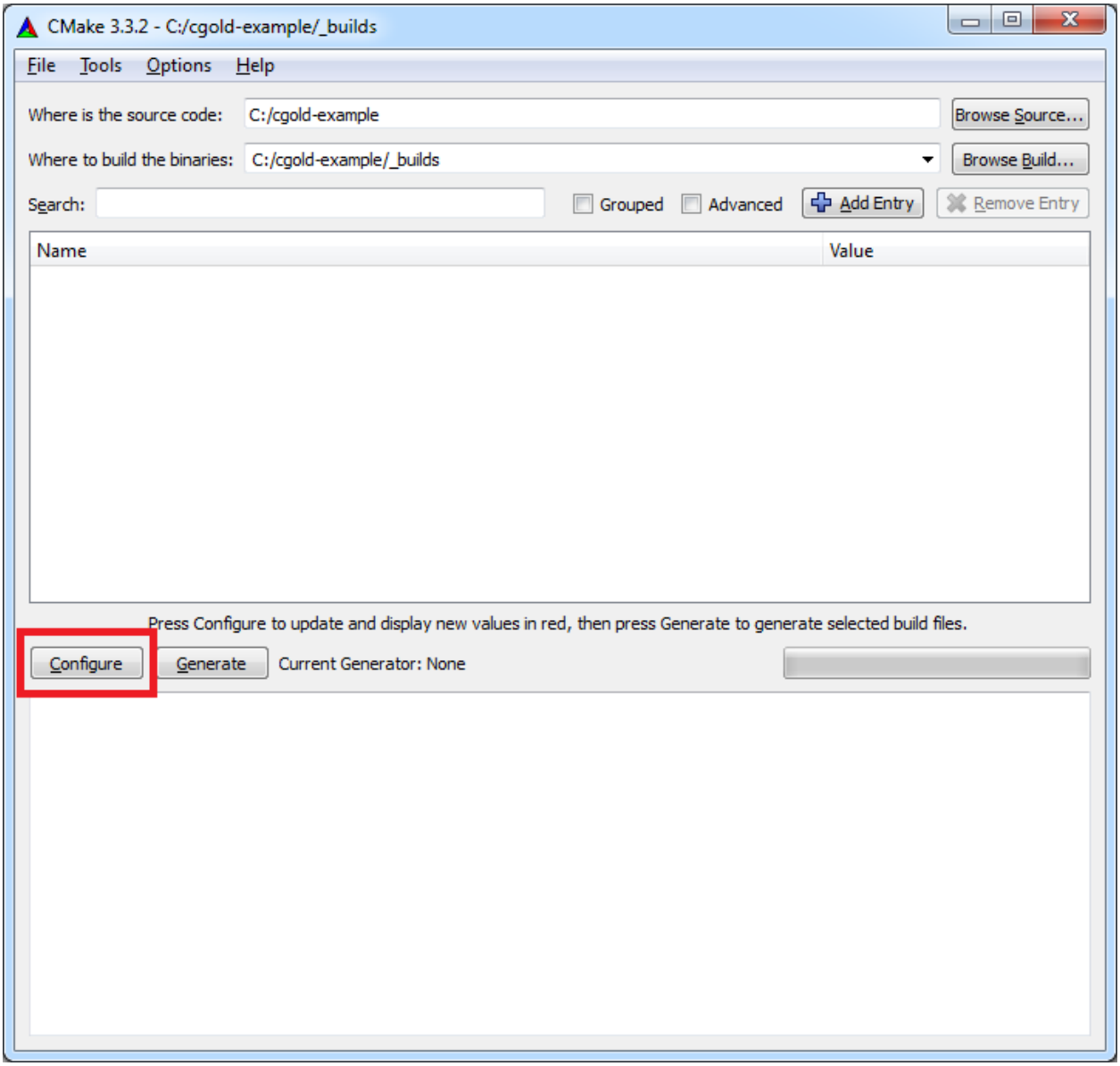
Find directory with `CMakeLists.txt` and click `Make New Folder` to create `_builds` directory:



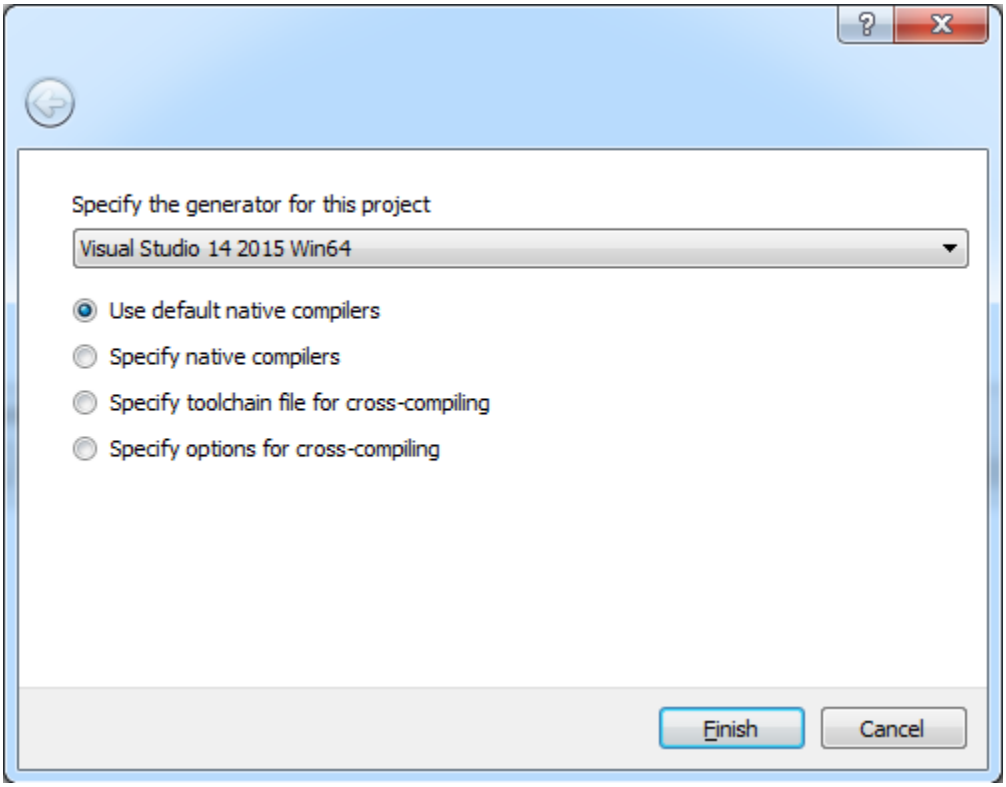
Check the resulted layout:



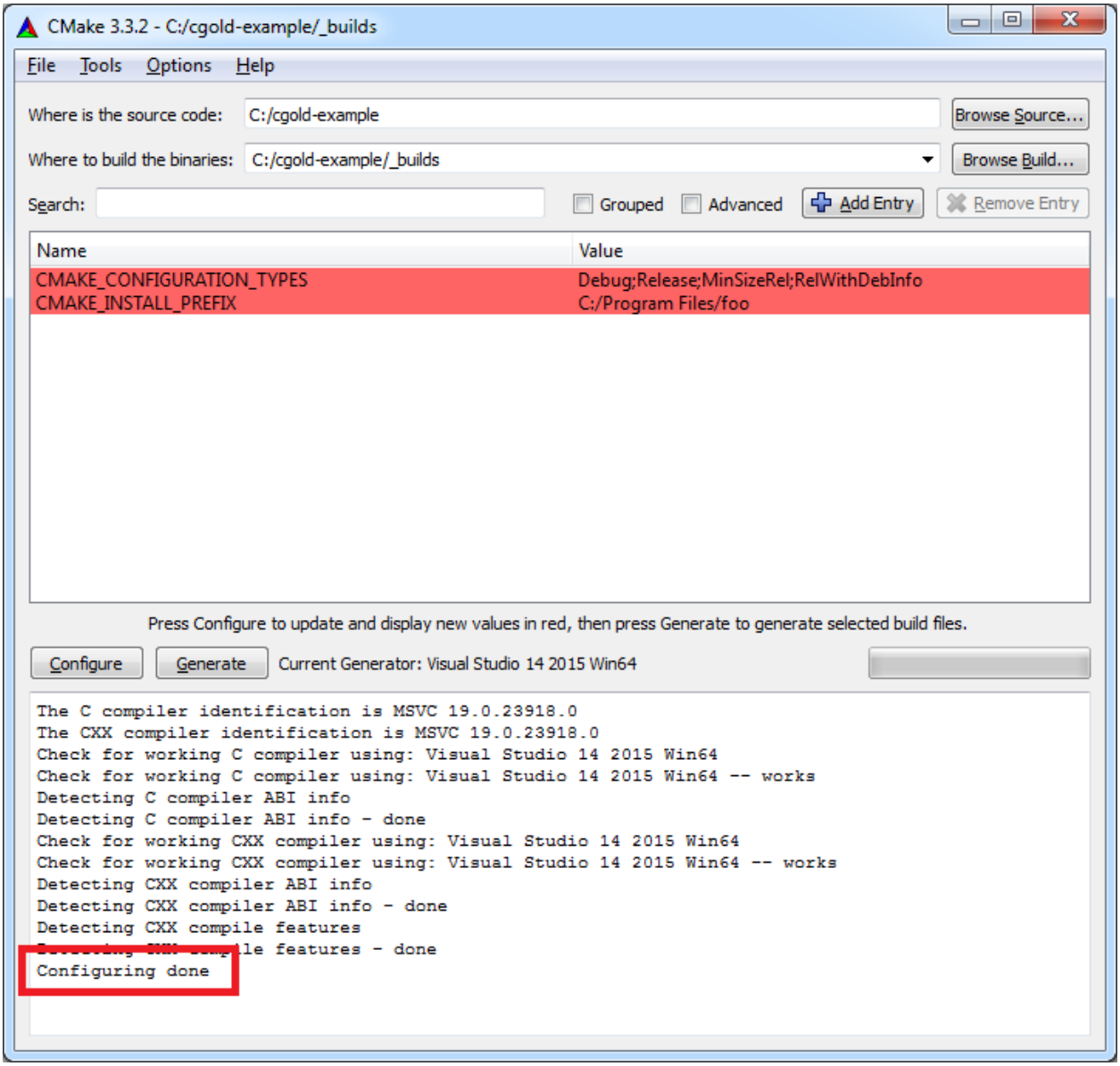
Click on `Configure` to process `CMakeLists.txt`:



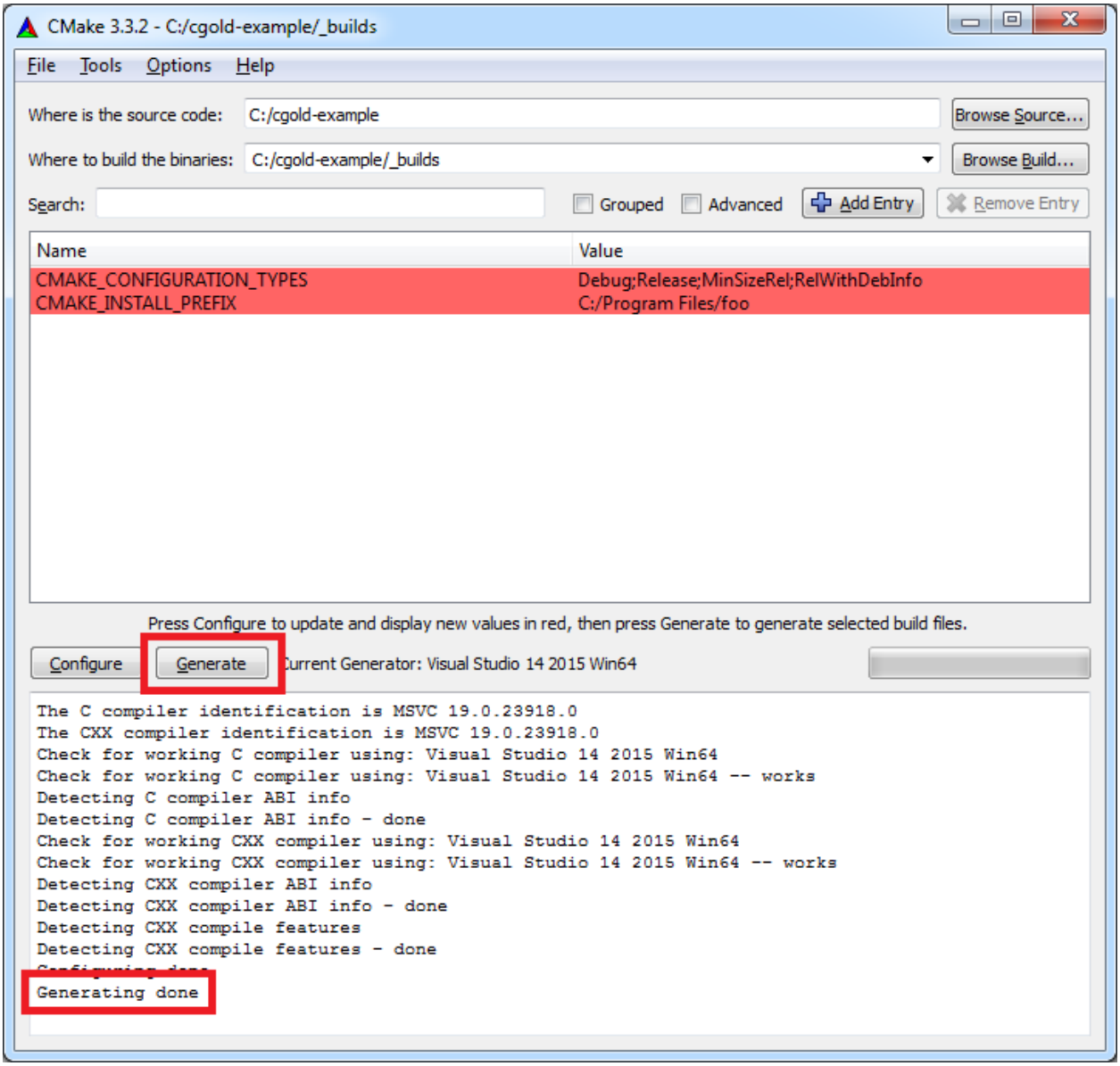
CMake will ask for the generator you want to use. Pick Visual Studio you have installed and add `Win64` to have x64 target:



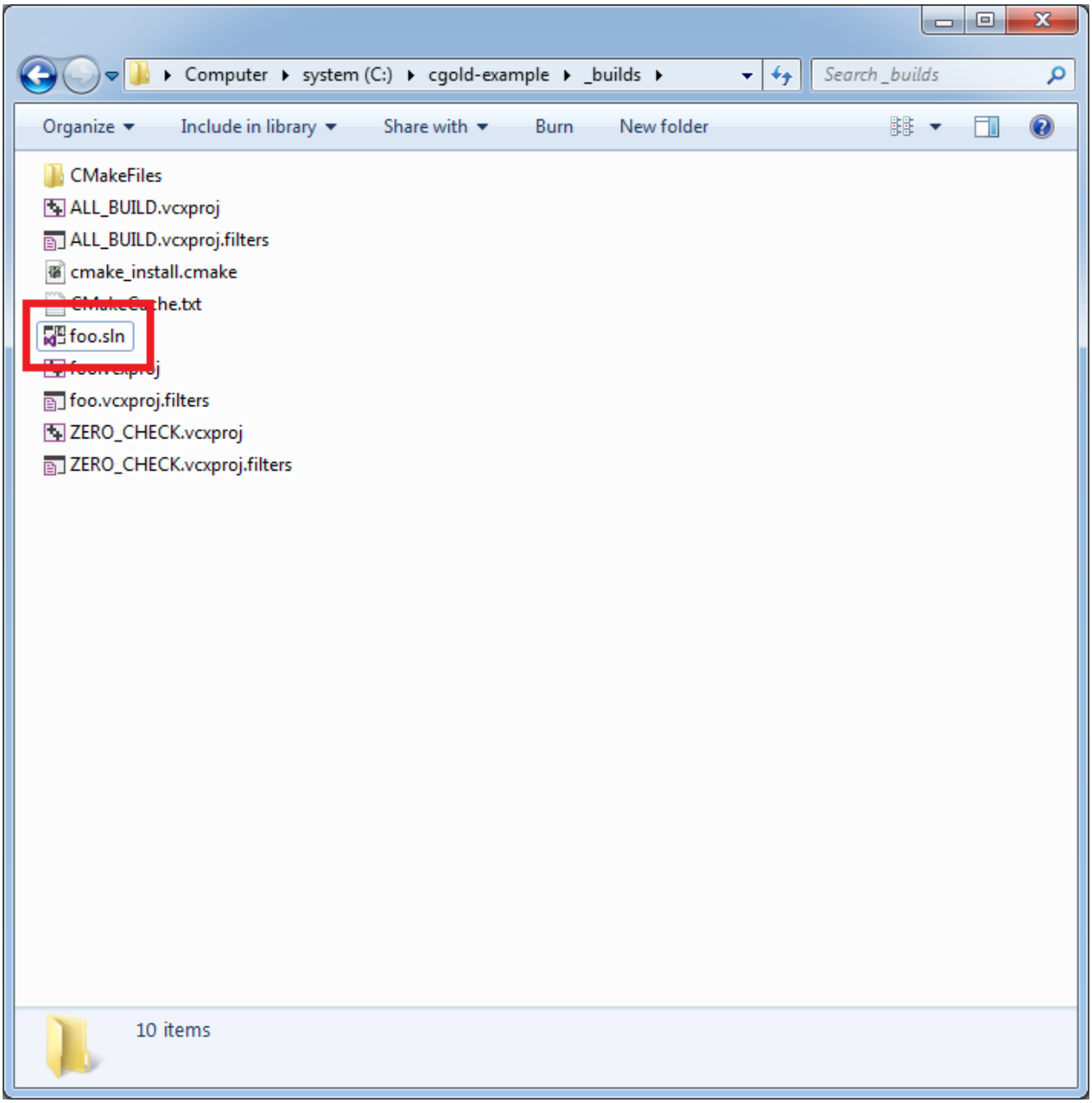
After you click `Finish` CMake will run internal tests on build tool to check that everything works correctly. You can see `Configuring done` message when finished:



For now there was no native build tool files generated, on this step user is able to do additional tuning of project. We don't want such tuning now so will run `Generate`:



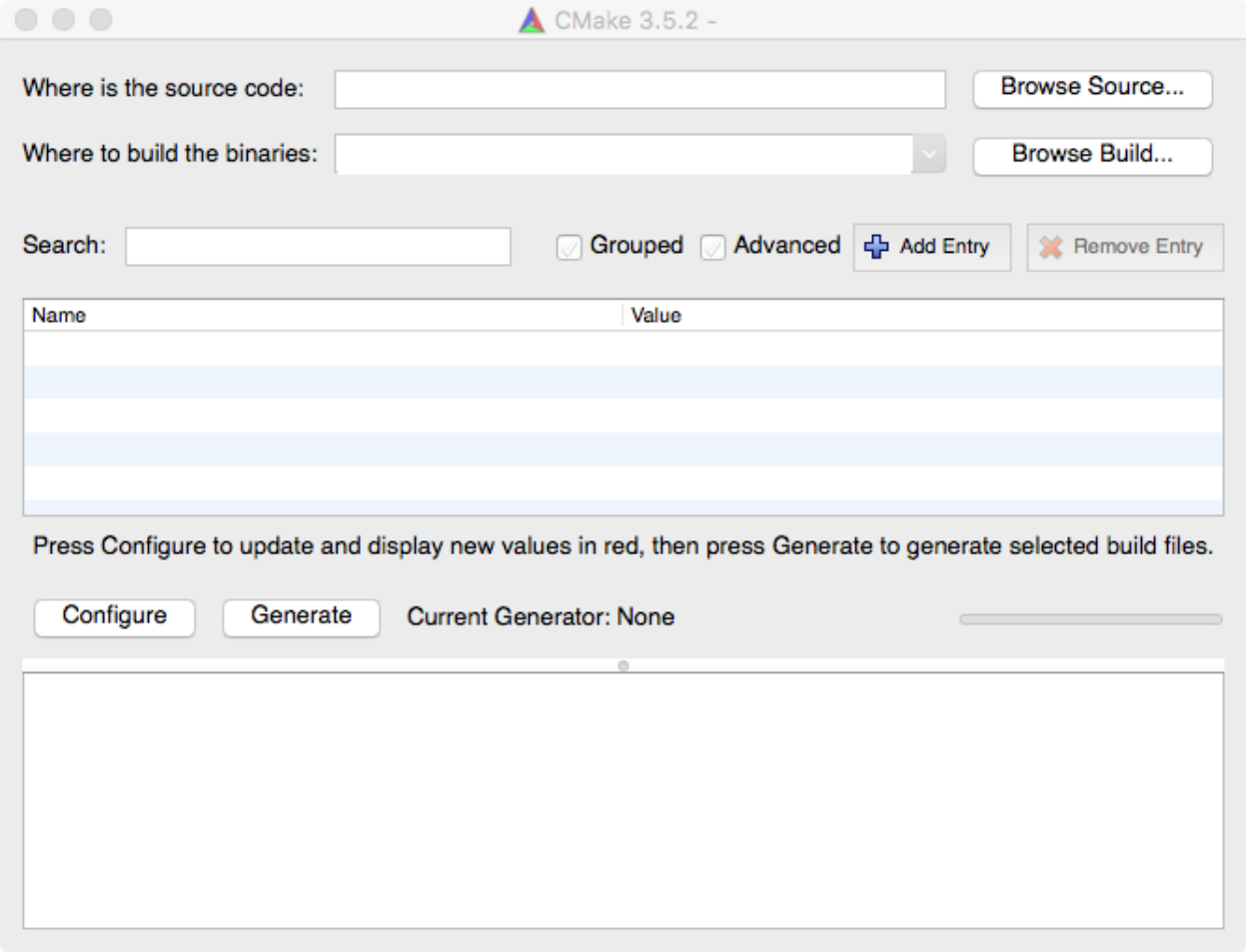
Now if you take a look at `_builds` folder you can find generated Visual Studio solution file:



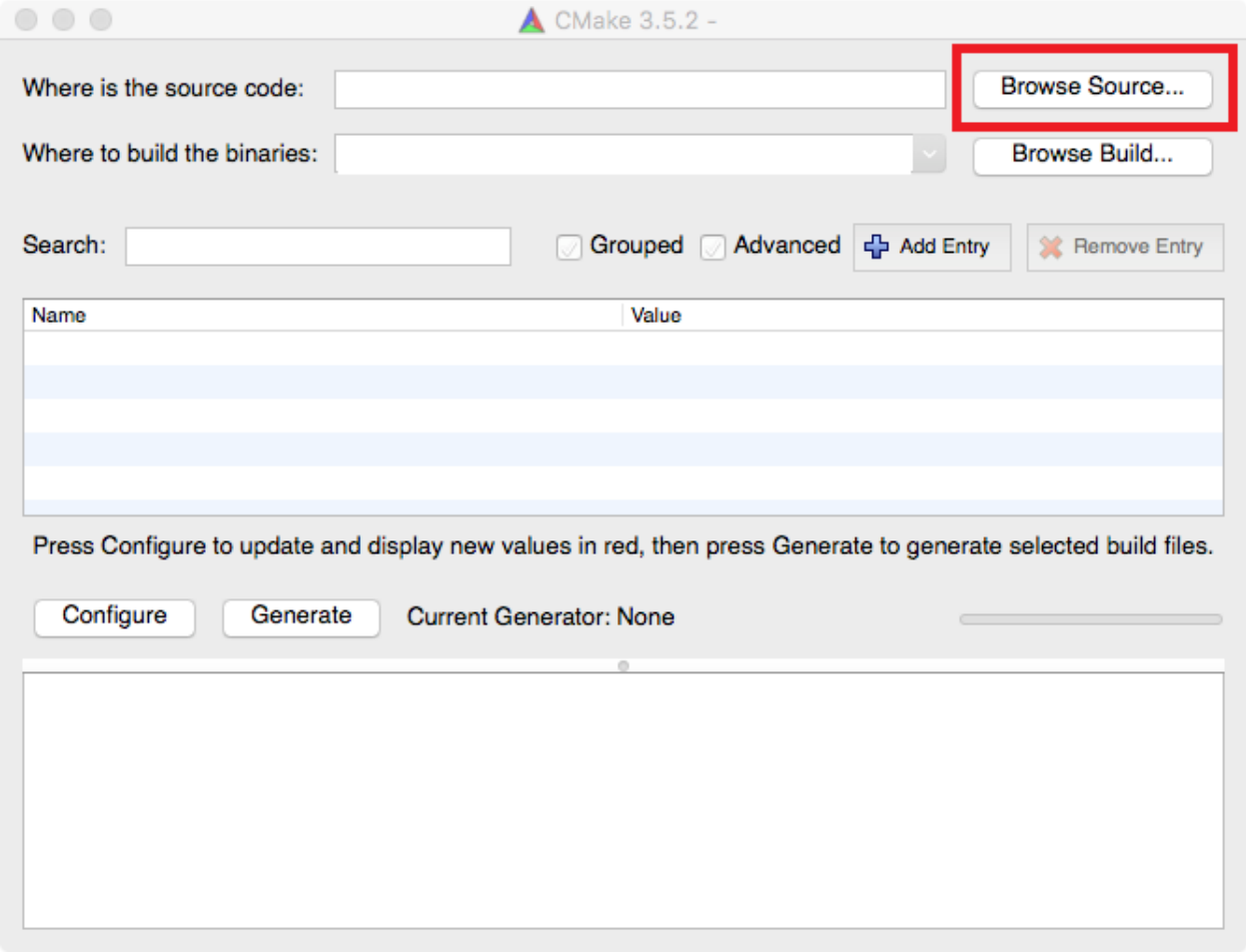
Open `foo.sln` and *run executable*.

2.5.2 GUI: Xcode

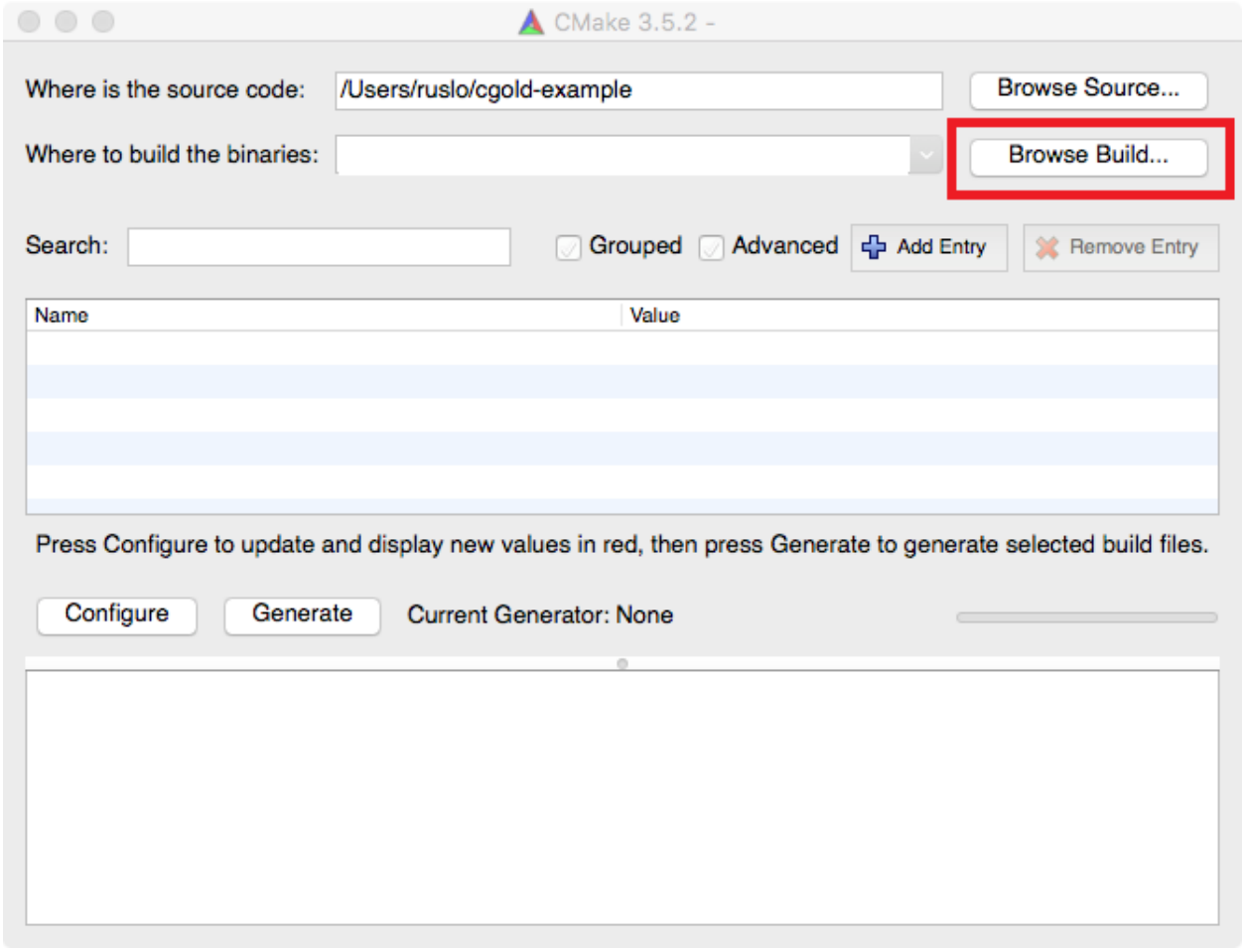
Open CMake GUI:



Click `Browse Source...` and find directory with `CMakeLists.txt` and `foo.cpp`:

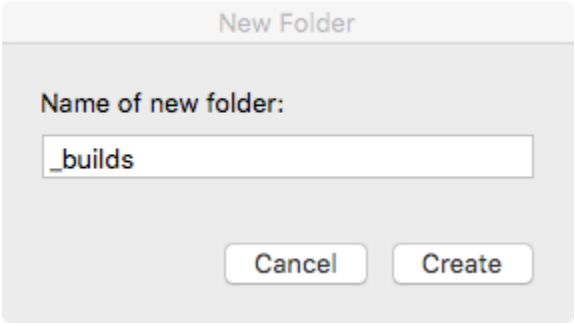


Now we need to choose directory where to put all temporary files. Let's create separate directory so we can keep our original directory clean. Click `Browse Build..`:

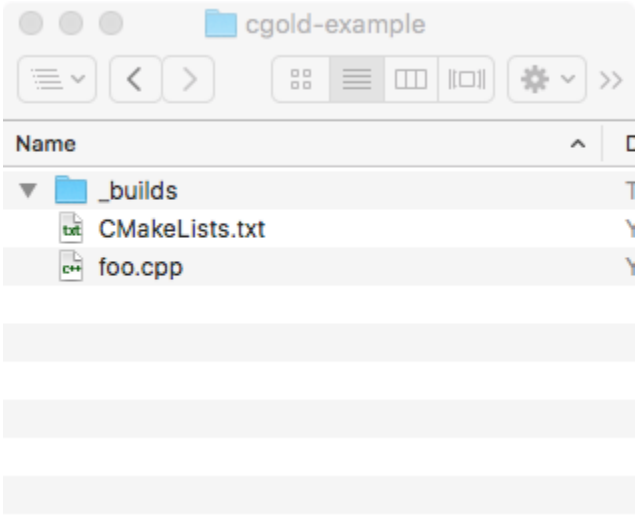


Find directory with `CMakeLists.txt` and click `New Folder` to create `_builds` directory:

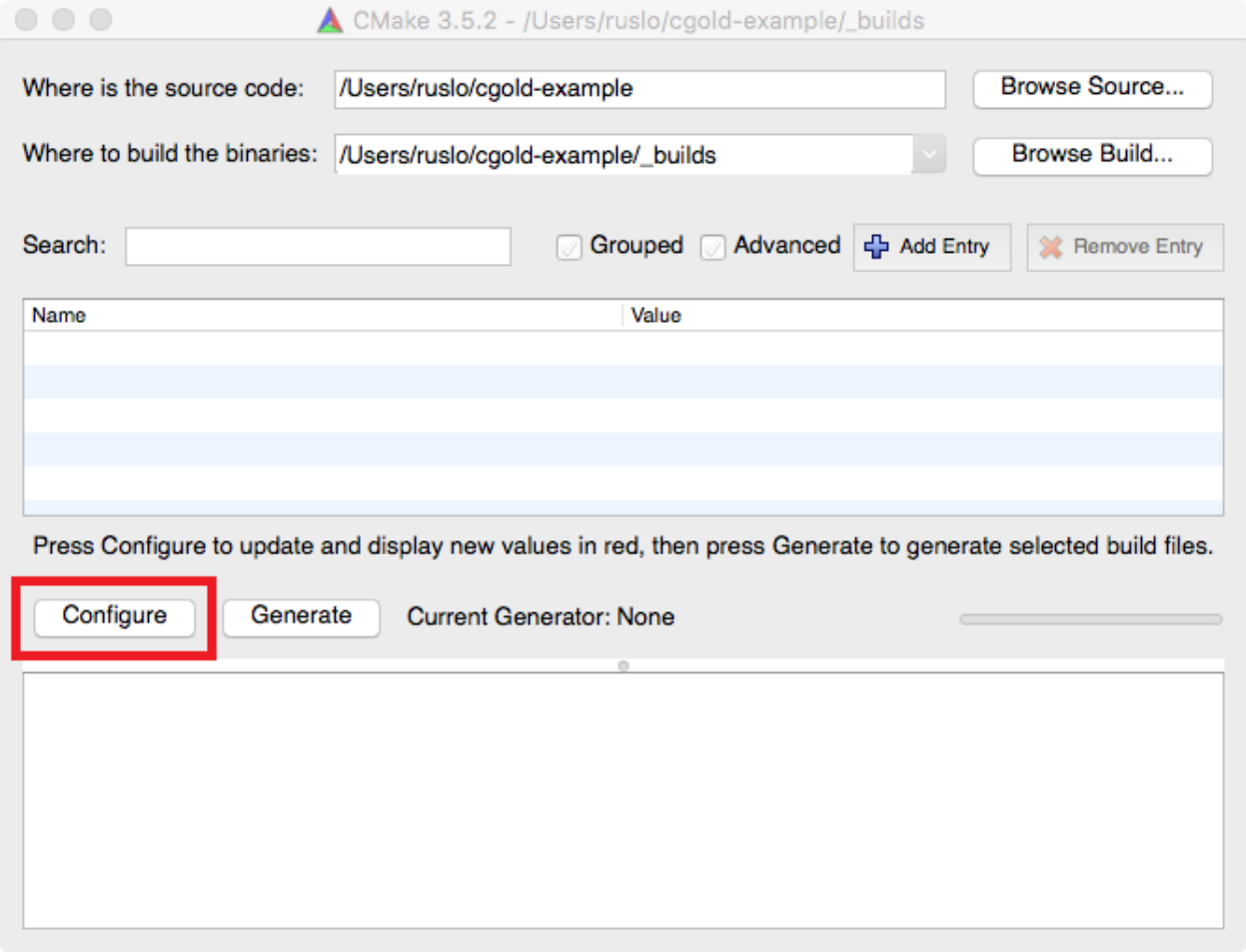
Enter `_builds` and click `Create`:



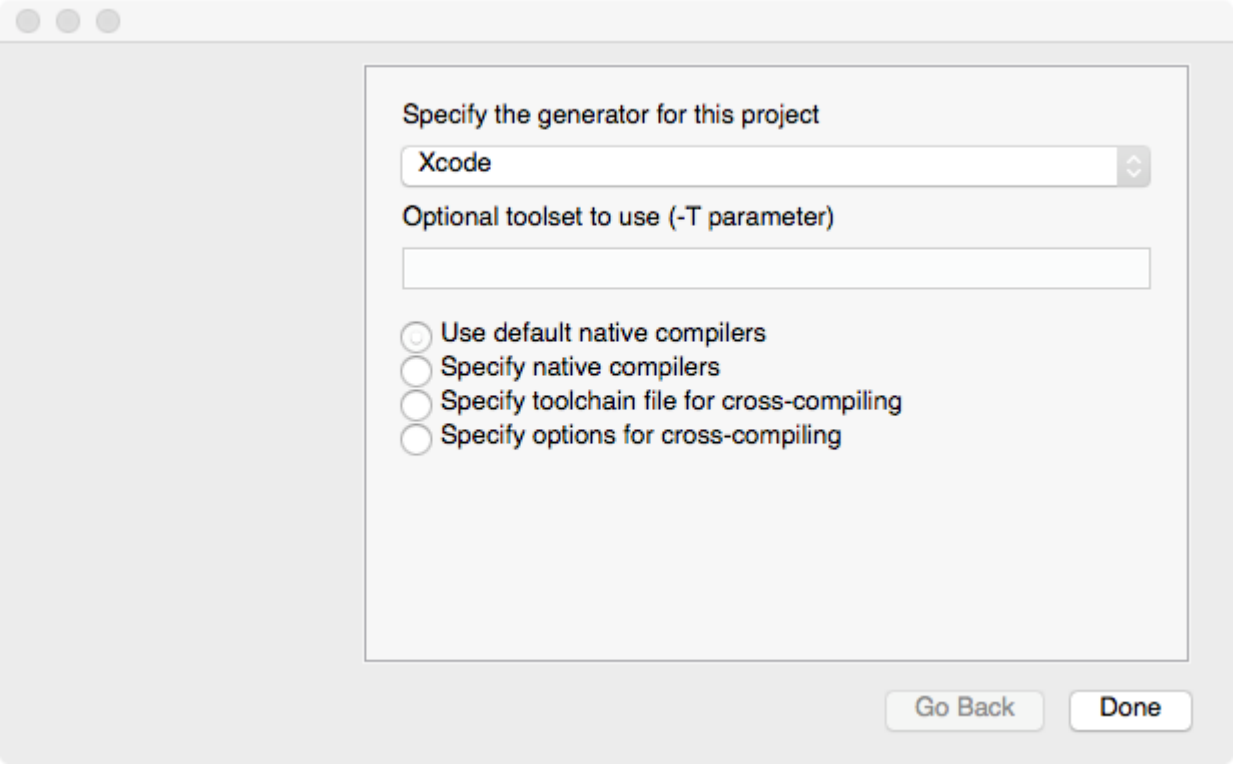
Check the resulted layout:



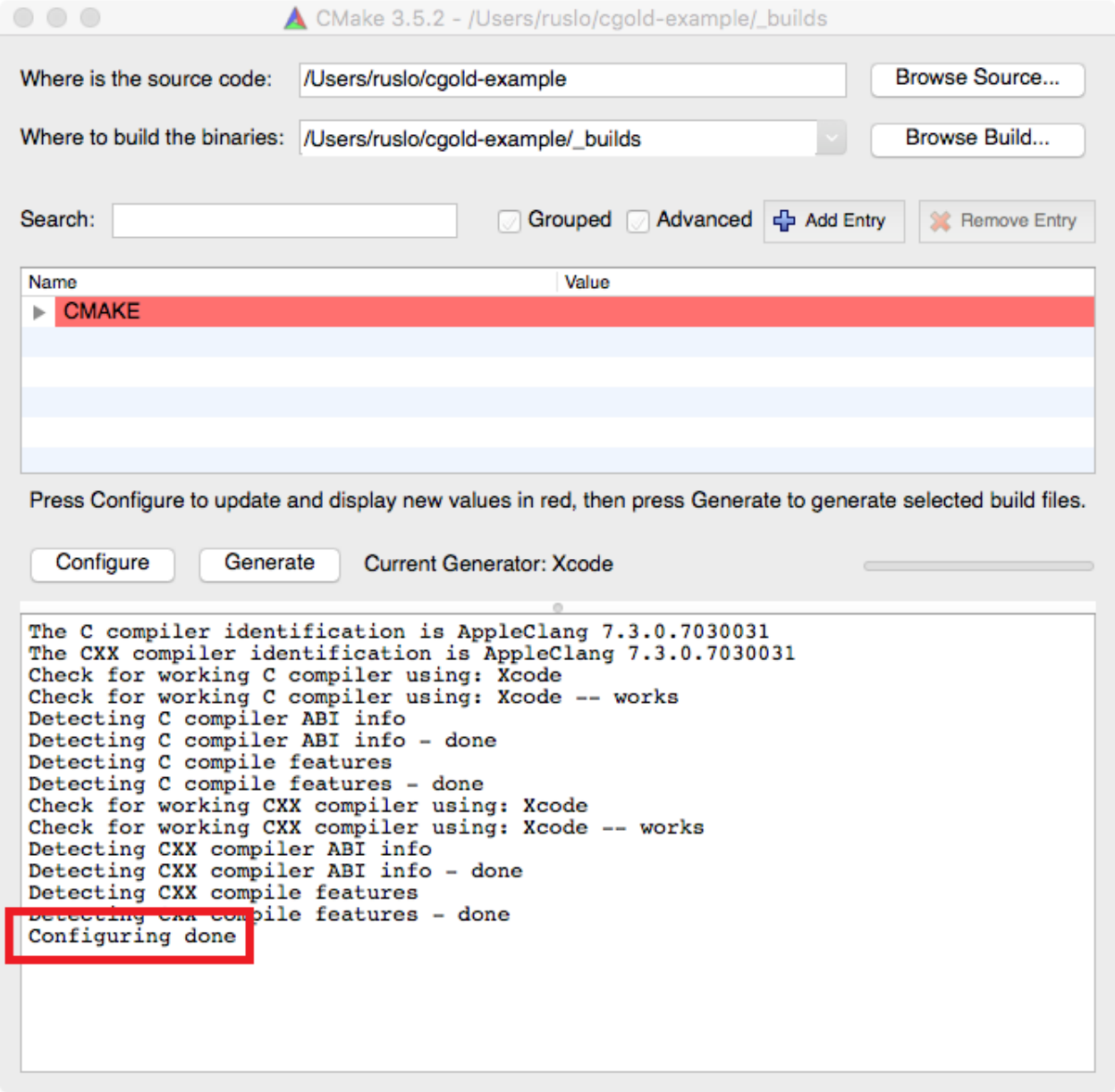
Click on `Configure` to process `CMakeLists.txt`:



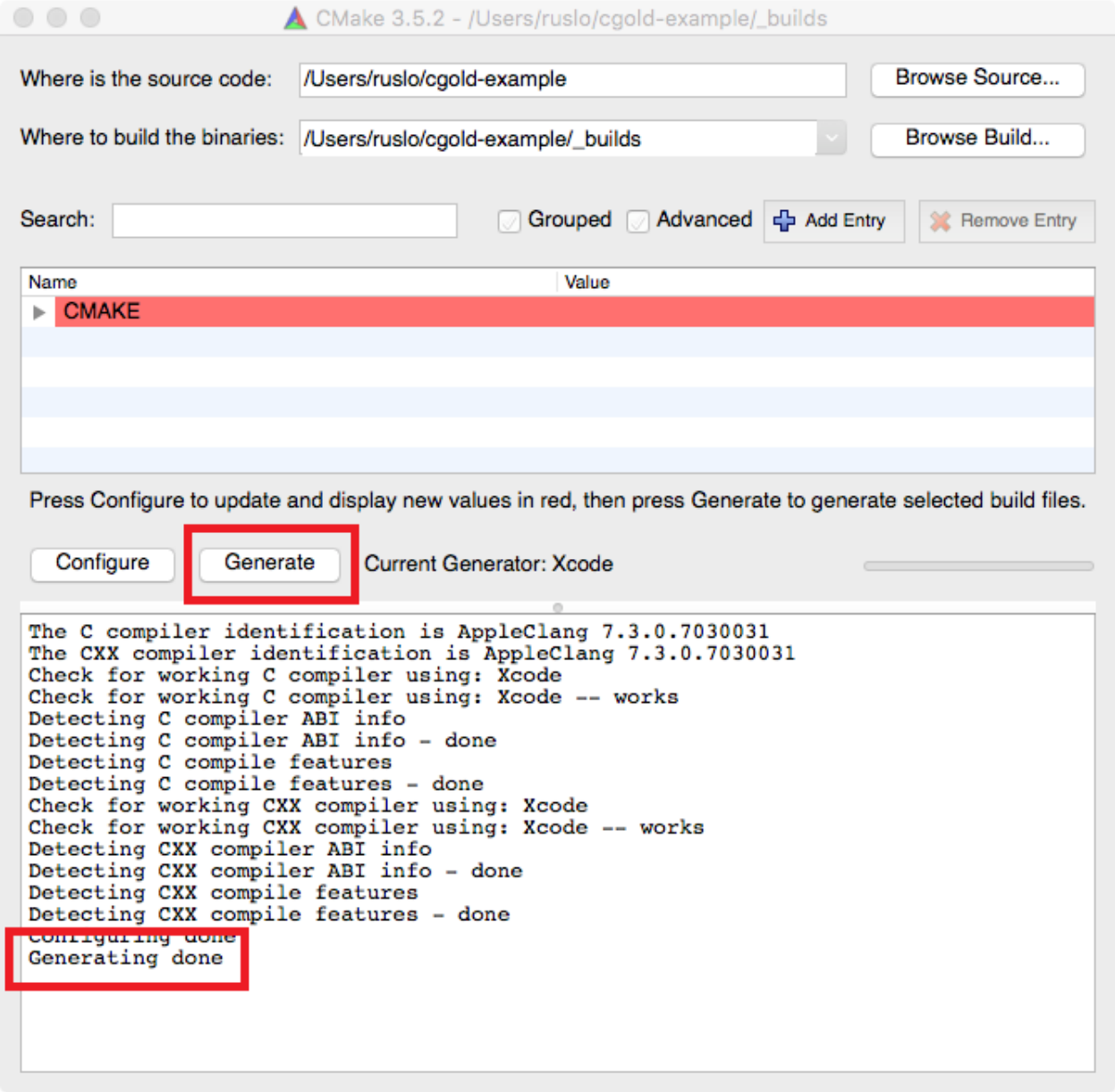
CMake will ask for the generator you want to use, pick Xcode:



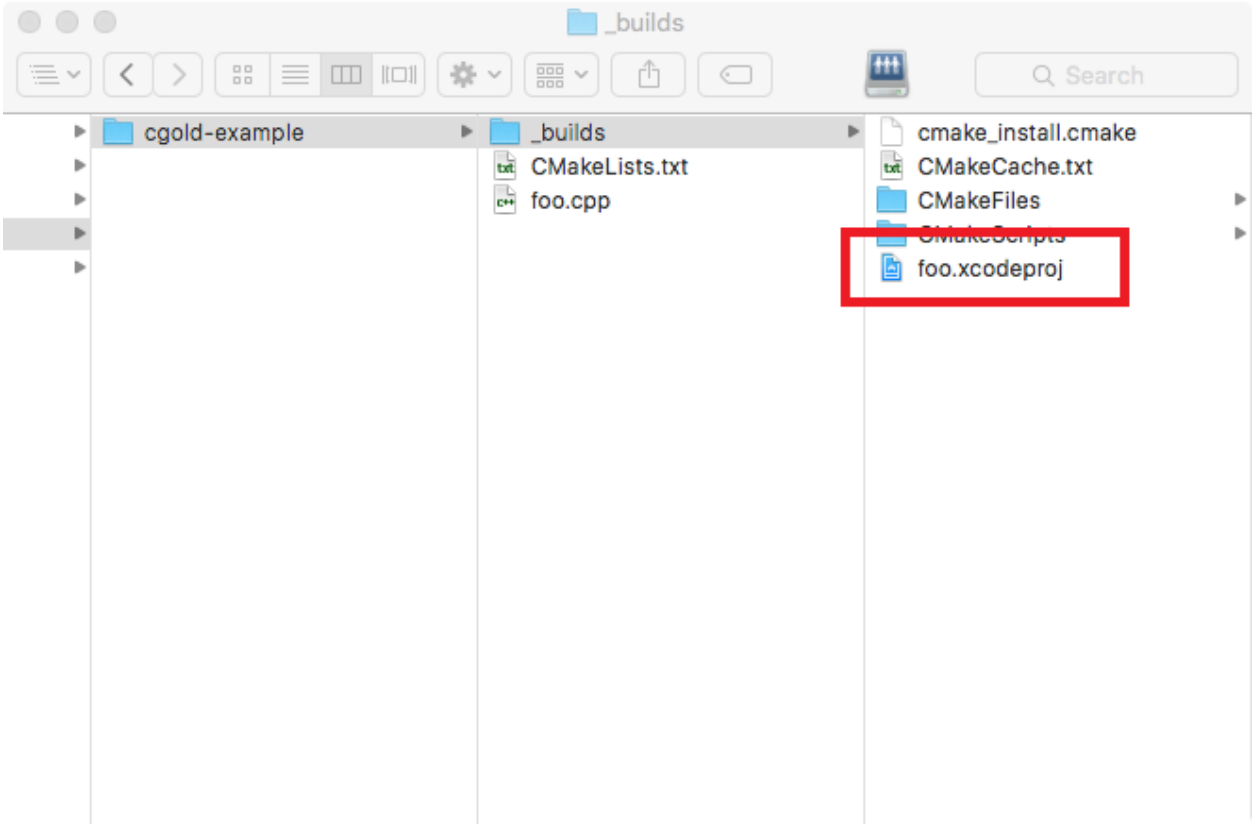
After you click `Done` CMake will run internal tests on build tool to check that everything works correctly. You can see `Configuring done` message when finished:



For now there was no native build tool files generated, on this step user is able to do additional tuning of the project. We don't want such tuning now so will run `Generate`:



Now if you take a look at `_builds` folder you can find generated Xcode project file:



Open `foo.xcodeproj` and *run executable*.

2.5.3 CLI: Visual Studio

Run `cmd.exe` and go to the directory with sources:

```
> cd C:\cgold-example

[cgold-example]> dir

... CMakeLists.txt
... foo.cpp
```

Generate Visual Studio solution using CMake. Use *-H. -B_builds* for specifying paths and `-G "Visual Studio 14 2015 Win64"` for the generator:

```
[cgold-example]> cmake -H. -B_builds -G "Visual Studio 14 2015 Win64"
-- The C compiler identification is MSVC 19.0.23918.0
-- The CXX compiler identification is MSVC 19.0.23918.0
-- Check for working C compiler using: Visual Studio 14 2015 Win64
-- Check for working C compiler using: Visual Studio 14 2015 Win64 -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler using: Visual Studio 14 2015 Win64
-- Check for working CXX compiler using: Visual Studio 14 2015 Win64 -- 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: C:/cgold-example/_builds
```

You can start IDE by `start _builds\foo.sln` and *run example from IDE* or *keep using command line*.

2.5.4 CLI: Xcode

Open terminal and go to the directory with sources:

```
> cd cgold-example
[cgold-example]> ls
CMakeLists.txt foo.cpp
```

Generate Xcode project using CMake. Use *-H. -B_builds* for specifying paths and `-GXcode` for the generator:

```
[cgold-example]> cmake -H. -B_builds -GXcode
-- The C compiler identification is AppleClang 7.3.0.7030031
-- The CXX compiler identification is AppleClang 7.3.0.7030031
-- Check for working C compiler: /.../Xcode.app/Contents/Developer/Toolchains/
 ↪XcodeDefault.xctoolchain/usr/bin/clang
-- Check for working C compiler: /.../Xcode.app/Contents/Developer/Toolchains/
 ↪XcodeDefault.xctoolchain/usr/bin/clang -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /.../Xcode.app/Contents/Developer/Toolchains/
 ↪XcodeDefault.xctoolchain/usr/bin/clang++
-- Check for working CXX compiler: /.../Xcode.app/Contents/Developer/Toolchains/
 ↪XcodeDefault.xctoolchain/usr/bin/clang++ -- 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: /Users/ruslo/cgold-example/_builds
```

You can start IDE by `open _builds/foo.xcodeproj` (add `-a` to set the version of Xcode you need: `open -a /Applications/develop/ide/xcode/6.4/Xcode.app _builds/foo.xcodeproj`) and *run example from IDE* or *keep using command line*.

2.5.5 CLI: Make

Instructions are the same for both `Linux` and `OSX`. Open terminal and go to the directory with sources:

```
> cd cgold-example
[cgold-example]> ls
CMakeLists.txt foo.cpp
```

Generate Makefile using CMake. Use *-H. -B_builds* for specifying paths and `-G "Unix Makefiles"` for the generator (note that `Unix Makefiles` is usually the default generator so `-G` probably not needed at all):

```
[cgold-example]> cmake -H. -B_builds -G "Unix Makefiles"
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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: /.../cgold-example/_builds
```

Generated `Makefile` can be found in `_builds` directory:

```
> ls _builds/Makefile
_builds/Makefile
```

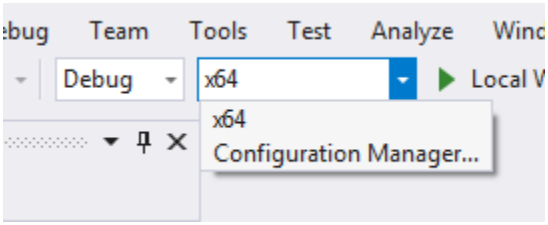
Next let's *build and run executable*.

2.6 Build and run executable

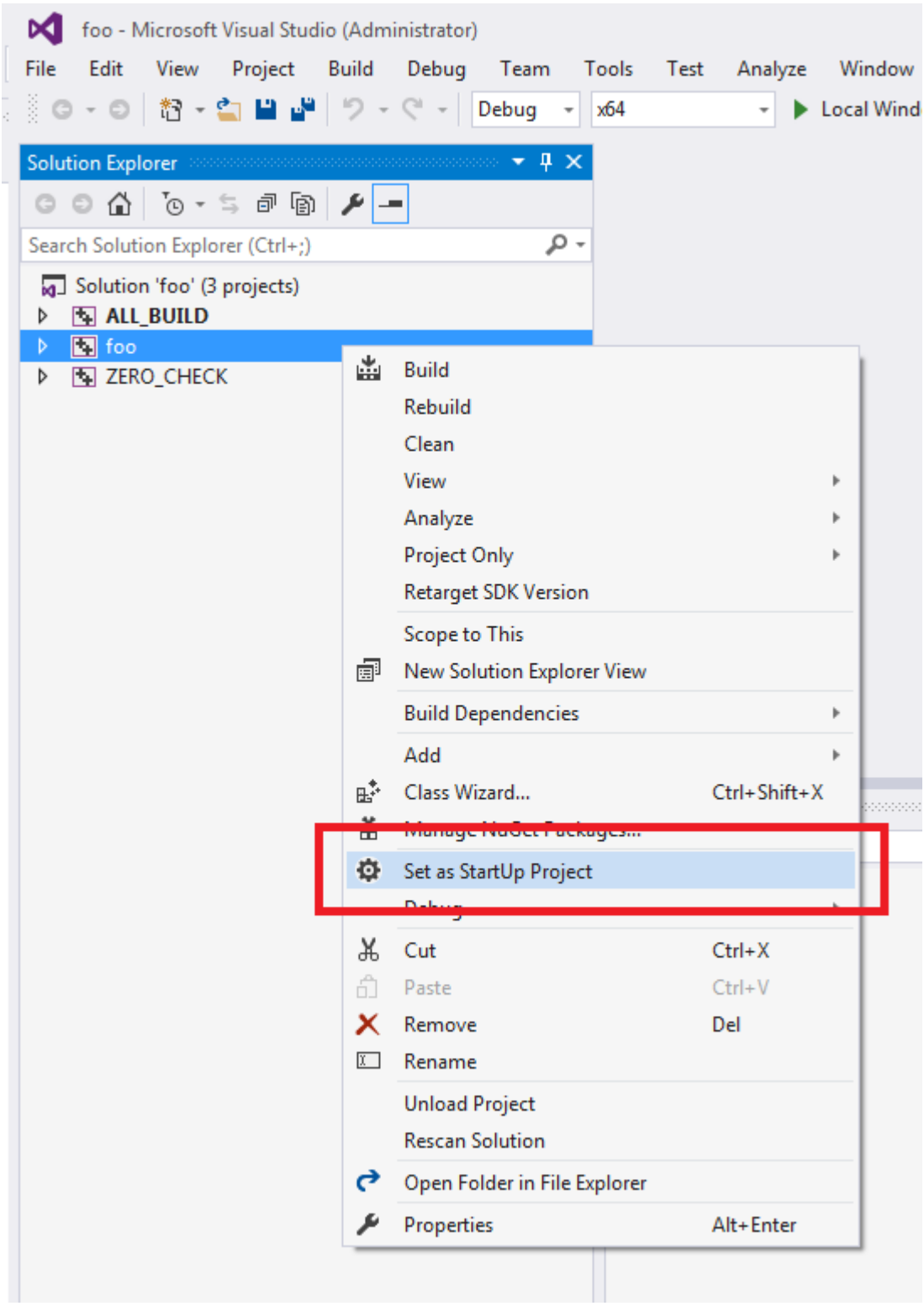
In this section we will build and run `foo` executable. In case if IDE was used you can do it by opening project in IDE or using command line (it doesn't matter how project was generated before: using GUI or CLI version of CMake).

2.6.1 IDE: Visual Studio

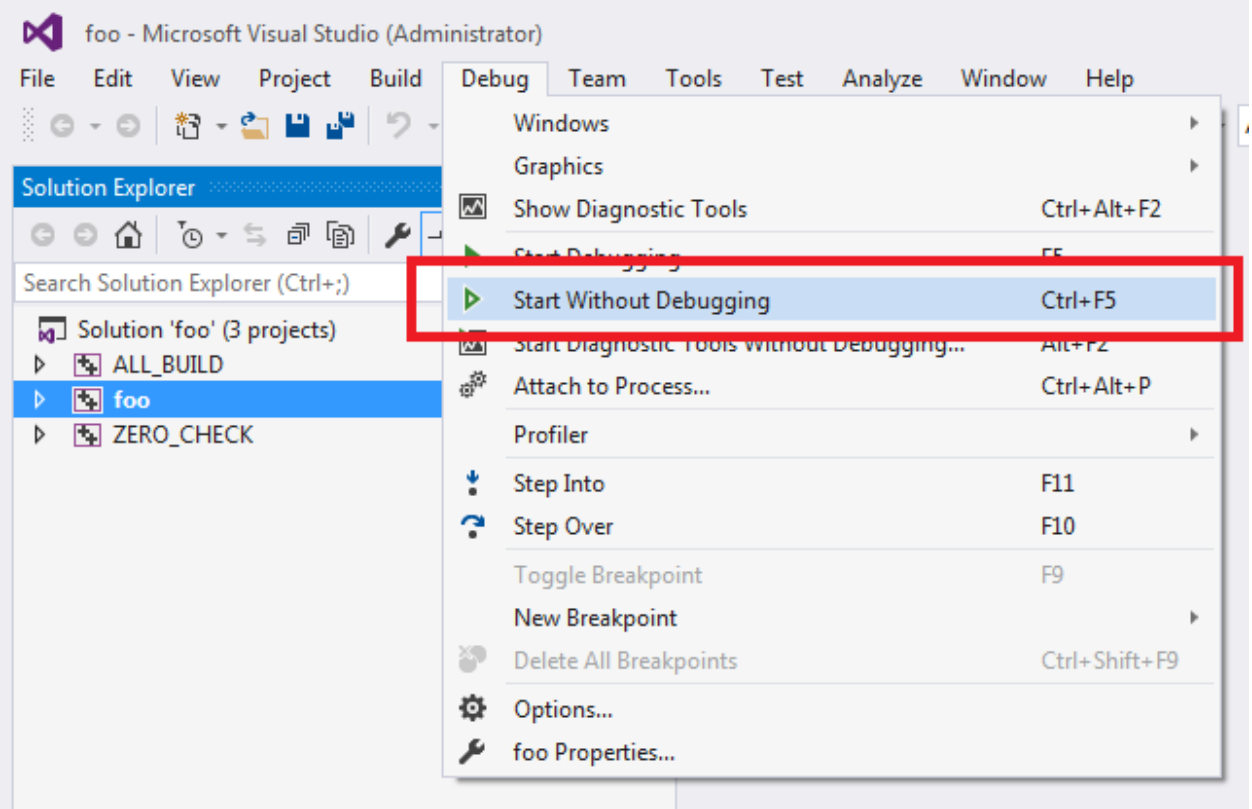
Since we used `* Win64` generator target's architecture is `x64`:



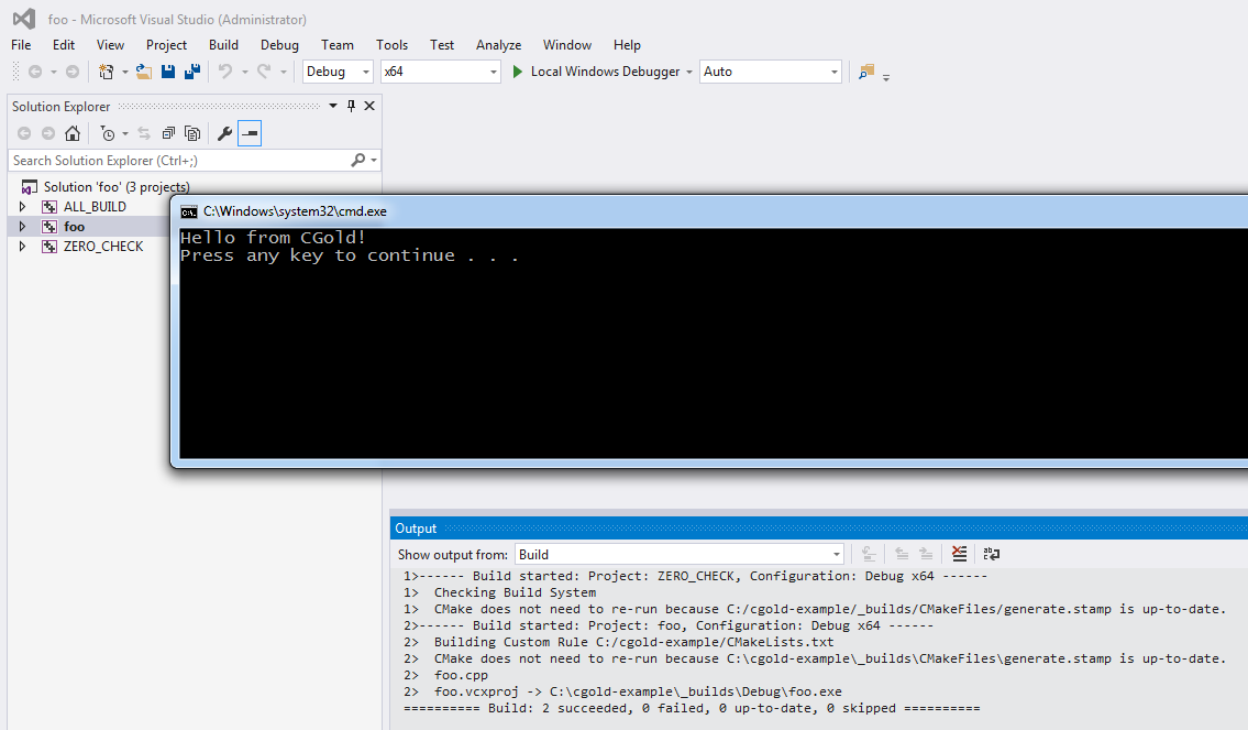
We need to tell Visual Studio that target we want to run is `foo`. This can be done by right clicking on `foo` target in `Solution Explorer` and choosing `Set as StartUp Project`:



To run executable go to *Debug* → *Start Without Debugging*:



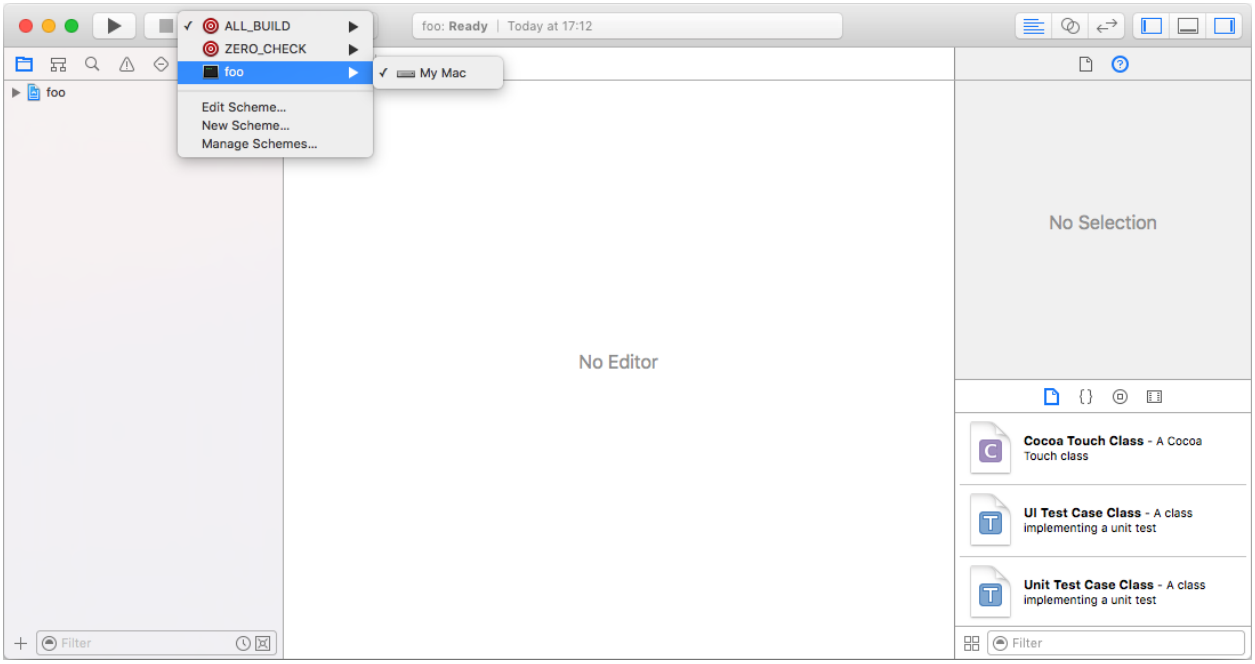
Visual Studio will build target first then execute it:



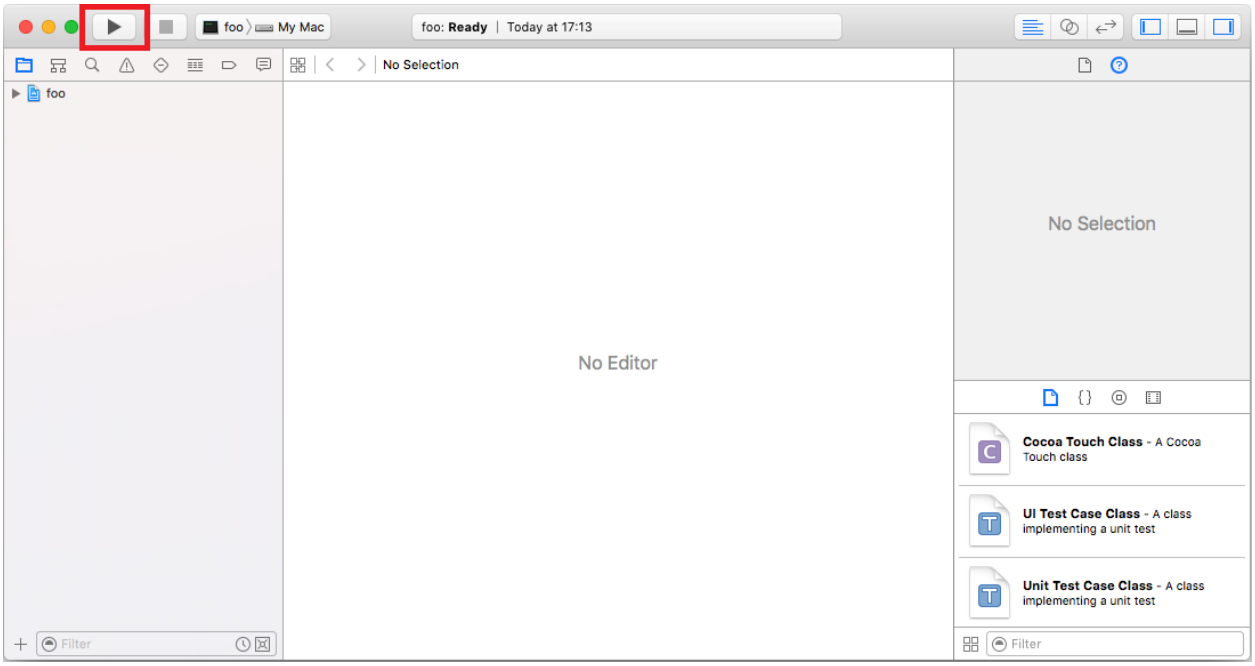
Done!

2.6.2 IDE: Xcode

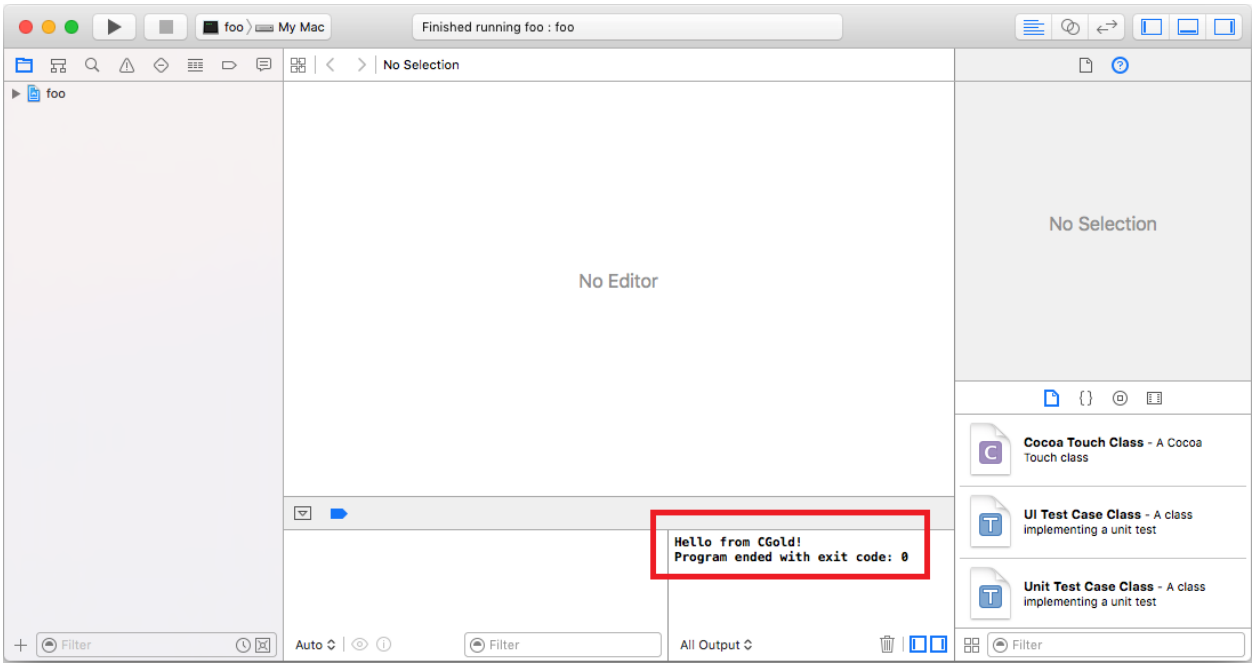
Choose the target you want to run:



Press `Run` button:



Result will be shown in `Debug area`:



Done!

2.6.3 CLI: Visual Studio

To build Visual Studio solution from command line `MSBuild.exe` can be used. You must add `MSBuild.exe` location to PATH or open Visual Studio Developer Prompt instead of `cmd.exe` (run `where msbuild` to check) and run `msbuild _builds\foo.sln`

But CMake offer cross-tool way to do exactly the same: `cmake --build _builds` (no need to have `MSBuild.exe` in PATH).

```
[cgold-example]> cmake --build _builds

...

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.54
```

By default Debug variant of `foo.exe` will be build, you can run it by:

```
[cgold-example]> .\_builds\Debug\foo.exe
Hello from CGold!
```

Done!

2.6.4 CLI: Xcode

To build Xcode project from command line `xcodebuild` can be used. Check it can be found:

```
> which xcodebuild
/usr/bin/xcodebuild
```

Go to the `_builds` directory and run build:

```
> cd _builds
[cgold-example/_builds]> xcodebuild
...

echo Build\ all\ projects
Build all projects

** BUILD SUCCEEDED **
```

But CMake offer cross-tool way to do exactly the same by `cmake --build _builds`:

```
[cgold-example]> cmake --build _builds
...

echo Build\ all\ projects
Build all projects

** BUILD SUCCEEDED **
```

By default Debug variant of `foo` will be build, you can run it by:

```
[cgold-example]> ./_builds/Debug/foo
Hello from CGold!
```

Done!

2.6.5 CLI: Make

Usually to build executable with Make you need to find directory with `Makefile` and run `make` in it:

```
> cd _builds
[cgold-example/_builds]> make
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX executable foo
[100%] Built target foo
```

But CMake offer cross-tool way to do exactly the same by `cmake --build _builds`:

```
[cgold-example]> cmake --build _builds
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX executable foo
[100%] Built target foo
```

Run `foo`:

```
[cgold-example]> ./_builds/foo
Hello from CGold!
```

Done!

CHAPTER 3

Tutorials

If you reached this section it means you can handle *basic configuration*. It's time to see everything in details and add more features.

Note: In provided examples:

- CMake will be run in command-line format but CMake-GUI will work in similar way, if behavior differs it will be noted explicitly
- For the host platform `Linux` is chosen, use analogous commands if you use another host. E.g. use `dir _builds` on Windows instead of `ls _builds`
- `Unix Makefiles` will be used as a generator. On *nix platforms this is default one. Peculiarities of other generators will be described explicitly

3.1 CMake stages

We start with a theory. Let's introduce some terminology about *CMake* commands we have executed *before*.

3.1.1 Configure step

On this step CMake will parse top level *CMakeLists.txt* of *source tree* and create *CMakeCache.txt* file with *cache variables*. Different types of variables will be described further in details. For CMake-GUI this step triggered by clicking on `Configure` button. For CMake command-line this step is combined with generate step so terms configure and generate will be used interchangeably. The end of this step expressed by `Configuring done` message from CMake.

GUI + Xcode example

Let's add message command to the example:

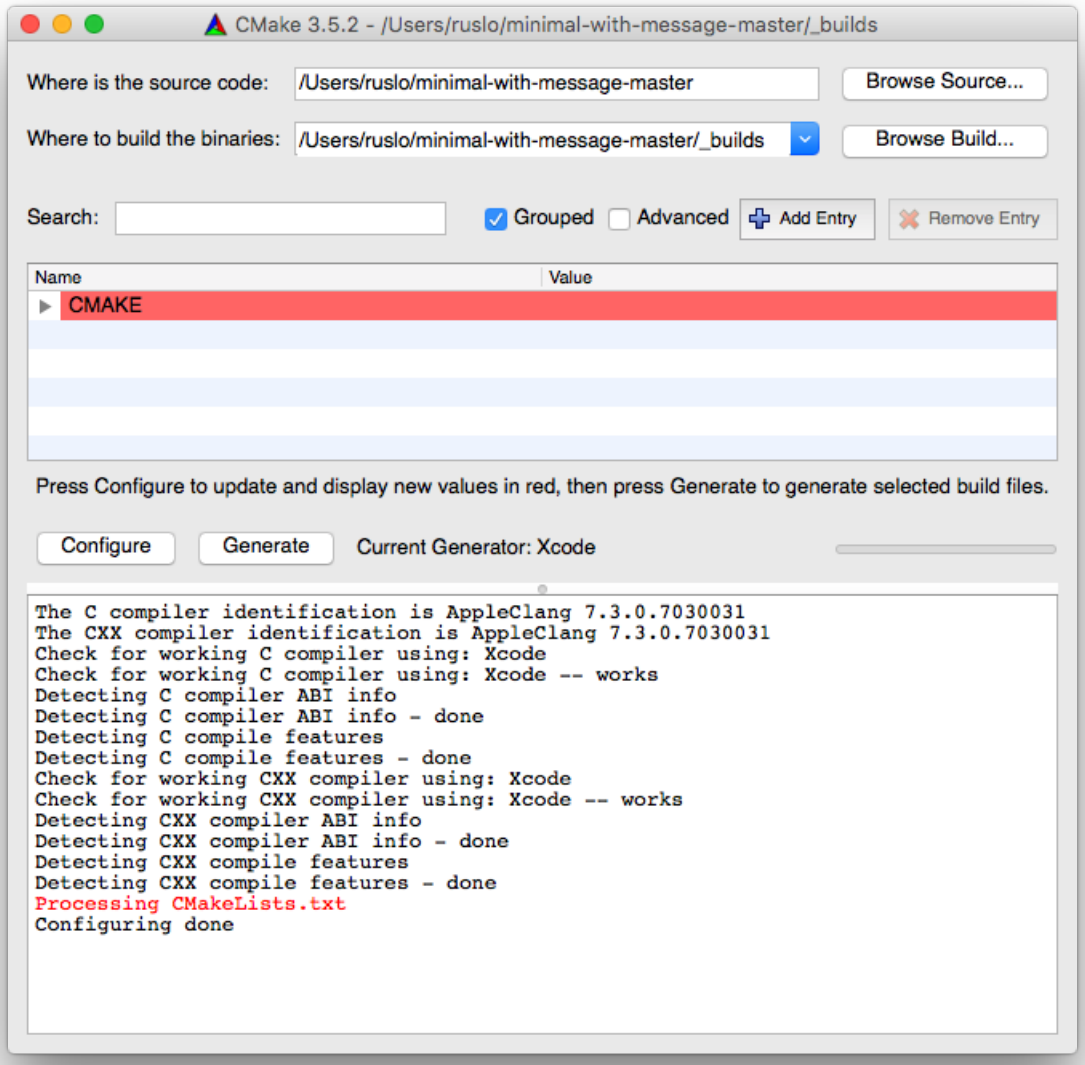
```
# CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_executable(foo foo.cpp)
```

```
message("Processing CMakeLists.txt")
```

Examples on GitHub

- Repository
- Latest ZIP

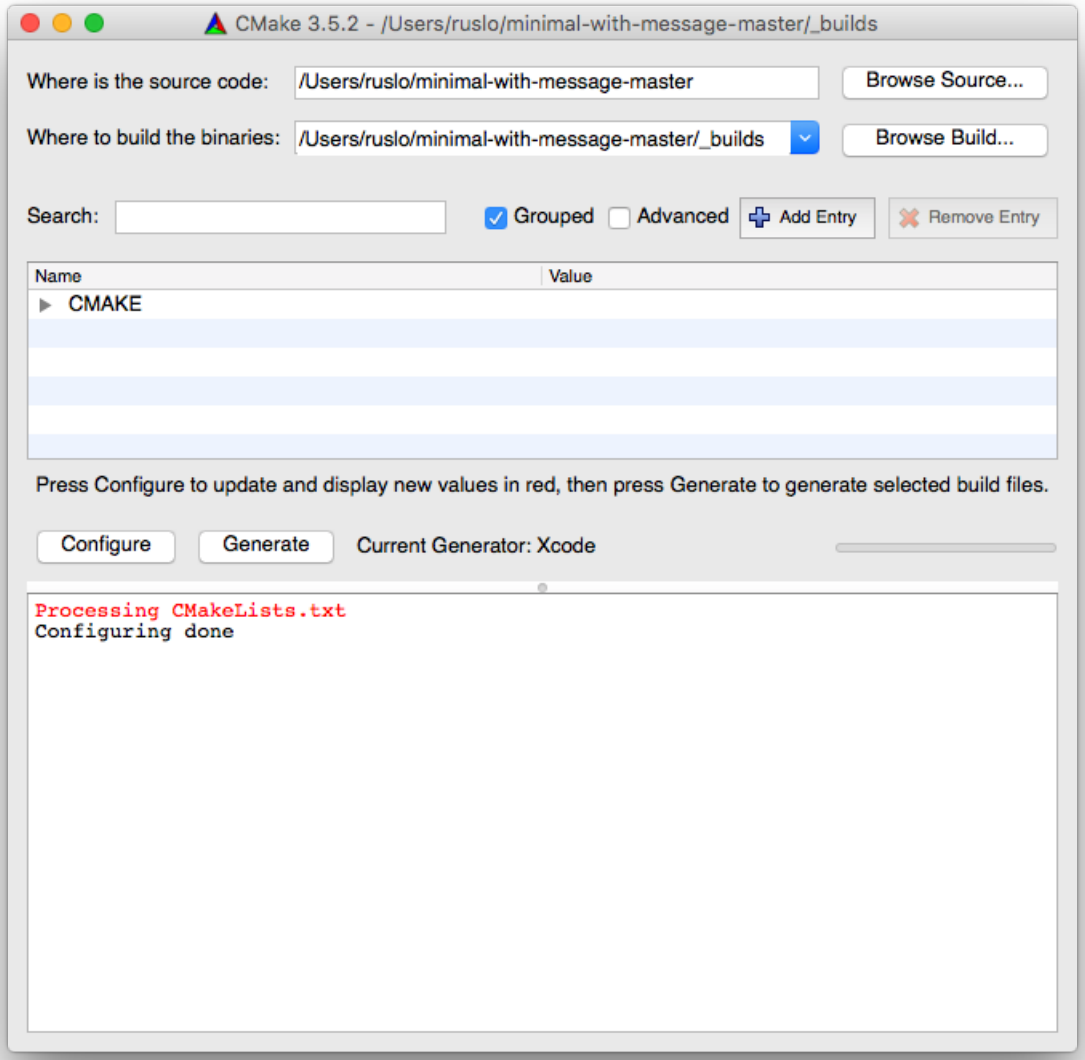
Line `Processing CMakeLists.txt` will be printed by CMake while parsing CMakeLists.txt file, i.e. on configure step. Open CMake-GUI, setup directories and hit `Configure`:



You can verify that there is no Xcode project generated by now but only CMakeCache.txt with cache variables:

```
[minimal-with-message-master]> ls _builds
CMakeCache.txt CMakeFiles/
```

Let's run configure one more time:



We still see `Process CMakeLists.txt` message which means that CMakeLists.txt was parsed again but there is no check/detect messages. This is because information about compiler and different tools detection results saved in CMake internal directories and reused. You may notice that second run happens much faster than first.

No surprises, there is still no Xcode project:

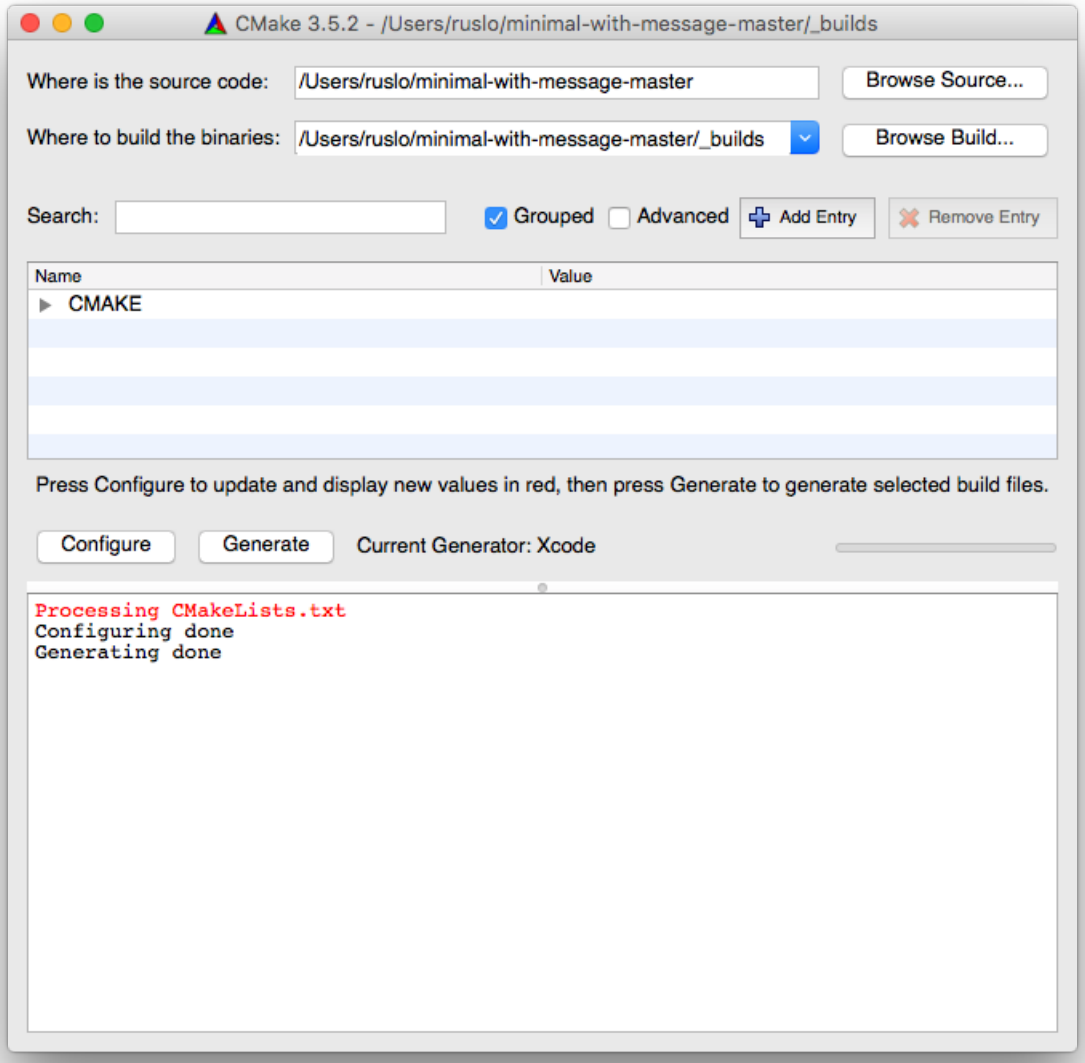
```
[minimal-with-message-master]> ls _builds
CMakeCache.txt CMakeFiles/
```

3.1.2 Generate step

On this step CMake will generate *native build tool* files using information from CMakeLists.txt and variables from CMakeCache.txt. For CMake-GUI this step triggered by clicking on `Generate` button. For CMake command-line this step is combined with configure step. The end of this step expressed by `Generating done` message from CMake.

GUI + Xcode example

Hit `Generate` now:



Now Xcode project created:

```
[minimal-with-message-master]> ls -d _builds/foo.xcodeproj
_builds/foo.xcodeproj/
```

Makefile example

Example of generating Makefile on Linux:

```
[minimal-with-message-master]> rm -rf _builds
[minimal-with-message-master]> cmake -H. -B_builds
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
Processing CMakeLists.txt
-- Configuring done
-- Generating done
-- Build files have been written to: /.../minimal-with-message-master/_builds
```

We see `Processing CMakeLists.txt`, `Configuring done` and `Generating done` messages meaning that CMakeLists.txt was parsed and both configure/generate steps combined into one action.

Verify Makefile generated:

```
[minimal-with-message-master]> ls _builds/Makefile
_builds/Makefile
```

If you run configure again CMakeLists.txt will be parsed one more time and `Processing CMakeLists.txt` will be issued:

```
[minimal-with-message-master]> cmake -H. -B_builds
Processing CMakeLists.txt
-- Configuring done
-- Generating done
-- Build files have been written to: /.../minimal-with-message-master/_builds
```

3.1.3 Build step

This step is orchestrated by native build tool. On this step targets of your project will be build.

Xcode example

Run native tool build:

```
[minimal-with-message-master]> cmake --build _builds

=== BUILD AGGREGATE TARGET ZERO_CHECK OF PROJECT foo WITH CONFIGURATION Debug ===

Check dependencies
```

```
...

=== BUILD TARGET foo OF PROJECT foo WITH CONFIGURATION Debug ===

...

    /.../Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/
↪clang -x c++ ...
        -c /.../minimal-with-message-master/foo.cpp
        -o /.../minimal-with-message-master/_builds/foo.build/Debug/foo.build/Objects-
↪normal/x86_64/foo.o

...

    /.../Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/
↪clang++ ...
        -o /Users/ruslo/minimal-with-message-master/_builds/Debug/foo

=== BUILD AGGREGATE TARGET ALL_BUILD OF PROJECT foo WITH CONFIGURATION Debug ===

...

Build all projects

** BUILD SUCCEEDED **
```

You can see that `foo.cpp` compiled into `foo.o` and then executable `foo` created. There is no `Processing CMakeLists.txt` message in output because on this stage CMake doesn't parse CMakeLists.txt however reconfigure may be triggered on build step automatically, this will be shown in *workflow* section.

Makefile example

Run native tool build:

```
[minimal-with-message-master]> cmake --build _builds
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX executable foo
[100%] Built target foo
```

You can see that `foo.cpp` compiled into `foo.cpp.o` and then executable `foo` created. There is no `Processing CMakeLists.txt` message in output because on this stage CMake doesn't parse CMakeLists.txt however reconfigure may be triggered on build step automatically, this will be shown in *workflow* section.

3.2 Out-of-source build

The next important term is "out-of-source build". "Out-of-source build" is a good practice of keeping separately generated files from *binary tree* and source files from *source tree*. CMake do support contrary "in-source build" layout but such approach has no real benefits and unrecommended.

3.2.1 Multiple configurations

Out-of-source build allow you to have different configurations simultaneously without conflicts, e.g. Debug and Release variant:

```
> cmake -H. -B_builds/Debug -DCMAKE_BUILD_TYPE=Debug
> cmake -H. -B_builds/Release -DCMAKE_BUILD_TYPE=Release
```

or any other kind of customization, e.g. options:

```
> cmake -H. -B_builds/feature-on -DFOO_FEATURE=ON
> cmake -H. -B_builds/feature-off -DFOO_FEATURE=OFF
```

generators:

```
> cmake -H. -B_builds/xcode -G Xcode
> cmake -H. -B_builds/make -G "Unix Makefiles"
```

platforms:

```
> cmake -H. -B_builds/osx -G Xcode
> cmake -H. -B_builds/ios -G Xcode -DCMAKE_TOOLCHAIN_FILE=/.../ios.cmake
```

3.2.2 VCS friendly

Out-of-source build allow you to ignore temporary binaries by just adding `_builds` directory to the no-tracking-files list:

```
# .gitignore

_builds
```

compare it with similar file for in-source build:

```
# .gitignore

*.sln
*.vcxproj
*.vcxproj.filters
*.xcodeproj
CMakeCache.txt
CMakeFiles
CMakeScripts
Debug/*
Makefile
Win32/*
cmake_install.cmake
foo
foo.build/*
foo.dir/*
foo.exe
x64/*
```

3.2.3 Other notes

In-source build at the first glance may looks more friendly for the developers who used to store projects/solution files in *VCS*. But in fact out-of-source build will remind you one more time that now your workflow changed, CMake is in charge and *you should not* edit your project settings in IDE.

Another note is that out-of-source mean not only set `cmake -B_builds` but also remember to put any kind of automatically generated files to `_builds`. E.g. if you have C++ template `myproject.h.in` which is used to generate `myproject.h`, then you need to keep `myproject.h.in` in source tree and put `myproject.h` to the binary tree.

3.3 Workflow

There is a nice feature in *CMake* that can greatly simplify developer's workflow: *native build tool* will watch CMake sources for changes and run re-configure step on update automatically. In command-line terms it means that you have to run `cmake -H. -B_builds` **only once**, you don't need to run configure again after modification of CMakeLists.txt - you can keep using `cmake --build`.

3.3.1 Makefile example

Back to the example with `message`:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_executable(foo foo.cpp)

message("Processing CMakeLists.txt")
```

Examples on GitHub

- Repository
- Latest ZIP

Generate Makefile:

```
[minimal-with-message]> cmake -H. -B_builds
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
```

```
-- Detecting CXX compile features - done
Processing CMakeLists.txt
-- Configuring done
-- Generating done
-- Build files have been written to: /.../minimal-with-message/_builds
```

And run build:

```
[minimal-with-message]> cmake --build _builds
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX executable foo
[100%] Built target foo
```

Executable `foo` created from `foo.cpp` source. Make tool knows that if there are no changes in `foo.cpp` then no need to build and link executable. If you run build again there will be no compile and link stage:

```
[minimal-with-message]> cmake --build _builds
[100%] Built target foo
```

Let's "modify" `foo.cpp` source:

```
[minimal-with-message]> touch foo.cpp
[minimal-with-message]> cmake --build _builds
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX executable foo
[100%] Built target foo
```

Make detects that executable `foo` is out-of-date and rebuild it. Well, that's what build systems designed for :)

Now let's "change" CMakeLists.txt. Do we need to run `cmake -H. -B_builds` again? The answer is NO - just keep using `cmake --build _builds`. CMakeLists.txt added as dependent file to the Makefile:

```
[minimal-with-message]> touch CMakeLists.txt
[minimal-with-message]> cmake --build _builds
Processing CMakeLists.txt
-- Configuring done
-- Generating done
-- Build files have been written to: /.../minimal-with-message/_builds
[100%] Built target foo
```

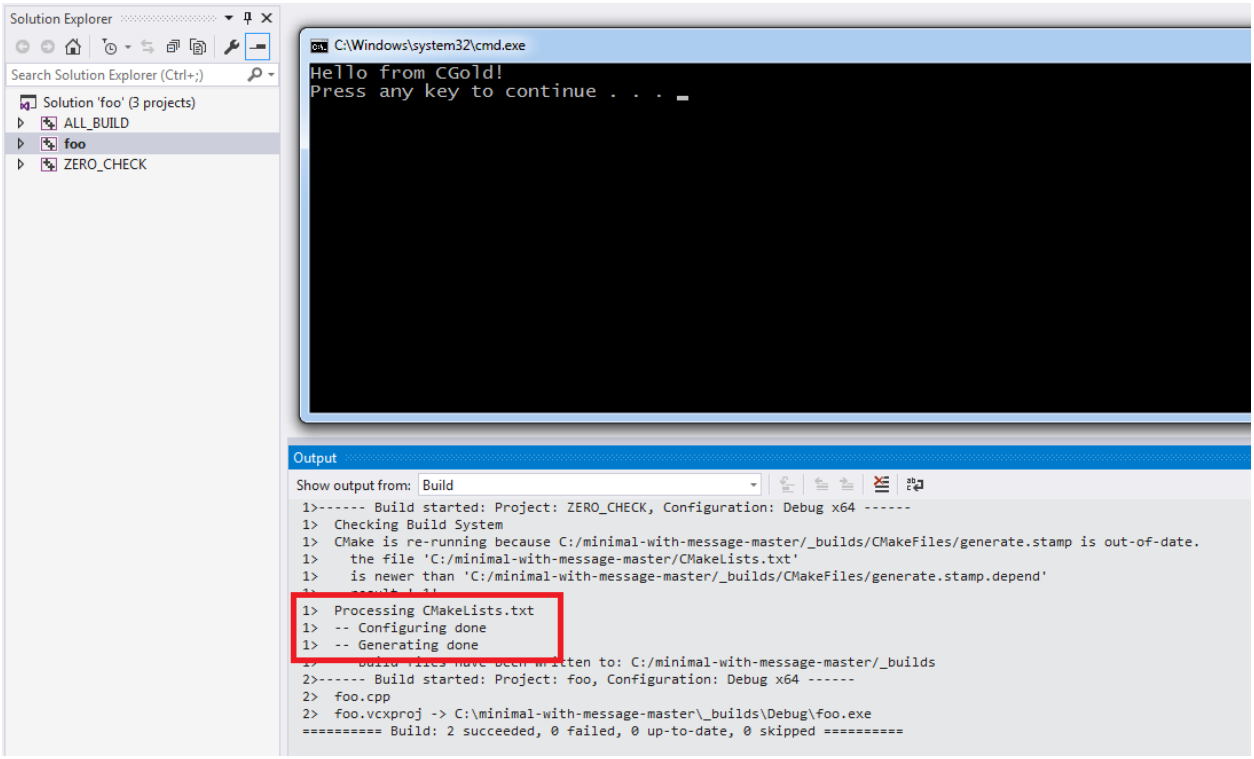
You see `Processing CMakeLists.txt`, `Configuring done` and `Generating done` indicating that CMake code parsed again and new Makefile generated. Since we don't change the way target `foo` is built (like adding new build flags or compile definitions) there is no compile/link stages.

If you "modify" both CMake and C++ code you will see the full configure/generate/build stack of commands:

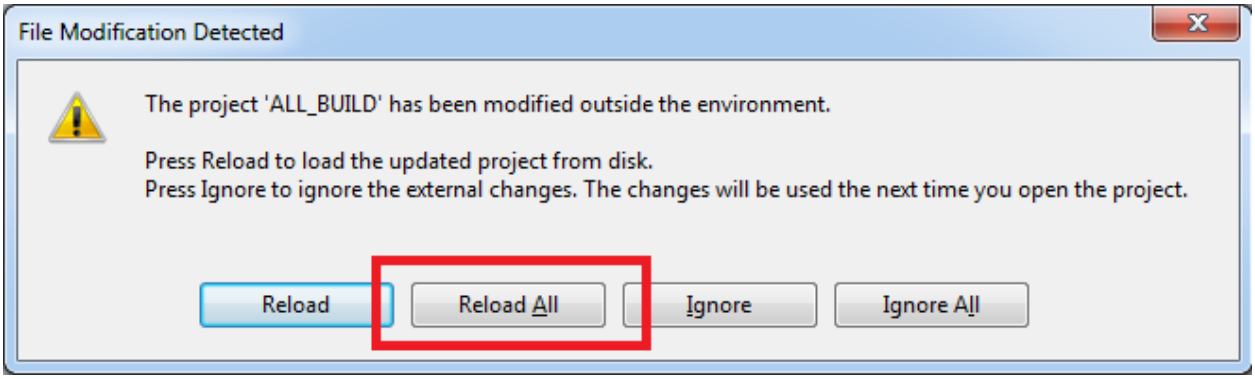
```
[minimal-with-message]> touch CMakeLists.txt foo.cpp
[minimal-with-message]> cmake --build _builds
Processing CMakeLists.txt
-- Configuring done
-- Generating done
-- Build files have been written to: /.../minimal-with-message/_builds
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX executable foo
[100%] Built target foo
```

3.3.2 Visual Studio example

Same is true for other generators as well. For example when you touch CMakeLists.txt and try to run `foo` target in Visual Studio:

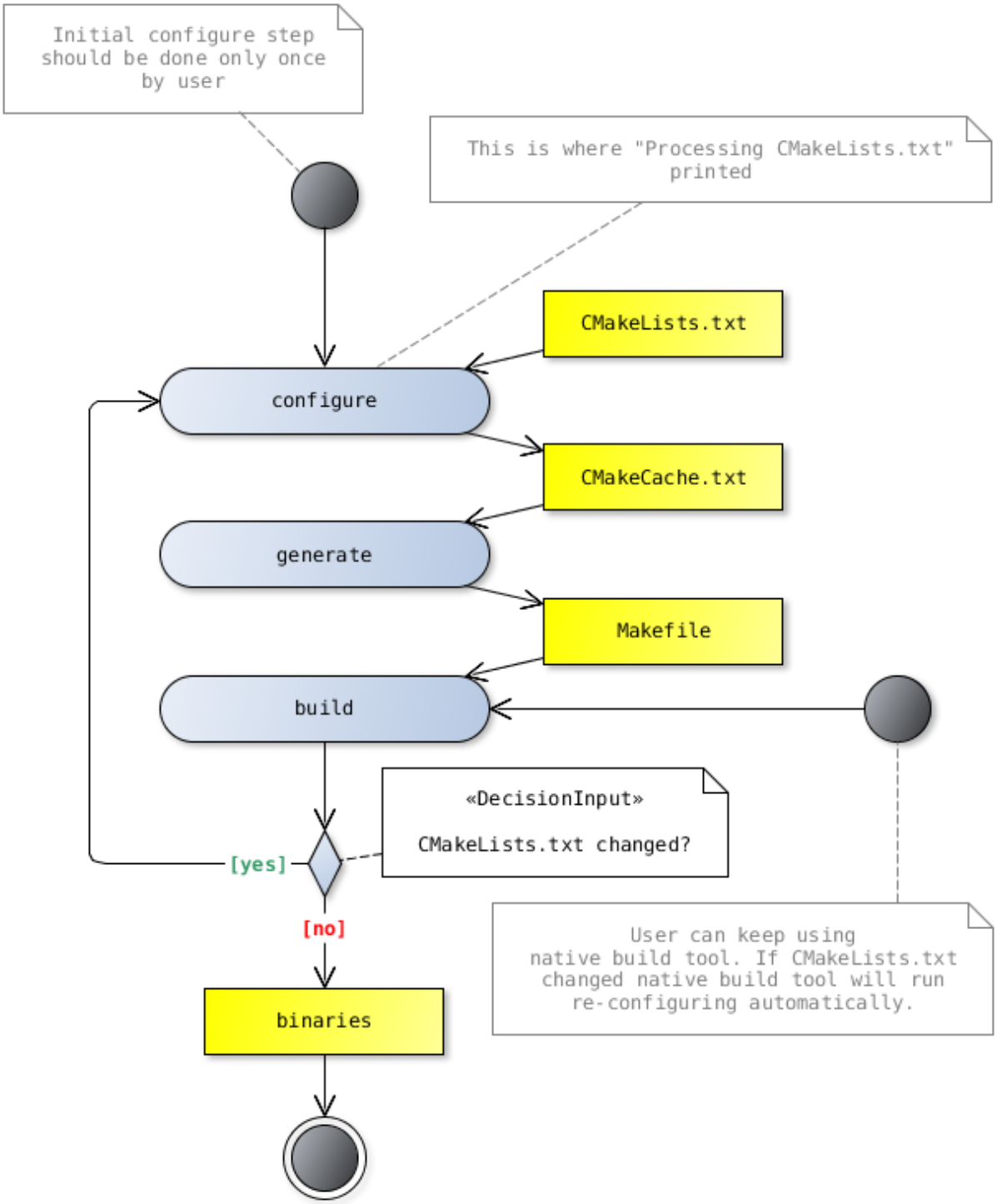


IDE will notify you about update of the project. You can click “Reload All” to reload new configuration:



3.3.3 UML activity diagram

Activity diagram for workflow described above:



3.3.4 Suspicious behavior

If your workflow doesn't match configure-once approach then it may be a symptom of wrongly written CMake code. Especially when you have to run `cmake -H. -B_builds` twice or when `cmake --build _builds` doesn't catch updates from CMake code.

CMake issue

- XCode: Real targets do not depend on ZERO_CHECK

3.4 Version and policies

Like any other piece of software *CMake* evolves, effectively introducing new features and deprecating dangerous or confusing behavior.

There are two entities that help you to manage difference between old and new versions of CMake:

- Command cmake_minimum_required: for checking what minimum version of CMake user should have to run your configuration
- CMake policies: for fine tuning newly introduced behavior

If you just want to experiment without worrying about backward compatibility, policies, warnings, etc. just set **first line** of CMakeLists.txt to `cmake_minimum_required(VERSION a.b.c)` where `a.b.c` is a current version of CMake you're using:

```
> cmake --version
cmake version 3.5.2

> cat CMakeLists.txt
cmake_minimum_required(VERSION 3.5.2)
```

3.4.1 cmake_minimum_required

CMake documentation

- cmake_minimum_required

What version to put into this command is mostly an executive decision. You need to know:

- what version is installed on users hosts?
- is it appropriate to ask them to install newer version?
- what features do they need?
- do you need to be backward compatible for one users and have fresh features for another?

The last case will fit most of them but will harder to maintain for developer and probably will require automatic testing system with good coverage.

See also:

- CMake versions for Hunter

For example the code with version `2.8` as a minimum one and with `3.0` features will look like:

```
cmake_minimum_required(VERSION 2.8)

if(NOT CMAKE_VERSION VERSION_LESS "3.0") # means 'NOT version < 3.0', i.e. 'version >
 ↪= 3.0'
```

```
  # Code with 3.0 features
endif()
```

Command `cmake_minimum_required` **must be the first** command in your *CMakeLists.txt*. If you're planning to support several versions of CMake then you need to put the smallest one in `cmake_minimum_required` and call it in the first line of CMakeLists.txt. Even if some commands look harmless at the first glance it may be not so in fact, e.g. `project` is the place where a lot of checks happens and where toolchain is loaded. If you run this example on `Cygwin` platform:

```
project(foo) # BAD CODE! You should check version first!
cmake_minimum_required(VERSION 3.0)

message("Using CMake version ${CMAKE_VERSION}")
```

CMake will think that you're running code with old policies and warns you:

```
[minimum-required-example]> cmake -Hbad -B_builds/bad
-- The C compiler identification is GNU 4.9.3
-- The CXX compiler identification is GNU 4.9.3
CMake Warning at /.../share/cmake-3.3.1/Modules/Platform/CYGWIN.cmake:15 (message):
  CMake no longer defines WIN32 on Cygwin!

  (1) If you are just trying to build this project, ignore this warning or
  quiet it by setting CMAKE_LEGACY_CYGWIN_WIN32=0 in your environment or in
  the CMake cache.  If later configuration or build errors occur then this
  project may have been written under the assumption that Cygwin is WIN32.
  In that case, set CMAKE_LEGACY_CYGWIN_WIN32=1 instead.

  (2) If you are developing this project, add the line

    set(CMAKE_LEGACY_CYGWIN_WIN32 0) # Remove when CMake >= 2.8.4 is required

  at the top of your top-level CMakeLists.txt file or set the minimum
  required version of CMake to 2.8.4 or higher.  Then teach your project to
  build on Cygwin without WIN32.
Call Stack (most recent call first):
  /.../share/cmake-3.3.1/Modules/CMakeSystemSpecificInformation.cmake:36 (include)
  CMakeLists.txt:1 (project)
...
-- Detecting CXX compile features - done
Using CMake version 3.3.1
...
```

Fixed version:

```
cmake_minimum_required(VERSION 3.0)
project(foo)

message("Using CMake version ${CMAKE_VERSION}")
```

with no warnings:

```
[minimum-required-example]> cmake -Hgood -B_builds/good
-- The C compiler identification is GNU 4.9.3
-- The CXX compiler identification is GNU 4.9.3
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
```

```
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++.exe
-- Check for working CXX compiler: /usr/bin/c++.exe -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
Using CMake version 3.3.1
-- Configuring done
-- Generating done
-- Build files have been written to: /.../minimum-required-example/_builds/good
```

See also:

- CMake issue #17712

Examples on GitHub

- Repository
- Latest ZIP

3.4.2 CMake policies

CMake documentation

- CMake policies

When new version of CMake released there may be a list of policies describing cases when behavior changed comparing to the previous CMake version.

Let's see how it works on practice. In CMake `3.0` policy CMP0038 was introduced. Before version `3.0` user can have target linked to itself, which make no sense and definitely **is a bug**:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

add_library(foo foo.cpp)

target_link_libraries(foo foo) # BAD CODE! Make no sense
```

Examples on GitHub

- Repository
- Latest ZIP

Works fine for CMake before `3.0`:

```
[policy-examples]> cmake --version
cmake version 2.8.12.2
```

```
[policy-examples]> rm -rf _builds
[policy-examples]> cmake -Hbug-2.8 -B_builds
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /.../policy-examples/_builds
```

For CMake version >= `3.0` warning will be reported:

```
[policy-examples]> cmake --version
cmake version 3.5.2
[policy-examples]> rm -rf _builds
[policy-examples]> cmake -Hbug-2.8 -B_builds
...
-- Configuring done
CMake Warning (dev) at CMakeLists.txt:4 (add_library):
  Policy CMP0038 is not set: Targets may not link directly to themselves.
  Run "cmake --help-policy CMP0038" for policy details.  Use the cmake_policy
  command to set the policy and suppress this warning.

  Target "foo" links to itself.
This warning is for project developers.  Use -Wno-dev to suppress it.

-- Generating done
-- Build files have been written to: /.../policy-examples/_builds
```

Assume you want to drop the support of old version and more to some new `3.0` features. When you set `cmake_minimum_required(VERSION 3.0)`

```
--- /examples/policy-examples/bug-2.8/CMakeLists.txt
+++ /examples/policy-examples/set-3.0/CMakeLists.txt
@@ -1,4 +1,4 @@
-cmake_minimum_required(VERSION 2.8)
+cmake_minimum_required(VERSION 3.0)
 project(foo)

 add_library(foo foo.cpp)
```

warning turns into error:

```
[policy-examples]> rm -rf _builds
[policy-examples]> cmake -Hset-3.0 -B_builds
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
```

```
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
CMake Error at CMakeLists.txt:4 (add_library):
  Target "foo" links to itself.


-- Generating done
-- Build files have been written to: /.../policy-examples/_builds
[policy-examples]> echo $?
1
```

Two cases will be shown below. In first case we want to keep support of old version (`2.8` for now) so it will work with both `CMake 2.8` and `CMake 3.0+`. In second case we decide to drop support of old version and move to `CMake 3.0+`. We'll see how it will affect policies. It will be shown in the end that in fact without **using new features** from `CMake 3.0` it doesn't make sense to change `cmake_minimum_required`.

Keep using old

Our project works fine with `CMake 2.8` however `CMake 3.0+` emits warning. We don't want to fix the error now but want only to suppress warning and explain to CMake that it should behaves like `CMake 2.8`.

Note: This approach described in documentation:

```
It is possible to disable the warning by explicitly requesting the OLD, or
backward compatible behavior using the cmake_policy() command
```

Let's add cmake_policy:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

cmake_policy(SET CMP0038 OLD)

add_library(foo foo.cpp)

target_link_libraries(foo foo) # BAD CODE! Make no sense
```

Looks good for `CMake 3.0+`:

```
[policy-examples]> cmake --version
cmake version 3.5.2
[policy-examples]> rm -rf _builds
[policy-examples]> cmake -Hunknown-2.8 -B_builds
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
```

```
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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
```

Are we done? No, `CMP0038` is introduced since `CMake 3.0` so `CMake 2.8` have no idea what this policy is about:

```
> cmake --version
cmake version 2.8.12.2
> rm -rf _builds
> cmake -Hunknown-2.8 -B_builds
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
CMake Error at CMakeLists.txt:4 (cmake_policy):
  Policy "CMP0038" is not known to this version of CMake.

-- Configuring incomplete, errors occurred!
```

We should protect new code with `if(POLICY CMP0038)` condition:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

if(POLICY CMP0038)
  # Policy CMP0038 introduced since CMake 3.0 so if we want to be compatible
  # with 2.8 (see cmake_minimum_required) we should put 'cmake_policy' under
  # condition.
  cmake_policy(SET CMP0038 OLD)
endif()

add_library(foo foo.cpp)

target_link_libraries(foo foo) # BAD CODE! Make no sense
```

Of course you should find the time, apply real fix and remove policy logic since it will not be needed anymore:

```
--- /examples/policy-examples/suppress-2.8/CMakeLists.txt
+++ /examples/policy-examples/fix-2.8/CMakeLists.txt
@@ -1,13 +1,4 @@
 cmake_minimum_required(VERSION 2.8)
 project(foo)

-if(POLICY CMP0038)
-  # Policy CMP0038 introduced since CMake 3.0 so if we want to be compatible
```

```
-  # with 2.8 (see cmake_minimum_required) we should put 'cmake_policy' under
-  # condition.
-  cmake_policy(SET CMP0038 OLD)
-endif()
-
 add_library(foo foo.cpp)
-
-target_link_libraries(foo foo) # BAD CODE! Make no sense
```

Final version:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

add_library(foo foo.cpp)
```

Moving to new version

With `cmake_minimum_required` updated to `3.0` warning turns into error. To suppress error without doing real fix (temporary solution) you can add `cmake_policy` directive:

```
cmake_minimum_required(VERSION 3.0)
project(foo)

cmake_policy(SET CMP0038 OLD)

add_library(foo foo.cpp)

target_link_libraries(foo foo) # BAD CODE! Make no sense
```

Note: We don't need to protect `cmake_policy` with `if(POLICY)` condition since `cmake_minimum_required(VERSION 3.0)` guarantee us that we are using `CMake 3.0+`.

Policy can be removed after real fix applied:

```
--- /examples/policy-examples/suppress-3.0/CMakeLists.txt
+++ /examples/policy-examples/fix-3.0/CMakeLists.txt
@@ -1,8 +1,4 @@
 cmake_minimum_required(VERSION 3.0)
 project(foo)

-cmake_policy(SET CMP0038 OLD)
-
 add_library(foo foo.cpp)
-
-target_link_libraries(foo foo) # BAD CODE! Make no sense
```

Final version:

```
cmake_minimum_required(VERSION 3.0)
project(foo)

add_library(foo foo.cpp)
```

You may notice that final version for both cases differs only in `cmake_minimum_required`:

```
--- /examples/policy-examples/fix-2.8/CMakeLists.txt
+++ /examples/policy-examples/fix-3.0/CMakeLists.txt
@@ -1,4 +1,4 @@
-cmake_minimum_required(VERSION 2.8)
+cmake_minimum_required(VERSION 3.0)
 project(foo)

 add_library(foo foo.cpp)
```

It means that there is no much sense in changing `cmake_minimum_required` without using any **new features**.

3.4.3 Summary

- Policies can be used to control CMake **behavior**
- Policies can be used to suppress warnings/errors
- `cmake_minimum_required` describe **features** you use in CMake code
- For backward compatibility new features can be protected with `if(CMAKE_VERSION ...)` directive

3.5 Project declaration

Next must-have command is project. Command `project(foo)` will set languages to C and C++ (default), declare some `foo_*` variables and run basic build tool checks.

CMake documentation

- project

3.5.1 Tools discovering

By default on calling `project` command CMake will try to detect compilers for default languages: C and C++. Let's add some variables and check where they are defined:

```
cmake_minimum_required(VERSION 2.8)

message("Before 'project':")
message("  C: '${CMAKE_C_COMPILER}'")
message("  C++: '${CMAKE_CXX_COMPILER}'")

project(Foo)

message("After 'project':")
message("  C: '${CMAKE_C_COMPILER}'")
message("  C++: '${CMAKE_CXX_COMPILER}'")
```

Examples on GitHub

- Repository

- Latest ZIP

Run test on `Linux`:

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Hset-compiler -B_builds
Before 'project':
  C: ''
  C++: ''
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
After 'project':
  C: '/usr/bin/cc'
  C++: '/usr/bin/c++'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

CMake will run tests for other tools as well, so try to avoid checking of anything before `project`, place all checks **after project declared**.

Also `project` is a place where toolchain file will be read.

```
# CMakeLists.txt

cmake_minimum_required(VERSION 2.8)

message("Before 'project'")

project(Foo)

message("After 'project'")
```

```
# toolchain.cmake

message("Processing toolchain")
```

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Htoolchain -B_builds -DCMAKE_TOOLCHAIN_FILE=toolchain.cmake
Before 'project'
Processing toolchain
Processing toolchain
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
```

```
Processing toolchain
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
Processing toolchain
-- Detecting C compiler ABI info - done
-- Detecting C compile features
Processing toolchain
Processing toolchain
Processing toolchain
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
Processing toolchain
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
Processing toolchain
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
Processing toolchain
Processing toolchain
Processing toolchain
-- Detecting CXX compile features - done
After 'project'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

Note: You may notice that toolchain read several times

3.5.2 Languages

If you don't have or don't need support for one of the default languages you can set language explicitly after name of the project. This is how to setup C-only project:

```
cmake_minimum_required(VERSION 2.8)

message("Before 'project':")
message("  C: '${CMAKE_C_COMPILER}'")
message("  C++: '${CMAKE_CXX_COMPILER}'")

project(Foo C)

message("After 'project':")
message("  C: '${CMAKE_C_COMPILER}'")
message("  C++: '${CMAKE_CXX_COMPILER}'")
```

There is no checks for C++ compiler and variable with path to C++ compiler is empty now:

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Hc-compiler -B_builds
Before 'project':
  C: ''
  C++: ''
-- The C compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
```

```
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
After 'project':
  C: '/usr/bin/cc'
  C++: ''
-- Configuring done
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

Of course you will not be able to build C++ targets anymore. Since CMake thinks that `*.cpp` extension is for C++ sources (by default) there will be error reported if C++ is not listed (discovering of C++ tools will not be triggered):

```
cmake_minimum_required(VERSION 2.8)
project(Foo C)

add_library(foo foo.cpp)
```

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Hcpp-not-found -B_builds
-- The C compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
CMake Error: Cannot determine link language for target "foo".
CMake Error: CMake can not determine linker language for target: foo
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

We can save some time by using special language `NONE` when we don't need any tools at all:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)
```

No checks for C or C++ compiler as you can see:

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Hno-language -B_builds
-- Configuring done
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

Note: Such form will be used widely in examples in cases when we don't need to build targets.

Note: For CMake 3.0+ sub-option `LANGUAGES` added, since it will be:

```
cmake_minimum_required(VERSION 3.0)
project(foo LANGUAGES NONE)
```

3.5.3 Variables

Command `project` declare `*_{SOURCE,BINARY}_DIR` variables. Since version `3.0` you can add `VERSION` which additionally declare `*_VERSION_{MAJOR,MINOR,PATCH,TWEAK}` variables:

```
cmake_minimum_required(VERSION 3.0)

message("Before project:")
message("  Source: ${PROJECT_SOURCE_DIR}")
message("  Binary: ${PROJECT_BINARY_DIR}")
message("  Version: ${PROJECT_VERSION}")
message("  Version (alt): ${PROJECT_VERSION_MAJOR}.${PROJECT_VERSION_MINOR}.${PROJECT_
↪VERSION_PATCH}")

project(Foo VERSION 1.2.7)

message("After project:")
message("  Source: ${PROJECT_SOURCE_DIR}")
message("  Binary: ${PROJECT_BINARY_DIR}")
message("  Version: ${PROJECT_VERSION}")
message("  Version (alt): ${PROJECT_VERSION_MAJOR}.${PROJECT_VERSION_MINOR}.${PROJECT_
↪VERSION_PATCH}")
```

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Hvariables -B_builds
Before project:
  Source:
  Binary:
  Version:
  Version (alt): ..
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
After project:
  Source: /.../project-examples/variables
  Binary: /.../project-examples/_builds
  Version: 1.2.7
  Version (alt): 1.2.7
-- Configuring done
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

You can use alternative `foo_{SOURCE,BINARY}_DIRS`/ `foo_VERSION_{MINOR,MAJOR,PATCH}` synonyms. This is useful when you have hierarchy of projects:

```
# Top level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

message("From top level:")
message("  Source (general): ${PROJECT_SOURCE_DIR}")
message("  Source (foo): ${foo_SOURCE_DIR}")

add_subdirectory(boo)
```

```
# CMakeLists.txt from 'boo' directory

cmake_minimum_required(VERSION 2.8)
project(boo)

message("From subdirectory 'boo':")
message("  Source (general): ${PROJECT_SOURCE_DIR}")
message("  Source (foo): ${foo_SOURCE_DIR}")
message("  Source (boo): ${boo_SOURCE_DIR}")
```

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Hhierarchy -B_builds
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
From top level:
  Source (general): /.../project-examples/hierarchy
  Source (foo): /.../project-examples/hierarchy
From subdirectory 'boo':
  Source (general): /.../project-examples/hierarchy/boo
  Source (foo): /.../project-examples/hierarchy
  Source (boo): /.../project-examples/hierarchy/boo
-- Configuring done
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

As you can see we are still able to use `foo_*` variables even if new command `project(boo)` called.

3.5.4 When not declared

CMake will implicitly declare `project` in case there is no such command in top-level CMakeLists.txt. This will be equal to calling `project` before any other commands. It means that `project` will be called **before** `cmake_minimum_required` so can lead to problems described in *previous section*:

```
# Top level CMakeLists.txt

message("Before 'cmake_minimum_required'")
cmake_minimum_required(VERSION 2.8)

add_subdirectory(boo)
```

```
# CMakeLists.txt in directory 'boo'

cmake_minimum_required(VERSION 2.8)
project(boo)
```

```
[project-examples]> rm -rf _builds
[project-examples]> cmake -Hnot-declared -B_builds
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
Before 'cmake_minimum_required'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../project-examples/_builds
```

3.5.5 Summary

- You must have `project` command in your top-level `CMakeLists.txt`
- Use `project` to declare non divisible monolithic hierarchy of targets
- Try to minimize the number of instructions before `project` and verify that variables are declared in such block of CMake code

3.6 Variables

> *There are only two kinds of languages: the ones people complain about and the ones nobody uses.*
> – Bjarne Stroustrup

We have touched already some simple syntax like dereferencing variable `A` by `${A}` in `message` command: `message("This is A: ${A}")`. Cache variables was mentioned in *CMake stages*. Here is an overview of different types of variables with examples.

CMake documentation

- Language: variables

Examples on GitHub

- Repository
- Latest ZIP

3.6.1 Regular variables

Regular vs cache

Unlike *cache variables* regular (normal) CMake variables have scope and don't outlive CMake runs.

If in the next example you run the CMake configure step twice, without removing the cache:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

message("Regular variable (before): ${abc}")
message("Cache variable (before): ${xyz}")

set(abc "123")
set(xyz "321" CACHE STRING "")

message("Regular variable (after): ${abc}")
message("Cache variable (after): ${xyz}")
```

```
[usage-of-variables]> rm -rf _builds

[usage-of-variables]> cmake -Hcache-vs-regular -B_builds
Regular variable (before):
Cache variable (before):
Regular variable (after): 123
Cache variable (after): 321
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds

[usage-of-variables]> cmake -Hcache-vs-regular -B_builds
Regular variable (before):
Cache variable (before): 321
Regular variable (after): 123
Cache variable (after): 321
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

You can see that the regular CMake variable `abc` is created from scratch each time

```
[usage-of-variables]> rm -rf _builds

[usage-of-variables]> cmake -Hcache-vs-regular -B_builds
Regular variable (before):
Cache variable (before):
```

```
Regular variable (after): 123
Cache variable (after): 321
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds

[usage-of-variables]> cmake -Hcache-vs-regular -B_builds
Regular variable (before):
Cache variable (before): 321
Regular variable (after): 123
Cache variable (after): 321
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

And the cache variable `xyz` is created only once and reused on second run

```
[usage-of-variables]> rm -rf _builds

[usage-of-variables]> cmake -Hcache-vs-regular -B_builds
Regular variable (before):
Cache variable (before):
Regular variable (after): 123
Cache variable (after): 321
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds

[usage-of-variables]> cmake -Hcache-vs-regular -B_builds
Regular variable (before):
Cache variable (before): 321
Regular variable (after): 123
Cache variable (after): 321
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

You can find cache variable `xyz` in *CMakeCache.txt*:

```
[usage-of-variables]> grep xyz _builds/CMakeCache.txt
xyz:STRING=321
```

Unlike regular `abc`:

```
[usage-of-variables]> grep abc _builds/CMakeCache.txt
[usage-of-variables]> echo $?
1
```

Scope of variable

Each variable is linked to the scope where it was defined. Commands add_subdirectory and function introduce their own scopes:

```
# Top level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
```

```
project(foo NONE)

set(abc "123")

message("Top level scope (before): ${abc}")

add_subdirectory(boo)

message("Top level scope (after): ${abc}")
```

```
# CMakeLists.txt from 'boo' directory

set(abc "456")

message("Directory 'boo' scope: ${abc}")
```

There are two variables `abc` defined. One in top level scope and another in scope of `boo` directory:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hdirectory-scope -B_builds
Top level scope (before): 123
Directory 'boo' scope: 456
Top level scope (after): 123
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

New scope

When a new scope is created it will be initialized with the variables of the parent scope. Command unset can remove a variable from the current scope. If a variable is not found in the current scope it will be dereferenced to an empty string:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

function(foo)
  message("[foo]: Scope for function 'foo' copied from parent 'boo': { abc = '${abc}
↪', xyz = '${xyz}' }")
  unset(abc)
  message("[foo]: Command 'unset(abc)' will remove variable from current scope: { abc␣
↪= '${abc}', xyz = '${xyz}' }")
endfunction()

function(boo)
  message("[boo]: Scope for function 'boo' copied from parent: { abc = '${abc}', xyz␣
↪= '${xyz}' }")
  set(abc "789")
  message("[boo]: Command 'set(abc ...)' modify current scope, state: { abc = '${abc}
↪', xyz = '${xyz}' }")
  foo()
endfunction()

set(abc "123")
set(xyz "456")
```

```
message("Top level scope state: { abc = '${abc}', xyz = '${xyz}' }")

boo()
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Htake-from-parent-scope -B_builds
Top level scope state: { abc = '123', xyz = '456' }
[boo]: Scope for function 'boo' copied from parent: { abc = '123', xyz = '456' }
[boo]: Command 'set(abc ...)' modify current scope, state: { abc = '789', xyz = '456'␣
↪}
[foo]: Scope for function 'foo' copied from parent 'boo': { abc = '789', xyz = '456' }
[foo]: Command 'unset(abc)' will remove variable from current scope: { abc = '', xyz␣
↪= '456' }
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Same scope

`include` and `macro` don't introduce a new scope, so commands like `set` and `unset` will affect the current scope:

```
# Top level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(abc "123")

message("abc (before): ${abc}")
include("./modify-abc.cmake")
message("abc (after): ${abc}")

macro(modify_xyz)
  set(xyz "789")
endmacro()

set(xyz "336")

message("xyz (before): ${xyz}")
modify_xyz()
message("xyz (after): ${xyz}")
```

```
# modify-abc.cmake module

set(abc "456")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hsame-scope -B_builds
abc (before): 123
abc (after): 456
xyz (before): 336
xyz (after): 789
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Parent scope

A variable can be set to the parent scope by specifying `PARENT_SCOPE`:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(abc "") # clear

function(scope_2)
  message("Scope 2 (before): '${abc}'")
  set(abc "786" PARENT_SCOPE)
  message("Scope 2 (after): '${abc}'")
endfunction()

function(scope_1)
  message("Scope 1 (before): '${abc}'")
  scope_2()
  message("Scope 1 (after): '${abc}'")
endfunction()

message("Top level (before): '${abc}'")
scope_1()
message("Top level (after): '${abc}'")
```

Variable **will only be set to parent** scope:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hparent-scope -B_builds
Top level (before): ''
Scope 1 (before): ''
Scope 2 (before): ''
Scope 2 (after): ''
Scope 1 (after): '786'
Top level (after): ''
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Current scope will not be affected:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hparent-scope -B_builds
Top level (before): ''
Scope 1 (before): ''
Scope 2 (before): ''
Scope 2 (after): ''
Scope 1 (after): '786'
Top level (after): ''
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

As well as parent of the parent:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hparent-scope -B_builds
Top level (before): ''
```

```
Scope 1 (before): ''
Scope 2 (before): ''
Scope 2 (after): ''
Scope 1 (after): '786'
Top level (after): ''
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

From cache

If variable is not found in the current scope, it will be taken from the cache:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "789" CACHE STRING "")
set(a "123")

message("Regular variable from current scope: ${a}")

unset(a)

message("Cache variable if regular not found: ${a}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hfrom-cache -B_builds
Regular variable from current scope: 123
Cache variable if regular not found: 789
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Cache unset regular

Note that the order of commands is important because `set(... CACHE ...)` will remove the regular variable with the same name from current scope:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "123")
set(a "789" CACHE STRING "")

message("Regular variable unset, take from cache: ${a}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hcache-remove-regular -B_builds
Regular variable unset, take from cache: 789
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Confusing

This may lead to a quite confusing behavior:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

function(set_abc_globally)
  message("Function scope before cache modify = ${abc}")
  set(abc "789" CACHE STRING "")
  message("Function scope after cache modify = ${abc}")
endfunction()

set(abc "123")

set_abc_globally()

message("Parent scope is not affected, take variable from current scope, not cache = $
↪{abc}")
```

In this example `set(... CACHE ...)` will remove `abc` only from scope of function and not from top level scope:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hcache-confuse -B_builds
Function scope before cache modify = 123
Function scope after cache modify = 789
Parent scope is not affected, take variable from current scope, not cache = 123
-- Configuring done
-- Generating done
-- build files have been written to: /.../usage-of-variables/_builds
```

This will be even more confusing if you run this example one more time without removing cache:

```
[usage-of-variables]> cmake -Hcache-confuse -B_builds
Function scope before cache modify = 123
Function scope after cache modify = 123
Parent scope is not affected, take variable from current scope, not cache = 123
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Since variable `abc` already stored in cache command `set(... CACHE ...)` has no effect and **will not remove** regular `abc` from scope of function.

Names

Variable names are case-sensitive:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "123")
set(b "567")
set(aBc "333")

set(A "321")
```

```
set(B "765")
set(ABc "777")

message("a: ${a}")
message("b: ${b}")
message("aBc: ${aBc}")

message("A: ${A}")
message("B: ${B}")
message("ABc: ${ABc}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hcase-sensitive -B_builds
a: 123
b: 567
aBc: 333
A: 321
B: 765
ABc: 777
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Name of variable may consist of **any** characters:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set("abc" "123")
set("ab c" "456")
set("ab?c" "789")
set("/usr/bin/bash" "987")
set("C:\\Program Files\\" "654")
set(" " "321")

function(print_name varname)
  message("Variable name: '${varname}', value: '${${varname}}'")
endfunction()

print_name("abc")
print_name("ab c")
print_name("ab?c")
print_name("/usr/bin/bash")
print_name("C:\\Program Files\\")
print_name(" ")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hany-names -B_builds
Variable name: 'abc', value: '123'
Variable name: 'ab c', value: '456'
Variable name: 'ab?c', value: '789'
Variable name: '/usr/bin/bash', value: '987'
Variable name: 'C:\Program Files\', value: '654'
Variable name: ' ', value: '321'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Quotes

In the previous example, the quote character " was used to create a name containing a space - this is called *quoted argument*. Note that the argument must start and end with a quote character, otherwise it becomes an *unquoted argument*. In this case, the quote character will be treated as part of the string:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "Quoted argument")
set(b x-"Unquoted argument")
set(c x"a;b;c")

message("a = '${a}'")
message("b = '${b}'")

message("c =")
foreach(x ${c})
  message("  '${x}'")
endforeach()
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hquotes -B_builds
a = 'Quoted argument'
b = 'x-"Unquoted argument"'
c =
  'x"a'
  'b'
  'c"'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

As you can see the variable `b` contains quotes now and for list `c` quotes are part of the elements: `x"a`, `c"`.

CMake documentation

- Quoted argument
- Unquoted argument

Dereferencing

Dereferenced variable can be used in creation of new variable:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "xyz")

set(b "${a}_321")
set(${a}_1 "456")
set(variable_${a} "${a} + ${b} + 155")

message("b: '${b}'")
```

```
message("xyz_1: '${xyz_1}'")
message("variable_xyz: '${variable_xyz}'")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hdereference -B_builds
b: 'xyz_321'
xyz_1: '456'
variable_xyz: 'xyz + xyz_321 + 155'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Or new variable name:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "xyz")

set(b "${a}_321")
set(${a}_1 "456")
set(variable_${a} "${a} + ${b} + 155")

message("b: '${b}'")
message("xyz_1: '${xyz_1}'")
message("variable_xyz: '${variable_xyz}'")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hdereference -B_builds
b: 'xyz_321'
xyz_1: '456'
variable_xyz: 'xyz + xyz_321 + 155'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Or even both:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "xyz")

set(b "${a}_321")
set(${a}_1 "456")
set(variable_${a} "${a} + ${b} + 155")

message("b: '${b}'")
message("xyz_1: '${xyz_1}'")
message("variable_xyz: '${variable_xyz}'")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hdereference -B_builds
b: 'xyz_321'
xyz_1: '456'
variable_xyz: 'xyz + xyz_321 + 155'
-- Configuring done
```

```
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Nested dereferencing

Dereferencing of variable by `${...}` will happen as many times as needed:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

foreach(lang C CXX)
  message("Compiler for language ${lang}: ${CMAKE_${lang}_COMPILER}")
  foreach(build_type DEBUG RELEASE RELWITHDEBINFO MINSIZEREL)
    message("Flags for language ${lang} + build type ${build_type}: ${CMAKE_${lang}_
↪FLAGS_${build_type}}")
  endforeach()
endforeach()
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hnested-dereference -B_builds
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
Compiler for language C: /usr/bin/cc
Flags for language C + build type DEBUG: -g
Flags for language C + build type RELEASE: -O3 -DNDEBUG
Flags for language C + build type RELWITHDEBINFO: -O2 -g -DNDEBUG
Flags for language C + build type MINSIZEREL: -Os -DNDEBUG
Compiler for language CXX: /usr/bin/c++
Flags for language CXX + build type DEBUG: -g
Flags for language CXX + build type RELEASE: -O3 -DNDEBUG
Flags for language CXX + build type RELWITHDEBINFO: -O2 -g -DNDEBUG
Flags for language CXX + build type MINSIZEREL: -Os -DNDEBUG
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Types of variable

Variables always have type string but some commands can interpret them differently. For example the command `if` can treat strings as boolean, path, target name, etc.:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

set(condition_a "TRUE")
set(condition_b "NO")

set(path_to_this "${CMAKE_CURRENT_SOURCE_DIR}/CMakeLists.txt")

set(target_name foo)
add_library("${target_name}" foo.cpp)

if(condition_a)
  message("condition_a")
else()
  message("NOT condition_a")
endif()

if(condition_b)
  message("condition_b")
else()
  message("NOT condition_b")
endif()

if(EXISTS "${path_to_this}")
  message("File exists: ${path_to_this}")
else()
  message("File not exist: ${path_to_this}")
endif()

if(TARGET "${target_name}")
  message("Target exists: ${target_name}")
else()
  message("Target not exist: ${target_name}")
endif()
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Htypes-of-variable -B_builds
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
condition_a
NOT condition_b
File exists: /.../usage-of-variables/types-of-variable/CMakeLists.txt
Target exists: foo
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

CMake documentation

- if

Create list

Some commands can treat a variable as list. In this case the string value is split into elements separated by `;`. The command `set` can create such lists:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(l0 a b c)
set(l1 a;b;c)
set(l2 "a b" "c")
set(l3 "a;b;c")
set(l4 a "b;c")

message("l0 = 'a' + 'b' + 'c' = '${l0}'")
message("l1 = 'a;b;c' = '${l1}'")
message("l2 = 'a b' + 'c' = '${l2}'")
message("l3 = \"'a;b;c'\" = '${l3}'")
message("l4 = 'a' + 'b;c' = '${l4}'")

message("print by message: " ${l3})
message("print by message: " "a" "b" "c")
```

`set` creates **string** from elements and puts the `;` between them:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hlist -B_builds
l0 = 'a' + 'b' + 'c' = 'a;b;c'
l1 = 'a;b;c' = 'a;b;c'
l2 = 'a b' + 'c' = 'a b;c'
l3 = "'a;b;c'" = 'a;b;c'
l4 = 'a' + 'b;c' = 'a;b;c'
print by message: abc
print by message: abc
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

In case you want to add an element with space you can protect the element with `"`:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hlist -B_builds
l0 = 'a' + 'b' + 'c' = 'a;b;c'
l1 = 'a;b;c' = 'a;b;c'
l2 = 'a b' + 'c' = 'a b;c'
l3 = "'a;b;c'" = 'a;b;c'
l4 = 'a' + 'b;c' = 'a;b;c'
print by message: abc
print by message: abc
-- Configuring done
```

```
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

As seen with `l4` variable protecting `;` with `"` doesn't have any effect:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hlist -B_builds
l0 = 'a' + 'b' + 'c' = 'a;b;c'
l1 = 'a;b;c' = 'a;b;c'
l2 = 'a b' + 'c' = 'a b;c'
l3 = "'a;b;c'" = 'a;b;c'
l4 = 'a' + 'b;c' = 'a;b;c'
print by message: abc
print by message: abc
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

We are concatenating **string** `a` with **string** `b;c` and putting `;` between them. Final result is the **string** `a;b;c`. When a command interprets this string as list, such list has 3 elements. Hence **it's not a list** with two elements `a` and `b;c`.

The command `message` interprets `l3` as list with 3 elements, so in the end 4 arguments (value of type string) passed as input: `print by message:_`, `a`, `b`, `c`. Command `message` will concatenate them without any separator, hence string `print by message:  abc` will be printed:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hlist -B_builds
l0 = 'a' + 'b' + 'c' = 'a;b;c'
l1 = 'a;b;c' = 'a;b;c'
l2 = 'a b' + 'c' = 'a b;c'
l3 = "'a;b;c'" = 'a;b;c'
l4 = 'a' + 'b;c' = 'a;b;c'
print by message: abc
print by message: abc
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

CMake documentation

- set

Operations with list

The `list` command can be used to calculate length of list, get element by index, remove elements by index, etc.:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(l0 "a;b;c")
set(l1 "a" "b;c")
set(l2 "a" "b c")

list(LENGTH l0 l0_len)
list(LENGTH l1 l1_len)
```

```
list(LENGTH l2 l2_len)

message("length of '${l0}' (l0) = ${l0_len}")
message("length of '${l1}' (l1) = ${l1_len}")
message("length of '${l2}' (l2) = ${l2_len}")

list(GET l1 2 l1_2)
message("l1[2] = ${l1_2}")

message("Removing first item from l1 list: '${l1}'")
list(REMOVE_AT l1 0)
message("l1 = '${l1}'")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hlist-operations -B_builds
length of 'a;b;c' (l0) = 3
length of 'a;b;c' (l1) = 3
length of 'a;b c' (l2) = 2
l1[2] = c
Removing first item from l1 list: 'a;b;c'
l1 = 'b;c'
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

CMake documentation

- list

List with one empty element

Since list is really just a string **there is no such object** as "list with one empty element". Empty string is a list with no elements - length is 0. String `;` is a list with two empty elements - length is 2.

Historically result of appending empty element to an empty list is an empty list:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

function(add_element list_name element_name)
  message("Add '${${element_name}}' to list '${${list_name}}'")
  list(APPEND "${list_name}" "${${element_name}}")
  list(LENGTH "${list_name}" list_len)
  message("Result: '${${list_name}}' (length = ${list_len})\n")
  set("${list_name}" "${${list_name}}" PARENT_SCOPE)
endfunction()

message("\nAdding non-empty element to non-empty list.\n")
set(mylist "a;b")
set(element "c")
foreach(i RANGE 3)
  add_element(mylist element)
endforeach()

message("\nAdding empty element to non-empty list.\n")
```

```
set(mylist "a;b")
set(element "")
foreach(i RANGE 3)
  add_element(mylist element)
endforeach()

message("\nAdding empty element to empty list.\n")
set(mylist "")
set(element "")
foreach(i RANGE 3)
  add_element(mylist element)
endforeach()
```

```
[examples]> rm -rf _builds
[examples]> cmake -Husage-of-variables/empty-list -B_builds

Adding non-empty element to non-empty list.

Add 'c' to list 'a;b'
Result: 'a;b;c' (length = 3)

Add 'c' to list 'a;b;c'
Result: 'a;b;c;c' (length = 4)

Add 'c' to list 'a;b;c;c'
Result: 'a;b;c;c;c' (length = 5)

Add 'c' to list 'a;b;c;c;c'
Result: 'a;b;c;c;c;c' (length = 6)


Adding empty element to non-empty list.

Add '' to list 'a;b'
Result: 'a;b;' (length = 3)

Add '' to list 'a;b;'
Result: 'a;b;;' (length = 4)

Add '' to list 'a;b;;'
Result: 'a;b;;;' (length = 5)

Add '' to list 'a;b;;;'
Result: 'a;b;;;;' (length = 6)


Adding empty element to empty list.

Add '' to list ''
Result: '' (length = 0)

Add '' to list ''
Result: '' (length = 0)

Add '' to list ''
Result: '' (length = 0)

Add '' to list ''
```

```
Result: '' (length = 0)

-- Configuring done
-- Generating done
-- Build files have been written to: /.../examples/_builds
```

Recommendation

Use **short laconic lower-case** names (`a`, `i`, `mylist`, `objects`, etc.) for local variables that used **only by the current scope**. Use **long detailed upper-case** names (`FOO_FEATURE`, `BOO_ENABLE_SOMETHING`, etc.) for variables that used by **several scopes**.

For example it make no sense to use long names in function since function has it's own scope:

```
function(foo_something)
  set(FOO_SOMETHING_A 1)
  # ...
endfunction()
```

Using just `a` will be fine:

```
function(foo_something)
  set(a 1)
  # ...
endfunction()
```

Same with scope of *CMakeLists.txt*:

```
# Foo/CMakeLists.txt

message("Files:")
foreach(FOO_FILES_ITERATOR ${files})
  message("  ${FOO_FILES_ITERATOR}")
endforeach()
```

Prefer instead:

```
# Foo/CMakeLists.txt

message("Files:")
foreach(x ${files})
  message("  ${x}")
endforeach()
```

See also:

- *Cache names*

Compare it with C++ code:

```
// pretty bad idea
#define a

// good one
#define MYPROJECT_ENABLE_A
```

```
// does it make sense?
for (int array_iterator = 0; array_iterator < array.size(); ++array_iterator) {
  // use 'array_iterator'
}

// good one
for (int i = 0; i < array.size(); ++i) {
  // use 'i'
}
```

Summary

- All variables have a **string** type
- List is nothing but **string**, elements of list separated by `;`
- The way how variables are interpreted **depends on the command**
- Do not give same names for **cache** and **regular** variables
- `add_subdirectory` and `function` create **new scope**
- `include` and `macro` work in the **current scope**

3.6.2 Cache variables

Cache variables saved in *CMakeCache.txt* file:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(abc "687" CACHE STRING "")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hcache-cmakecachetxt -B_builds
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
[usage-of-variables]> grep abc _builds/CMakeCache.txt
abc:STRING=687
```

No scope

Unlike regular variables CMake cache variables have no scope and are set globally:

```
# Top level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo NONE)

add_subdirectory(boo)

message("A: ${A}")
```

```
# CMakeLists.txt from 'boo' directory

set(A "123" CACHE STRING "")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hcache-no-scope -B_builds
A: 123
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Double set

If variable is already in cache then command `set(... CACHE ...)` will have no effect - old variable will be used still:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(abc "123" CACHE STRING "")
message("Variable from cache: ${abc}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cp double-set/1/CMakeLists.txt double-set/
[usage-of-variables]> cmake -Hdouble-set -B_builds
Variable from cache: 123
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
[usage-of-variables]> grep abc _builds/CMakeCache.txt
abc:STRING=123
```

Update *CMakeLists.txt* (don't remove cache!):

```
--- /examples/usage-of-variables/double-set/1/CMakeLists.txt
+++ /examples/usage-of-variables/double-set/2/CMakeLists.txt
@@ -1,5 +1,5 @@
 cmake_minimum_required(VERSION 2.8)
 project(foo NONE)

-set(abc "123" CACHE STRING "")
+set(abc "789" CACHE STRING "")
 message("Variable from cache: ${abc}")
```

```
[usage-of-variables]> cp double-set/2/CMakeLists.txt double-set/
[usage-of-variables]> cmake -Hdouble-set -B_builds
Variable from cache: 123
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
[usage-of-variables]> grep abc _builds/CMakeCache.txt
abc:STRING=123
```

-D

Cache variable can be set by `-D` command line option. Variable that set by `-D` option take priority over `set(... CACHE ...)` command.

```
[usage-of-variables]> cmake -Dabc=444 -Hdouble-set -B_builds
Variable from cache: 444
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
[usage-of-variables]> grep abc _builds/CMakeCache.txt
abc:STRING=444
```

Initial cache

If there are a lot of variables to set it's not so convenient to use `-D`. In this case user can define all variables in separate file and load it by `-C`:

```
# cache.cmake

set(A "123" CACHE STRING "")
set(B "456" CACHE STRING "")
set(C "789" CACHE STRING "")
```

```
# CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo NONE)

message("A: ${A}")
message("B: ${B}")
message("C: ${C}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -C initial-cache/cache.cmake -Hinitial-cache -B_builds
loading initial cache file initial-cache/cache.cmake
A: 123
B: 456
C: 789
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Force

If you want to set cache variable even if it's already present in cache file you can add `FORCE`:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(A "123" CACHE STRING "" FORCE)
message("A: ${A}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -DA=456 -Hforce -B_builds
A: 123
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

This is quite surprising behavior for user and conflicts with the nature of cache variables that was designed to store variable once and globally.

Warning: `FORCE` usually is an indicator of badly designed CMake code.

Force as a workaround

`FORCE` can be used to fix the problem that described *earlier*:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(A "123")
set(A "456" CACHE STRING "")

message("A: ${A}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hno-force-confuse -B_builds
A: 456
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
[usage-of-variables]> cmake -Hno-force-confuse -B_builds
A: 123
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

With `FORCE` variable will be set even it's already present in cache, so regular variable with the same name will be unset too each time:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(A "123")
set(A "456" CACHE STRING "" FORCE)

message("A: ${A}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hforce-workaround -B_builds
A: 456
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
[usage-of-variables]> cmake -Hforce-workaround -B_builds
```

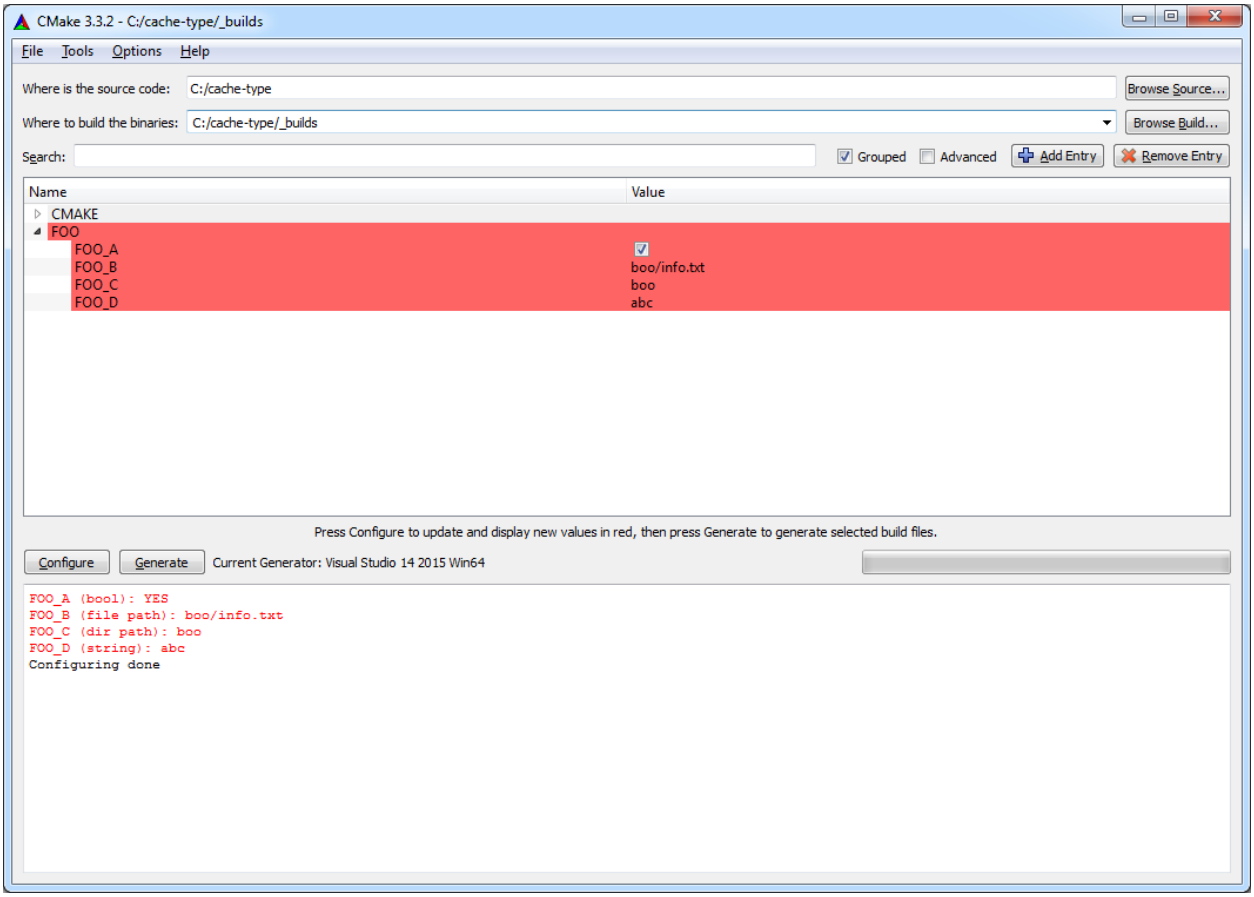
```
A: 456
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Cache type

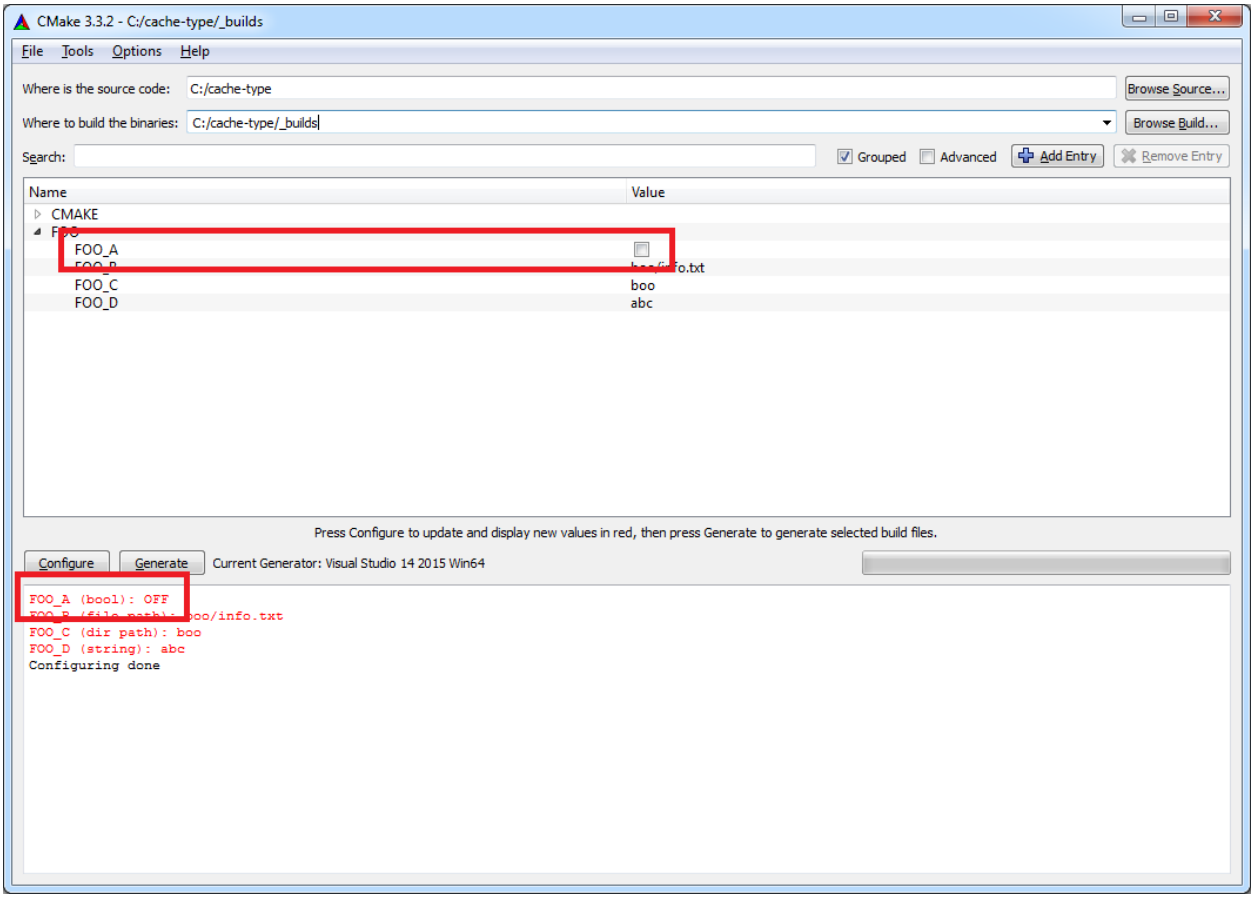
Though type of any variable is **always** string you can add some hints which will be used by CMake-GUI:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(FOO_A "YES" CACHE BOOL "Variable A")
set(FOO_B "boo/info.txt" CACHE FILEPATH "Variable B")
set(FOO_C "boo/" CACHE PATH "Variable C")
set(FOO_D "abc" CACHE STRING "Variable D")

message("FOO_A (bool): ${FOO_A}")
message("FOO_B (file path): ${FOO_B}")
message("FOO_C (dir path): ${FOO_C}")
message("FOO_D (string): ${FOO_D}")
```

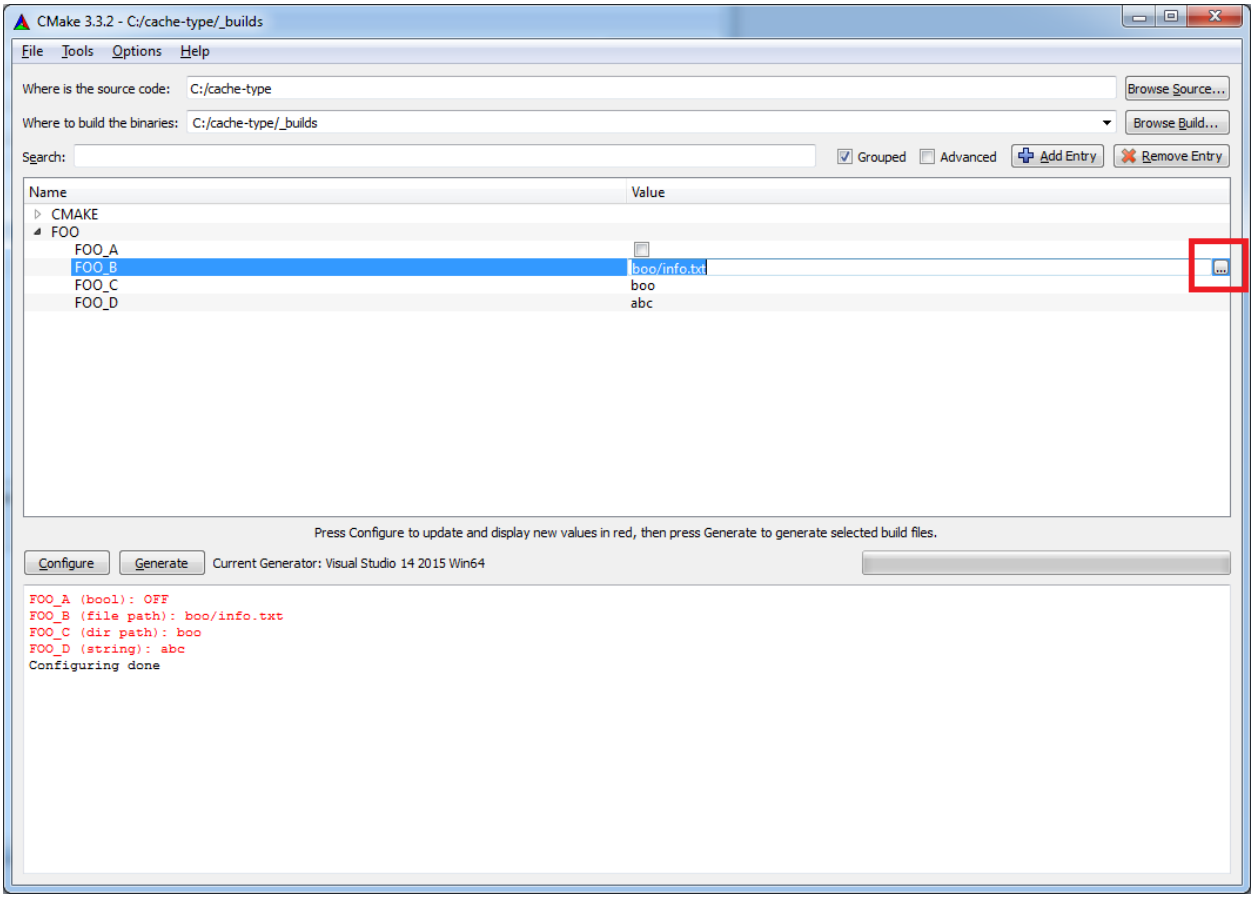
Run configure:



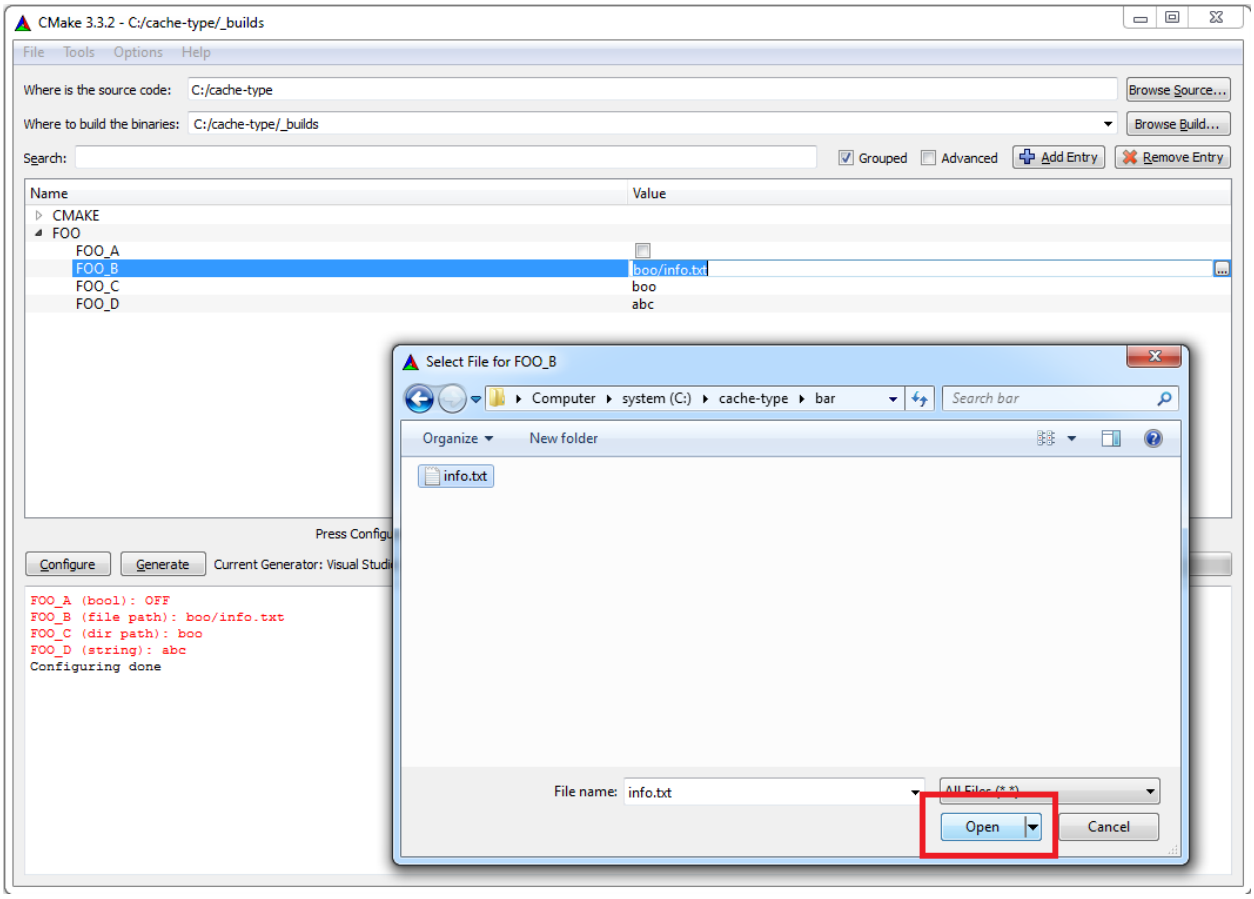
Variable `FOO_A` will be treated as boolean. Uncheck box and run configure:



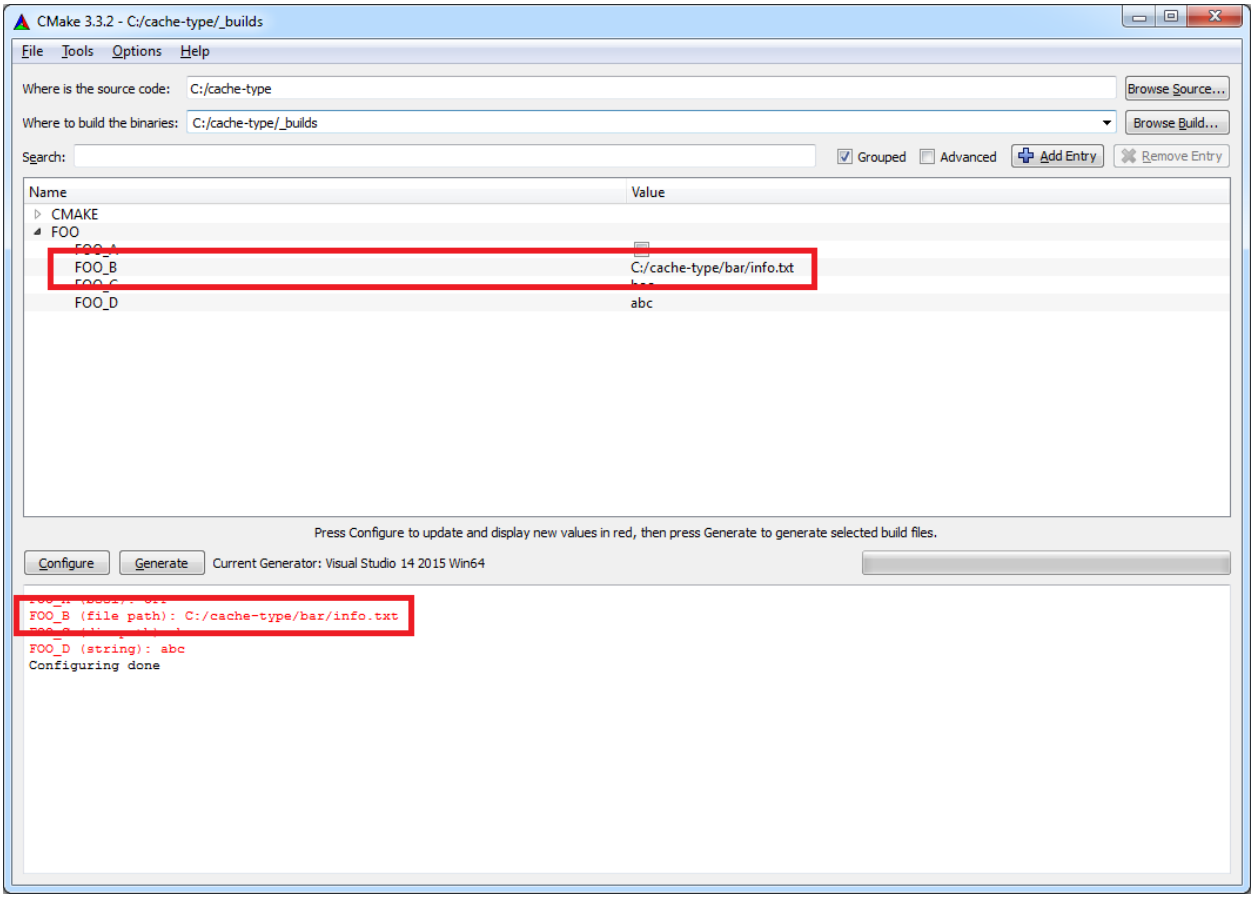
Variable `FOO_B` will be treated as path to the file. Click on `...`:


CMake 3.3.2 - C:/cache-type/_builds
File Tools Options Help
Where is the source code: C:/cache-type
Browse Source...
Where to build the binaries: C:/cache-type/_builds
Browse Build...
Search:
Grouped
Advanced
Add Entry
Remove Entry
Name
Value
CMAKE
FOO
FOO_A
FOO_B
boo/info.txt
FOO_C
boo
FOO_D
abc
Press Configure to update and display new values in red, then press Generate to generate selected build files.
Configure
Generate
Current Generator: Visual Studio 14 2015 Win64
FOO_A (bool): OFF
FOO_B (file path): boo/info.txt
FOO_C (dir path): boo
FOO_D (string): abc
Configuring done

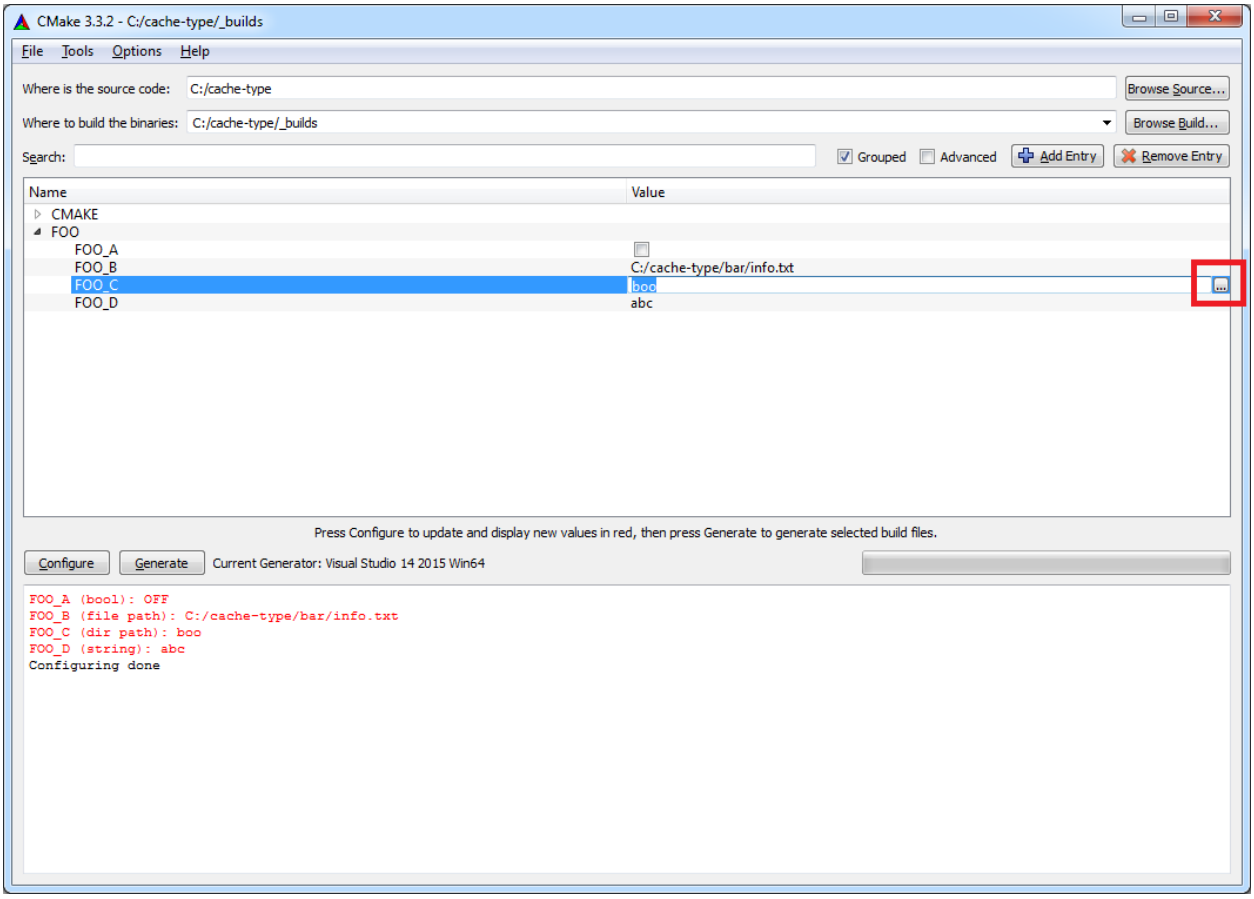
Select file:



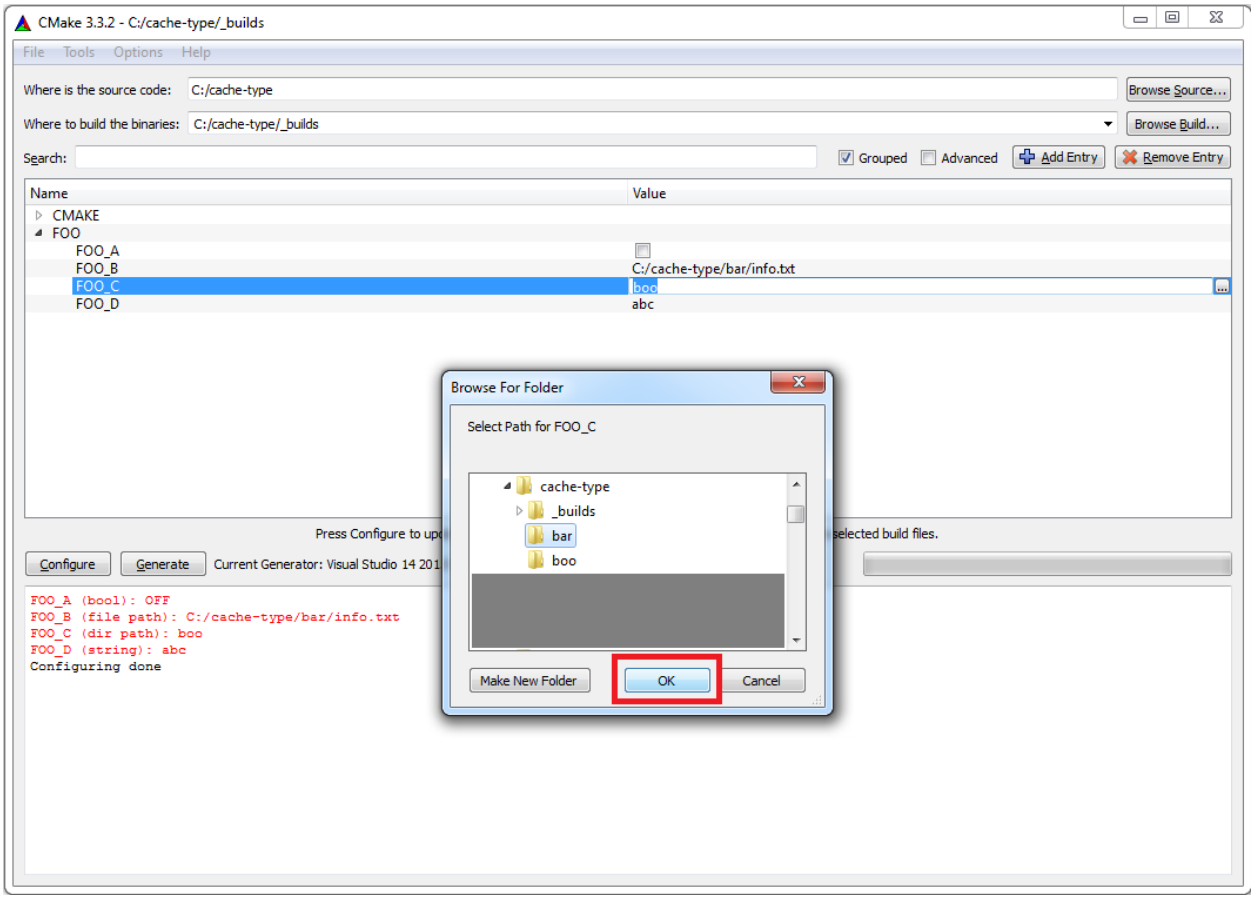
Run configure:



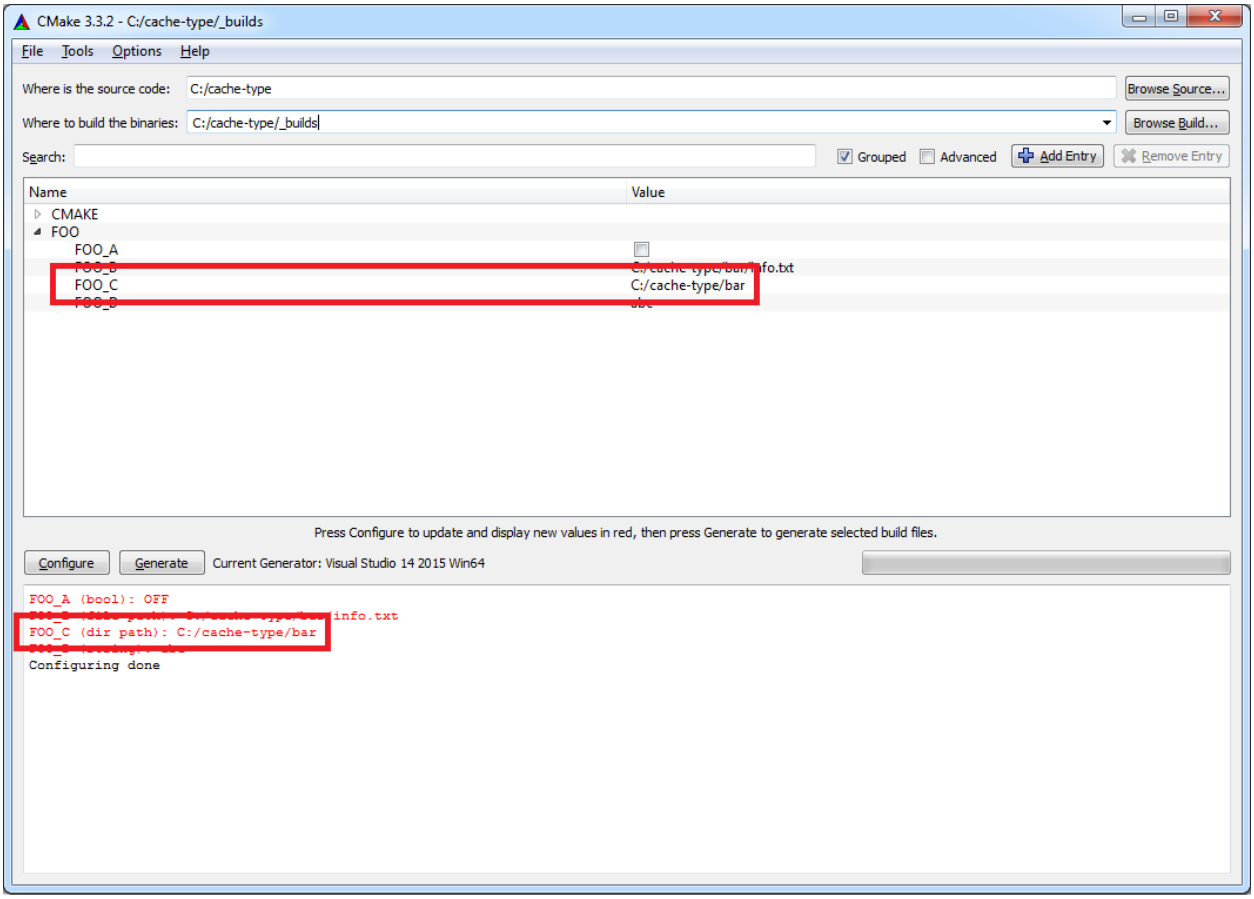
Variable `FOO_C` will be treated as path to directory. Click on `...`:



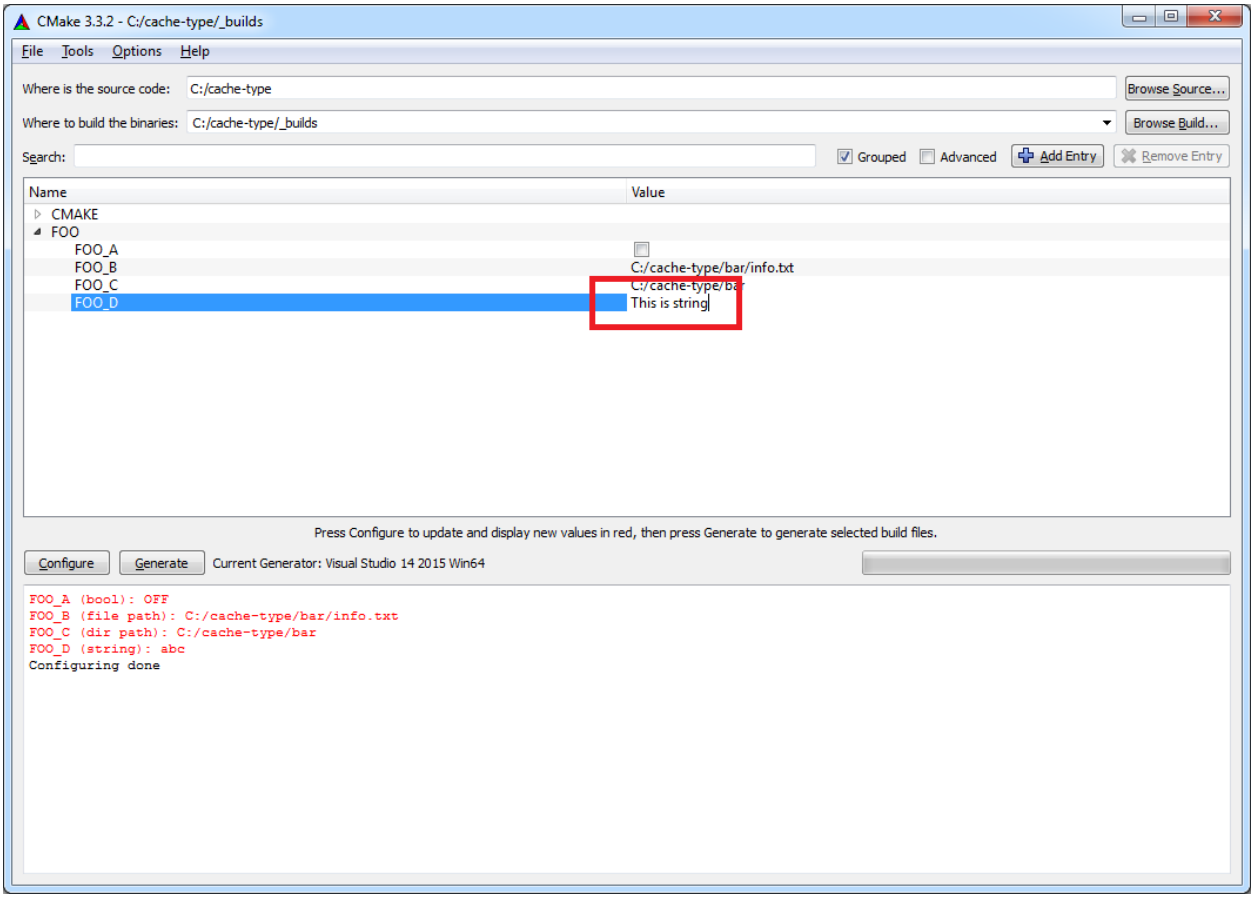
Select directory:



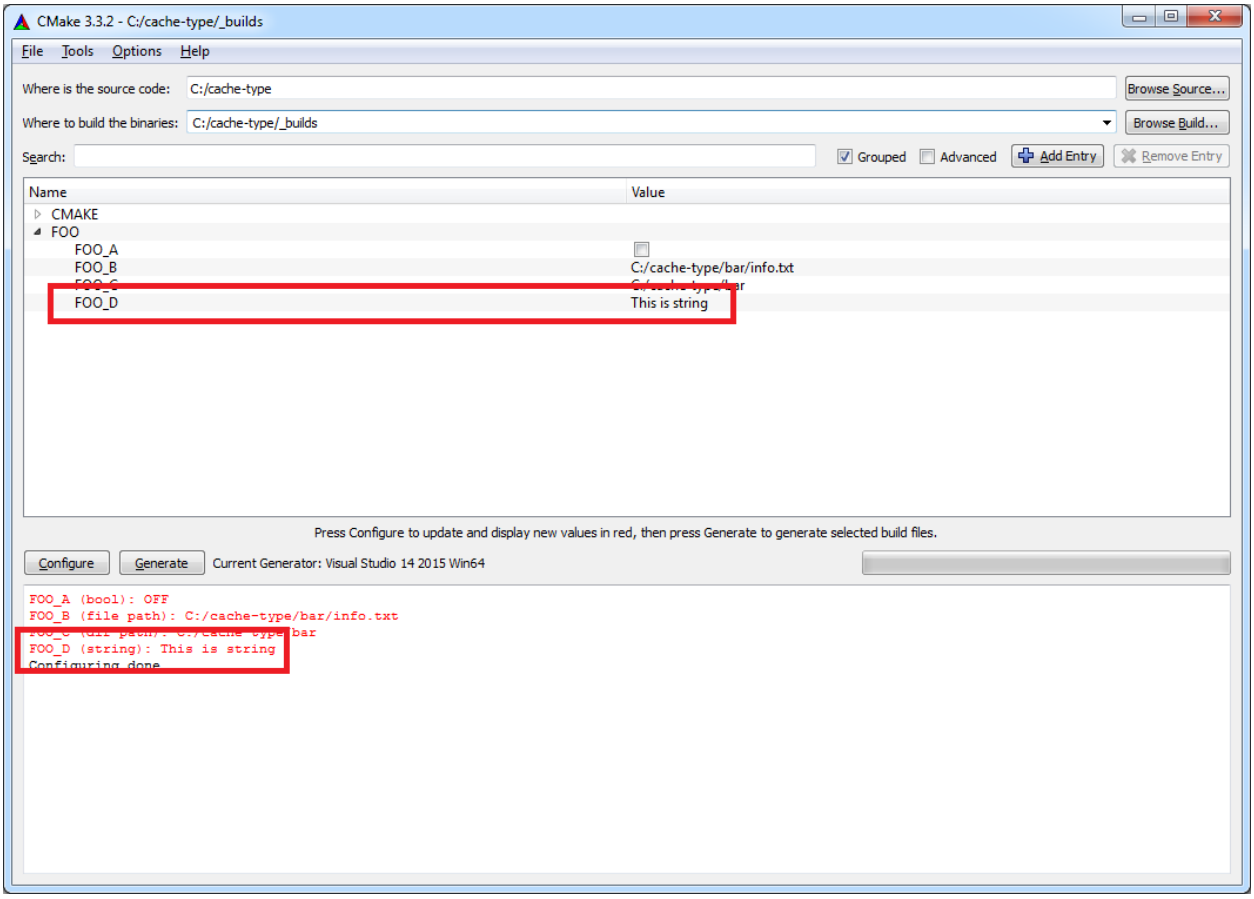
Run configure:



Variable `FOO_D` will be treated as string. Click near variable name and edit:



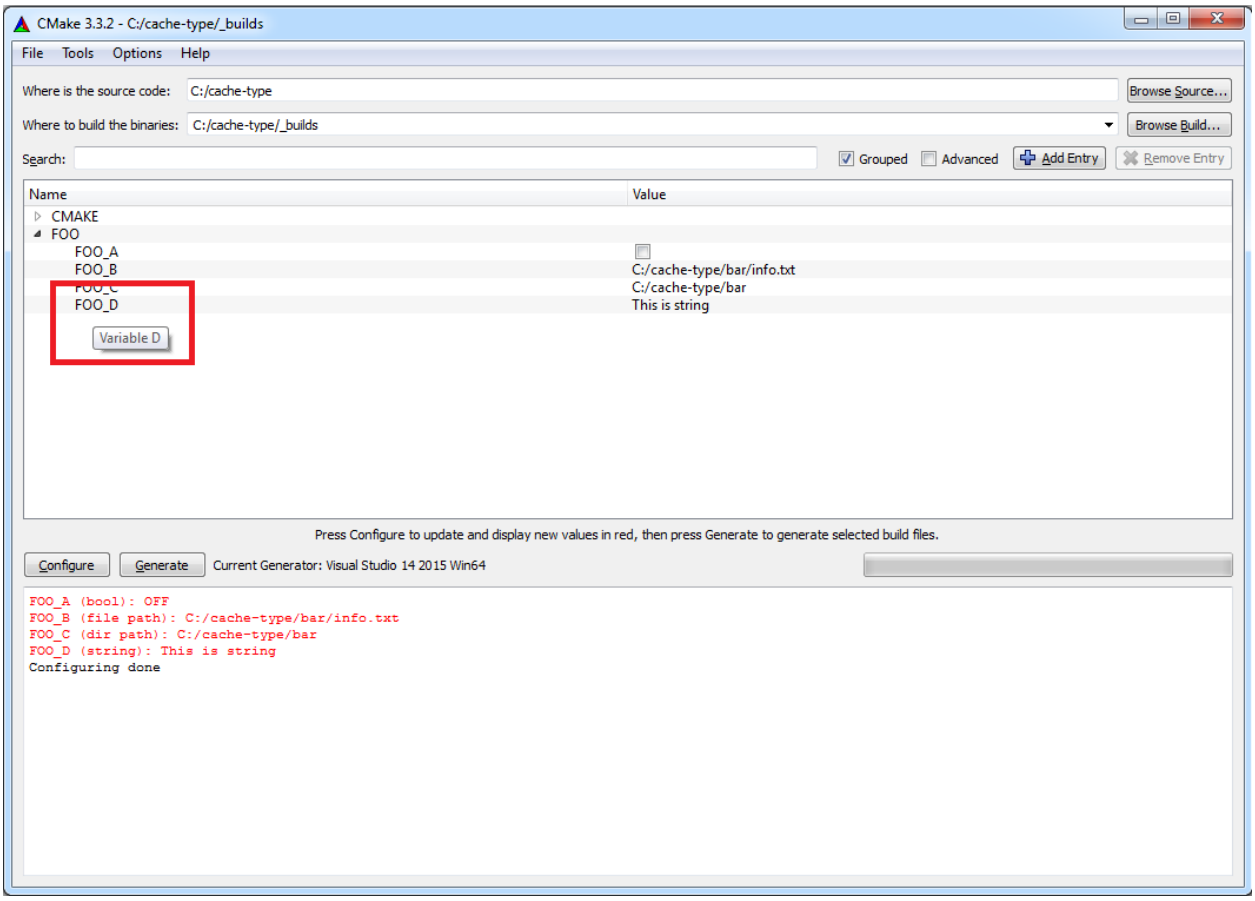
Run configure:



Description of variable:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(FOO_A "YES" CACHE BOOL "Variable A")
set(FOO_B "boo/info.txt" CACHE FILEPATH "Variable B")
set(FOO_C "boo/" CACHE PATH "Variable C")
set(FOO_D "abc" CACHE STRING "Variable D")

message("FOO_A (bool): ${FOO_A}")
message("FOO_B (file path): ${FOO_B}")
message("FOO_C (dir path): ${FOO_C}")
message("FOO_D (string): ${FOO_D}")
```

Will pop-up as a hint for users:



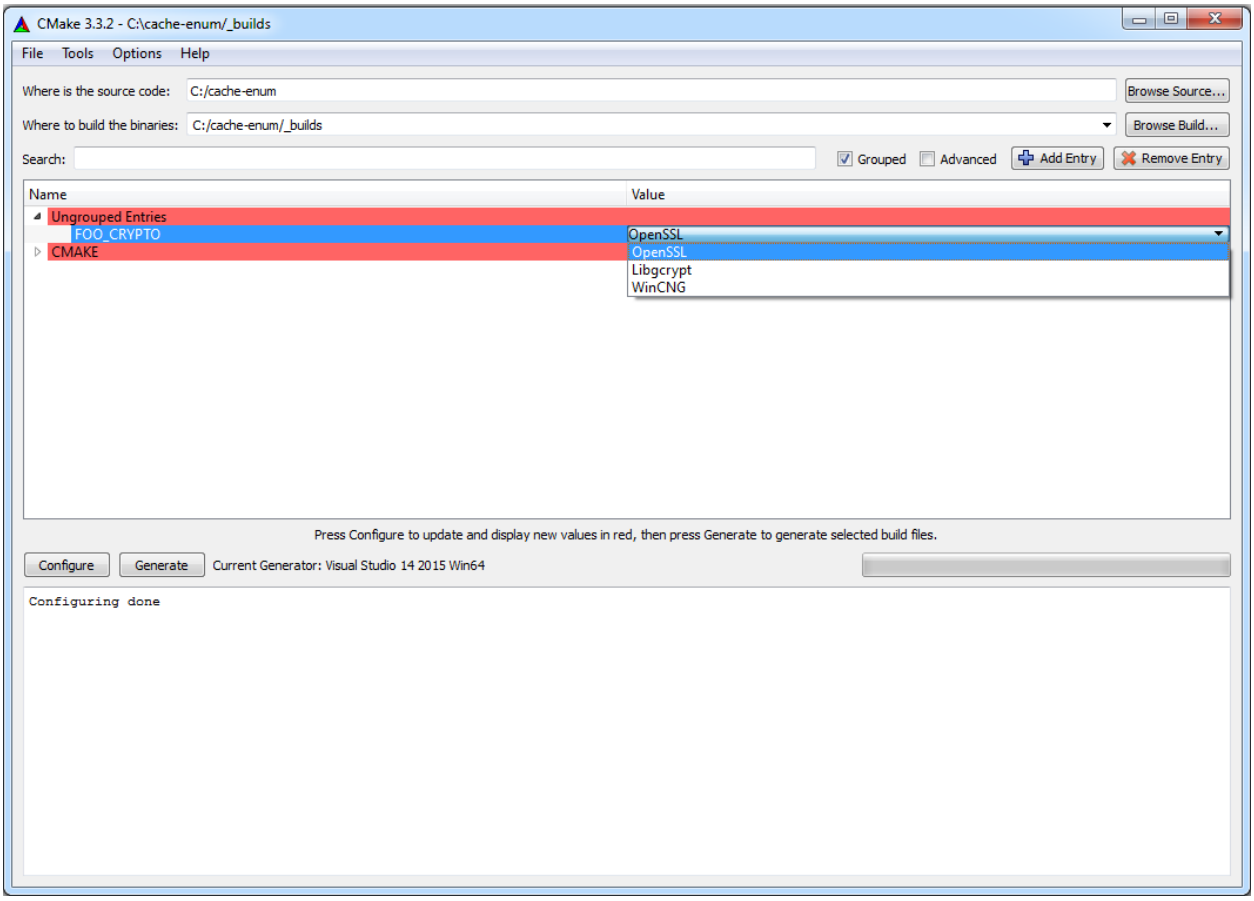
CMake documentation

- Cache entry

Enumerate

Selection widget can be created for variable of string type:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(FOO_CRYPTO "OpenSSL" CACHE STRING "Backend for cryptography")

set_property(CACHE FOO_CRYPTO PROPERTY STRINGS "OpenSSL;Libgcrypt;WinCNG")
```



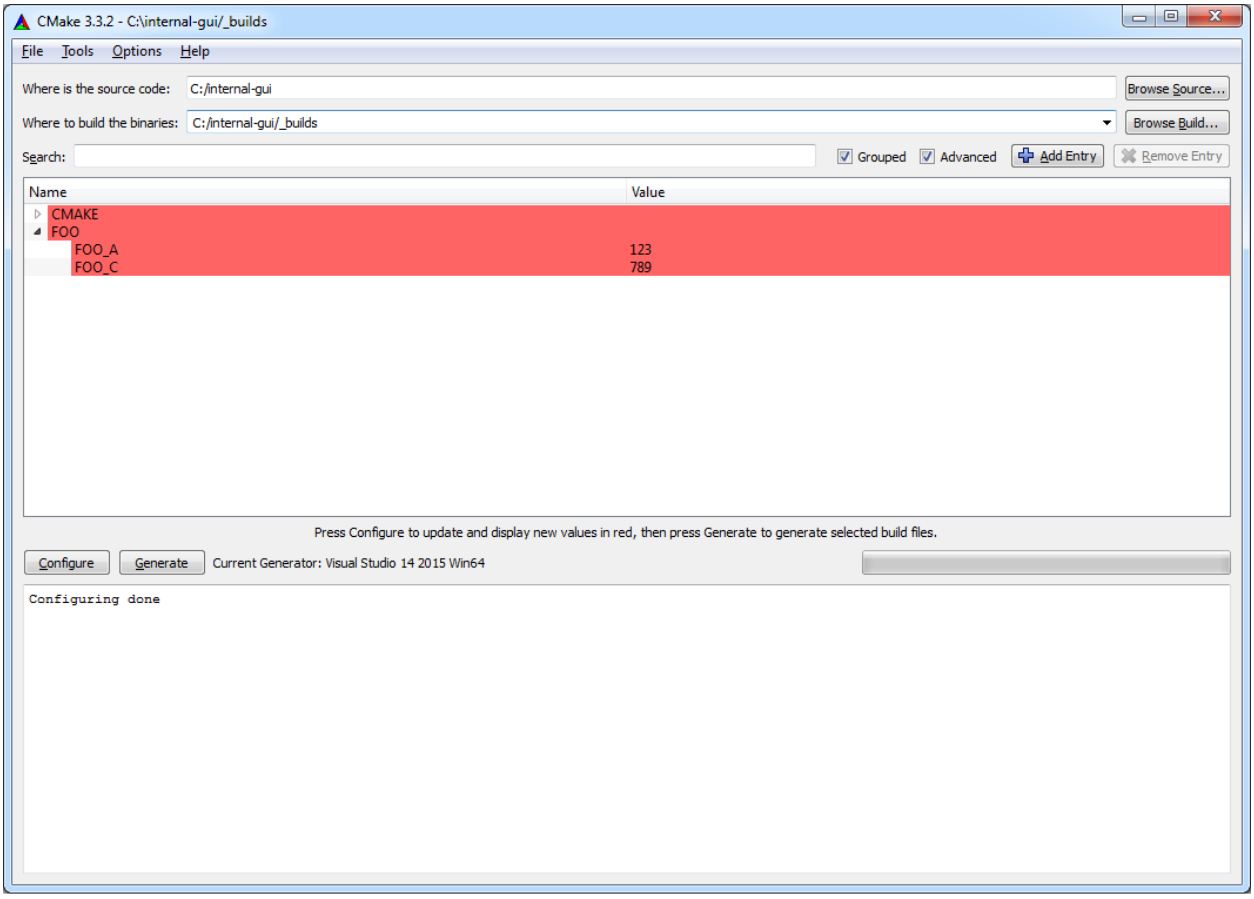
CMake documentation

- STRINGS property

Internal

Variable with type `INTERNAL` will not be shown in CMake-GUI (again, real type is a string still):

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(FOO_A "123" CACHE STRING "")
set(FOO_B "456" CACHE INTERNAL "")
set(FOO_C "789" CACHE STRING "")
```



Also such type of variable implies `FORCE`:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(FOO_A "123" CACHE INTERNAL "")
set(FOO_A "456" CACHE INTERNAL "")
set(FOO_A "789" CACHE INTERNAL "")

set(FOO_B "123" CACHE STRING "")
set(FOO_B "456" CACHE STRING "")
set(FOO_B "789" CACHE STRING "")

message("FOO_A (internal): ${FOO_A}")
message("FOO_B (string): ${FOO_B}")
```

Variable `FOO_A` will be set to `123` then **rewritten** to `456` because behavior is similar to variable **with FORCE**, then one more time to `789`, so final result is `789`. Variable `FOO_B` is a cache variable with **no FORCE** so first `123` will be set to cache, then since `FOO_B` is already in cache `456` and `789` **will be ignored**, so final result is `123`:

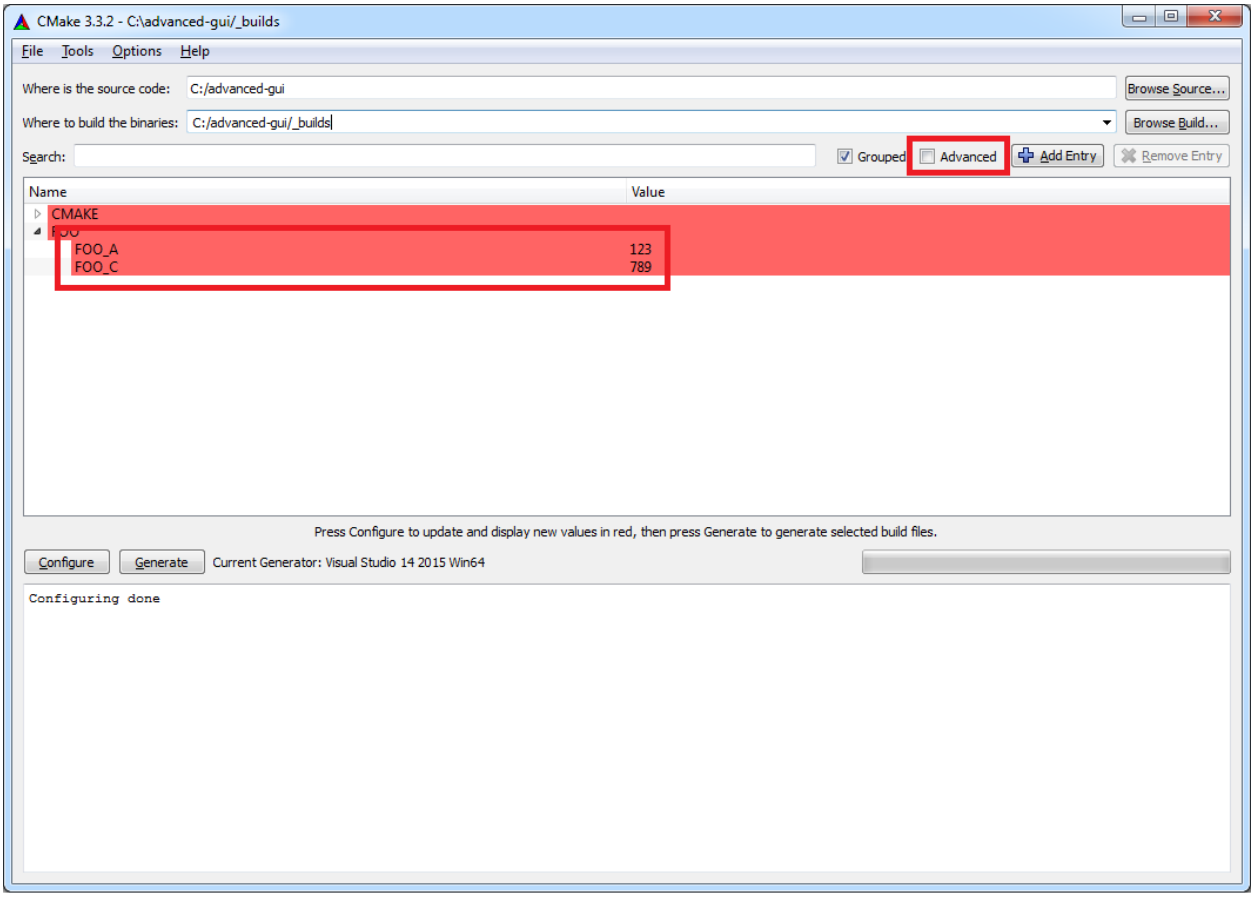
```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hinternal-force -B_builds
FOO_A (internal): 789
FOO_B (string): 123
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

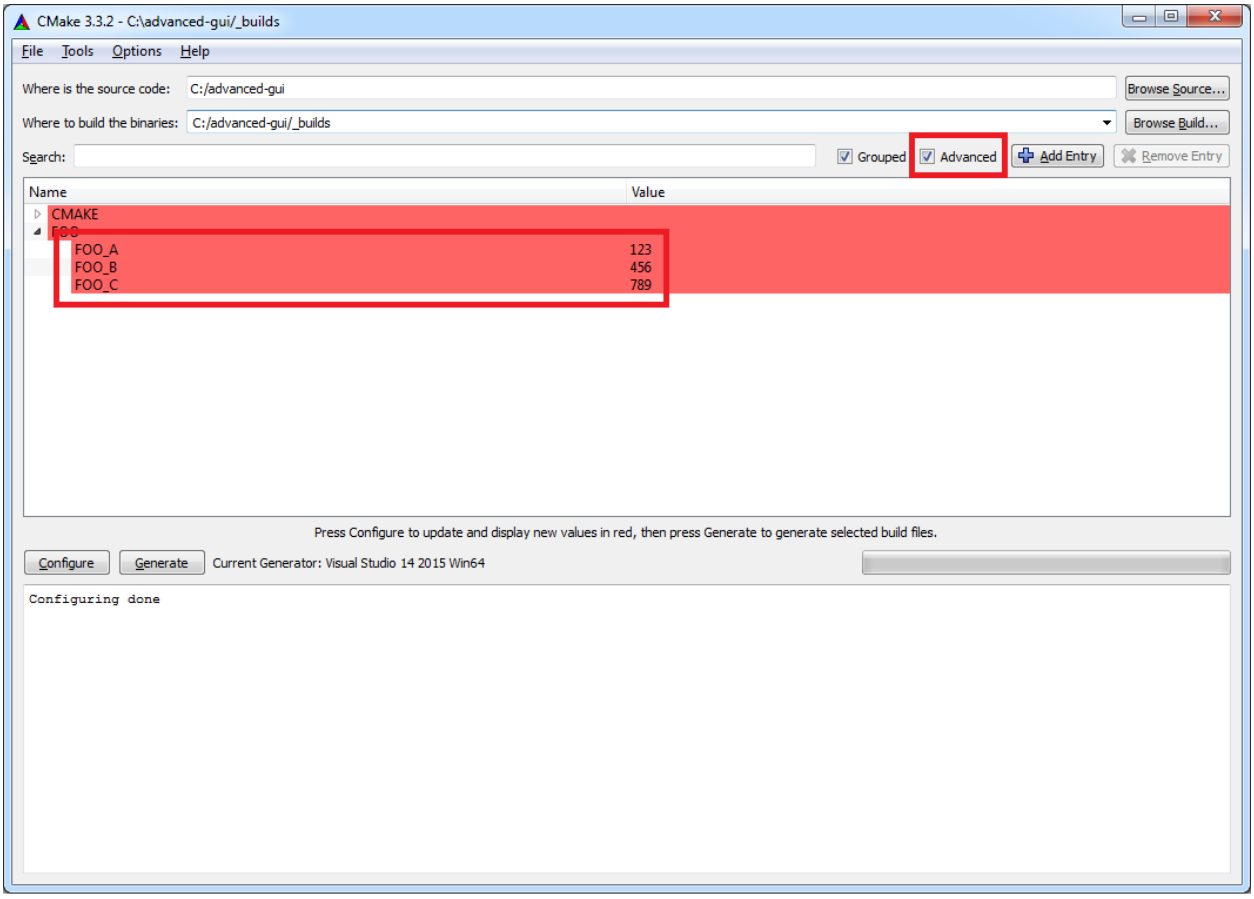
Advanced

If variable is marked as advanced:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(FOO_A "123" CACHE STRING "")
set(FOO_B "456" CACHE STRING "")
set(FOO_C "789" CACHE STRING "")

mark_as_advanced(FOO_B)
```

it will not be shown in CMake-GUI if `Advanced` checkbox is not set:





CMake documentation

- mark_as_advanced

Use case

The ability of cache variables respect user's settings fits perfectly for creating project's customization option:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(FOO_A "Default value for A" CACHE STRING "")
set(FOO_B "Default value for B")

message("FOO_A: ${FOO_A}")
message("FOO_B: ${FOO_B}")
```

Default value:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hproject-customization -B_builds
FOO_A: Default value for A
FOO_B: Default value for B
-- Configuring done
```

```
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

User's value:

```
[usage-of-variables]> cmake -DFOO_A=User -Hproject-customization -B_builds
FOO_A: User
FOO_B: Default value for B
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Note that such approach doesn't work for regular CMake variable `FOO_B`:

```
[usage-of-variables]> cmake -DFOO_B=User -Hproject-customization -B_builds
FOO_A: User
FOO_B: Default value for B
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Option

Command `option` can be used for creating boolean cache entry:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

option(FOO_A "Option A" OFF)
option(FOO_B "Option A" ON)

message("FOO_A: ${FOO_A}")
message("FOO_B: ${FOO_B}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hoption -B_builds
FOO_A: OFF
FOO_B: ON
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
[usage-of-variables]> grep FOO_ _builds/CMakeCache.txt
FOO_A:BOOL=OFF
FOO_B:BOOL=ON
```

CMake documentation

- option

Unset

If you want to remove variable `X` from cache you need to use `unset(X CACHE)`. Note that `unset(X)` will remove regular variable from current scope and have no effect on cache:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(X "123" CACHE STRING "X variable")
set(X "456")
message("[0] X = ${X}")

unset(X)
message("[1] X = ${X}")

unset(X CACHE)
message("[2] X = ${X}")

option(Y "Y option" ON)
set(Y OFF)
message("[0] Y = ${Y}")

unset(Y)
message("[1] Y = ${Y}")

unset(Y CACHE)
message("[2] Y = ${Y}")
```

When we have both cache and regular `X` variables regular variable has higher priority and will be printed:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hunset-cache -B_builds
[0] X = 456
[1] X = 123
[2] X =
[0] Y = OFF
[1] Y = ON
[2] Y =
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Command `unset(X)` will remove regular variable so cache variable will be printed:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hunset-cache -B_builds
[0] X = 456
[1] X = 123
[2] X =
[0] Y = OFF
[1] Y = ON
[2] Y =
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Command `unset(X CACHE)` will remove cache variable too. Now no variables left:

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hunset-cache -B_builds
[0] X = 456
[1] X = 123
[2] X =
```

```
[0] Y = OFF
[1] Y = ON
[2] Y =
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Since `option` do modify cache same logic applied here:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(X "123" CACHE STRING "X variable")
set(X "456")
message("[0] X = ${X}")

unset(X)
message("[1] X = ${X}")

unset(X CACHE)
message("[2] X = ${X}")

option(Y "Y option" ON)
set(Y OFF)
message("[0] Y = ${Y}")

unset(Y)
message("[1] Y = ${Y}")

unset(Y CACHE)
message("[2] Y = ${Y}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Hunset-cache -B_builds
[0] X = 456
[1] X = 123
[2] X =
[0] Y = OFF
[1] Y = ON
[2] Y =
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Recommendation

Because of the global nature of cache variables and options (well it's cache too) you should do prefix it with the name of the project to avoid clashing in case several projects are mixed together by `add_subdirectory`:

```
# top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(zoo)

add_subdirectory(boo)
add_subdirectory(foo)
```

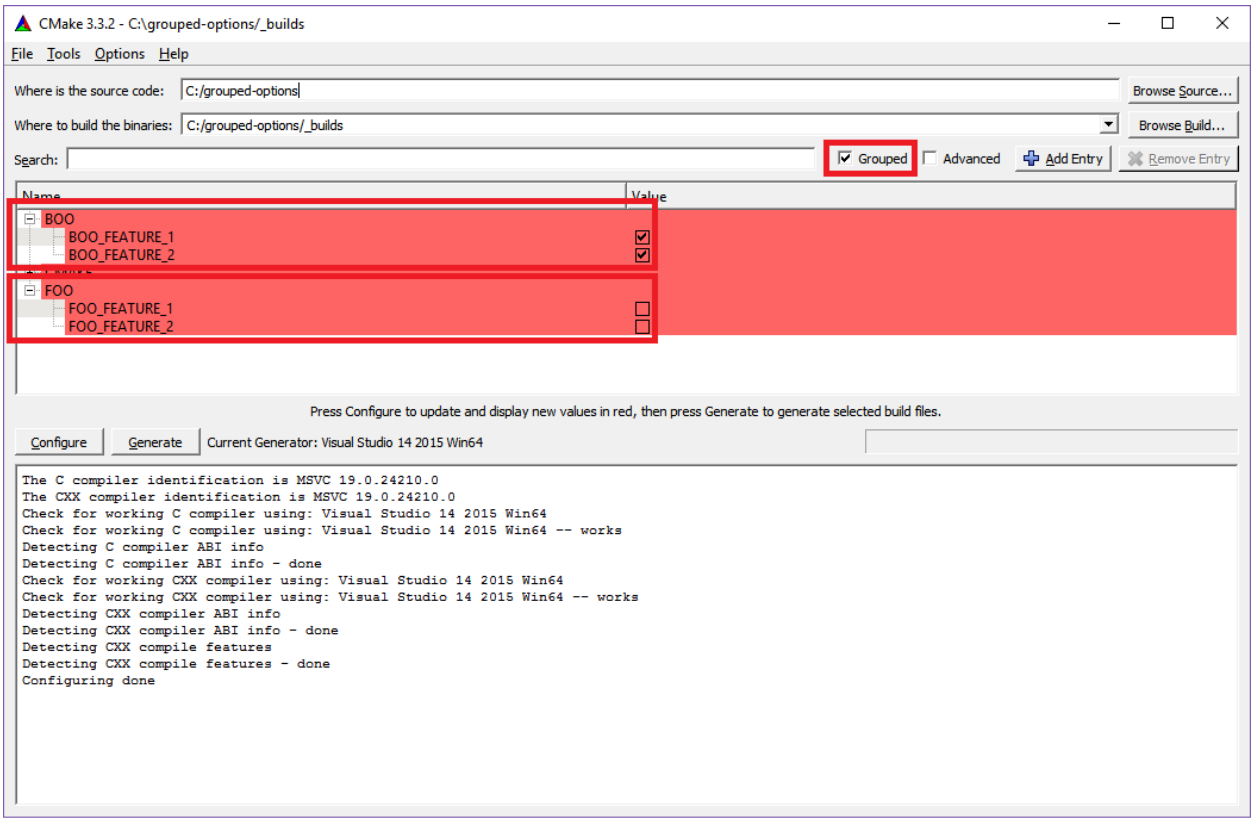
```
# foo/CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

option(FOO_FEATURE_1 "Enable feature 1" OFF)
option(FOO_FEATURE_2 "Enable feature 2" OFF)
```

```
# boo/CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(boo)

option(BOO_FEATURE_1 "Enable feature 1" ON)
option(BOO_FEATURE_2 "Enable feature 2" ON)
```

See also:

- *Module names*
- *Function names*

Besides the fact that both features can be set independently now also CMake-GUI will group them nicely:



Summary

- Use cache to set **global** variables
- Cache variables fits perfectly for expressing customized options: default value and respect user's value

- Type of cache variable helps CMake-GUI users
- Prefixes should be used to avoid clashing because of the global nature of cache variables

3.6.3 Environment variables

Read

Environment variable can be read by using `$ENV{...}` syntax:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

message("Environment variable USERNAME: $ENV{USERNAME}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> echo $USERNAME
ruslo
[usage-of-variables]> export USERNAME
[usage-of-variables]> cmake -Hread-env -B_builds
Environment variable USERNAME: ruslo
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Set

By using `set(ENV{...})` syntax CMake can set environment variable:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(ENV{USERNAME} "Jane Doe")
message("Environment variable USERNAME: $ENV{USERNAME}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> echo $USERNAME
ruslo
[usage-of-variables]> export USERNAME
[usage-of-variables]> cmake -Hset-env -B_builds
Environment variable USERNAME: Jane Doe
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Unset

Unset environment variable:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

unset(ENV{USERNAME})
message("Environment variable USERNAME: $ENV{USERNAME}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> echo $USERNAME
ruslo
[usage-of-variables]> export USERNAME
[usage-of-variables]> cmake -Hunset-env -B_builds
Environment variable USERNAME:
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Inheriting

Child process will inherit environment variables of parent:

```
# Top level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo NONE)

message("Set environment variable")

set(ENV{ABC} "This is ABC")

message("Top level ABC: $ENV{ABC}")

set(level1 "${CMAKE_CURRENT_LIST_DIR}/level1.cmake")

execute_process(
    COMMAND "${CMAKE_COMMAND}" -P "${level1}" RESULT_VARIABLE result
)

if(NOT result EQUAL 0)
  # Error
endif()

message("Unset environment variable")

unset(ENV{ABC})

message("Top level ABC: $ENV{ABC}")

execute_process(
    COMMAND "${CMAKE_COMMAND}" -P "${level1}" RESULT_VARIABLE result
)

if(NOT result EQUAL 0)
  # Error
endif()
```

```
# 'level1.cmake' script

message("Environment variable from level1: $ENV{ABC}")

set(level2 "${CMAKE_CURRENT_LIST_DIR}/level2.cmake")

execute_process(
```

```
    COMMAND "${CMAKE_COMMAND}" -P "${level2}" RESULT_VARIABLE result
)

if(NOT result EQUAL 0)
  # Error
endif()
```

```
# 'level2.cmake' script

message("Environment variable from level2: $ENV{ABC}")
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Henv-inherit -B_builds
Set environment variable
Top level ABC: This is ABC
Environment variable from level1: This is ABC
Environment variable from level2: This is ABC
Unset environment variable
Top level ABC:
Environment variable from level1:
Environment variable from level2:
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

Configure step

Note that in previous examples variable was set on *configure step*:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(ENV{ABC} "123")

message("Environment variable ABC: $ENV{ABC}")

add_custom_target(
    foo
    ALL
    "${CMAKE_COMMAND}" -P "${CMAKE_CURRENT_LIST_DIR}/script.cmake"
)
```

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> cmake -Henv-configure -B_builds
Environment variable ABC: 123
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
```

But environment variable remains the same on *build step*:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(ENV{ABC} "123")
```

```
message("Environment variable ABC: $ENV{ABC}")

add_custom_target(
    foo
    ALL
    "${CMAKE_COMMAND}" -P "${CMAKE_CURRENT_LIST_DIR}/script.cmake"
)
```

```
# script.cmake

message("Environment variable from script: $ENV{ABC}")
```

```
[usage-of-variables]> cmake --build _builds
Scanning dependencies of target foo
Environment variable from script:
Built target foo
```

No tracking

CMake doesn't track changes of used environment variables so if your CMake code depends on environment variable and you're planning to change it from time to time it will break normal *workflow*:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

set(target_name "$ENV{ABC}-tgt")
add_executable("${target_name}" foo.cpp)
```

Warning: Do not write code like that!

```
[usage-of-variables]> rm -rf _builds
[usage-of-variables]> export ABC=abc
[usage-of-variables]> cmake -Henv-depends -B_builds
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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: /.../usage-of-variables/_builds
[usage-of-variables]> cmake --build _builds
Scanning dependencies of target abc-tgt
```

```
[ 50%] Building CXX object CMakeFiles/abc-tgt.dir/foo.cpp.o
[100%] Linking CXX executable abc-tgt
[100%] Built target abc-tgt
```

Let's update environment variable:

```
[usage-of-variables]> export ABC=123
```

Name of the target **was not changed**:

```
[usage-of-variables]> cmake --build _builds
[100%] Built target abc-tgt
```

You have to run configure manually yourself:

```
[usage-of-variables]> cmake -Henv-depends -B_builds
-- Configuring done
-- Generating done
-- Build files have been written to: /.../usage-of-variables/_builds
[usage-of-variables]> cmake --build _builds
Scanning dependencies of target 123-tgt
[ 50%] Building CXX object CMakeFiles/123-tgt.dir/foo.cpp.o
[100%] Linking CXX executable 123-tgt
[100%] Built target 123-tgt
```

Summary

- CMake can set, unset and read environment variables
- Check carefully configure-build steps where you set environment variables
- Child processes will inherit environment variables of parent
- Do not make your CMake code depends on environment variable if that variable may change

3.7 CMake listfiles

There are several places where CMake code can live:

- `CMakeLists.txt` listfiles loaded by `add_subdirectory` command will help you to create source/binary tree. This is a skeleton of your project.
- `*.cmake` modules help you to organize/reuse CMake code.
- CMake scripts can be executed by `cmake -P` and help you to solve problems in cross-platform fashion without relying on system specific tools like bash or without introducing external tool dependency like Python.

Examples on GitHub

- Repository
- Latest ZIP

3.7.1 Subdirectories

Tree

CMakeLists.txt loaded by `add_subdirectory` command creates a node in a source tree:

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo NONE)

message("Top level CMakeLists.txt")

add_subdirectory(foo)
add_subdirectory(boo)
```

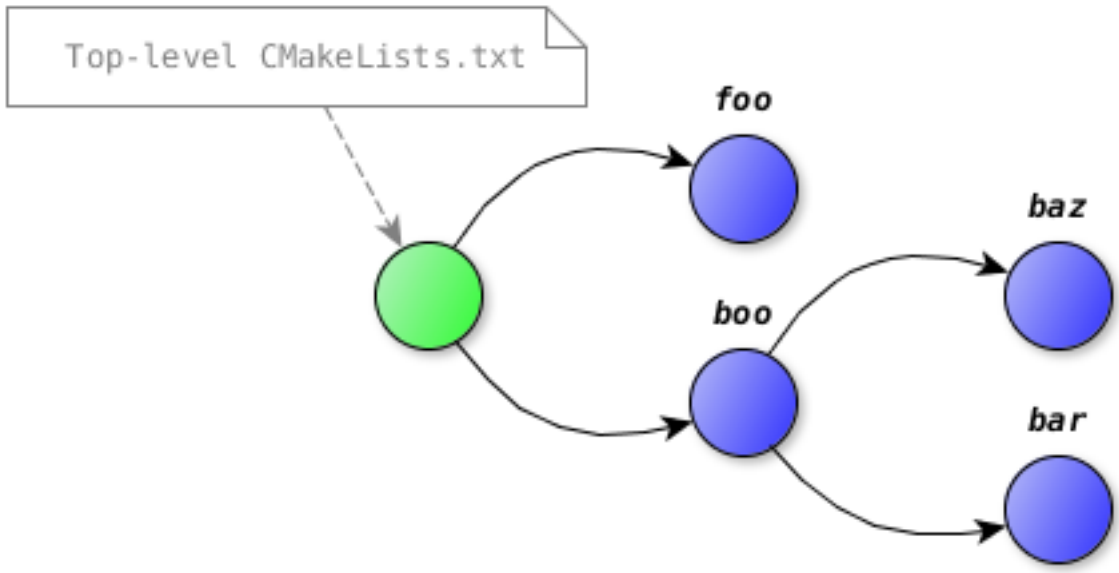
```
# foo/CMakeLists.txt

message("Processing foo/CMakeList.txt")
```

```
# boo/CMakeLists.txt

message("Processing boo/CMakeList.txt")

add_subdirectory(baz)
add_subdirectory(bar)
```

```
# boo/bar/CMakeLists.txt

message("Processing boo/bar/CMakeLists.txt")
```

```
# boo/baz/CMakeLists.txt

message("Processing boo/baz/CMakeLists.txt")
```

```
[cmake-sources]> rm -rf _builds
[cmake-sources]> cmake -Hsimple-tree -B_builds
Top level CMakeLists.txt
Processing foo/CMakeList.txt
Processing boo/CMakeList.txt
Processing boo/baz/CMakeLists.txt
Processing boo/bar/CMakeLists.txt
-- Configuring done
-- Generating done
-- Build files have been written to: /.../cmake-sources/_builds
```



Source variables

CMAKE_CURRENT_SOURCE_DIR variable will hold a full path to a currently processed node. Root of the tree is always available in CMAKE_SOURCE_DIR (see *-H*):

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo NONE)

message("Top level CMakeLists.txt")
message("CMAKE_SOURCE_DIR: ${CMAKE_SOURCE_DIR}")
message("CMAKE_CURRENT_SOURCE_DIR: ${CMAKE_CURRENT_SOURCE_DIR}")

add_subdirectory(foo)
add_subdirectory(boo)
```

```
# foo/CMakeLists.txt

message("Processing foo/CMakeList.txt")
message("CMAKE_SOURCE_DIR: ${CMAKE_SOURCE_DIR}")
message("CMAKE_CURRENT_SOURCE_DIR: ${CMAKE_CURRENT_SOURCE_DIR}")
```

```
# boo/CMakeLists.txt

message("Processing boo/CMakeList.txt")
message("CMAKE_SOURCE_DIR: ${CMAKE_SOURCE_DIR}")
message("CMAKE_CURRENT_SOURCE_DIR: ${CMAKE_CURRENT_SOURCE_DIR}")

add_subdirectory(baz)
add_subdirectory(bar)
```

```
# boo/bar/CMakeLists.txt

message("Processing boo/bar/CMakeLists.txt")
message("CMAKE_SOURCE_DIR: ${CMAKE_SOURCE_DIR}")
message("CMAKE_CURRENT_SOURCE_DIR: ${CMAKE_CURRENT_SOURCE_DIR}")
```

```
# boo/baz/CMakeLists.txt

message("Processing boo/baz/CMakeLists.txt")
message("CMAKE_SOURCE_DIR: ${CMAKE_SOURCE_DIR}")
message("CMAKE_CURRENT_SOURCE_DIR: ${CMAKE_CURRENT_SOURCE_DIR}")
```

```
[cmake-sources]> rm -rf _builds
[cmake-sources]> cmake -Hsimple-tree-source-vars -B_builds
Top level CMakeLists.txt
CMAKE_SOURCE_DIR: /.../cmake-sources/simple-tree-source-vars
CMAKE_CURRENT_SOURCE_DIR: /.../cmake-sources/simple-tree-source-vars
Processing foo/CMakeList.txt
CMAKE_SOURCE_DIR: /.../cmake-sources/simple-tree-source-vars
CMAKE_CURRENT_SOURCE_DIR: /.../cmake-sources/simple-tree-source-vars/foo
Processing boo/CMakeList.txt
CMAKE_SOURCE_DIR: /.../cmake-sources/simple-tree-source-vars
CMAKE_CURRENT_SOURCE_DIR: /.../cmake-sources/simple-tree-source-vars/boo
Processing boo/baz/CMakeLists.txt
CMAKE_SOURCE_DIR: /.../cmake-sources/simple-tree-source-vars
CMAKE_CURRENT_SOURCE_DIR: /.../cmake-sources/simple-tree-source-vars/boo/baz
Processing boo/bar/CMakeLists.txt
CMAKE_SOURCE_DIR: /.../cmake-sources/simple-tree-source-vars
CMAKE_CURRENT_SOURCE_DIR: /.../cmake-sources/simple-tree-source-vars/boo/bar
-- Configuring done
-- Generating done
-- Build files have been written to: /.../cmake-sources/_builds
```

CMake documentation

- CMAKE_SOURCE_DIR
- CMAKE_CURRENT_SOURCE_DIR

Binary tree

Same structure will be replicated in a *binary tree*. Information can be taken from `CMAKE_BINARY_DIR` (see *-B*) and `CMAKE_CURRENT_BINARY_DIR` variables:

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo NONE)

message("Top level CMakeLists.txt")
message("CMAKE_BINARY_DIR: ${CMAKE_BINARY_DIR}")
message("CMAKE_CURRENT_BINARY_DIR: ${CMAKE_CURRENT_BINARY_DIR}")

add_subdirectory(foo)
add_subdirectory(boo)
```

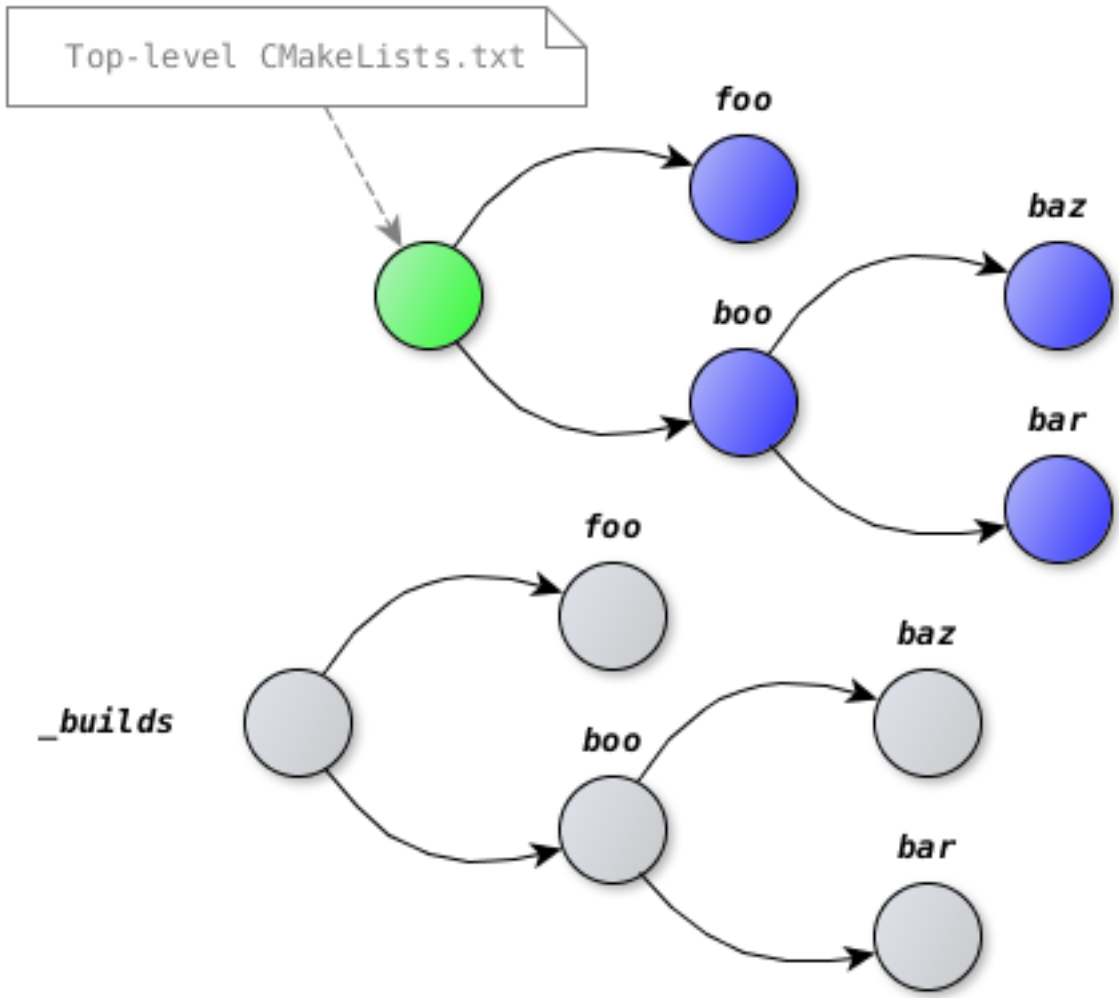
```
# foo/CMakeLists.txt

message("Processing foo/CMakeList.txt")
message("CMAKE_BINARY_DIR: ${CMAKE_BINARY_DIR}")
message("CMAKE_CURRENT_BINARY_DIR: ${CMAKE_CURRENT_BINARY_DIR}")
```

```
# boo/CMakeLists.txt

message("Processing boo/CMakeList.txt")
message("CMAKE_BINARY_DIR: ${CMAKE_BINARY_DIR}")
message("CMAKE_CURRENT_BINARY_DIR: ${CMAKE_CURRENT_BINARY_DIR}")

add_subdirectory(baz)
add_subdirectory(bar)
```

```
# boo/bar/CMakeLists.txt

message("Processing boo/bar/CMakeLists.txt")
message("CMAKE_BINARY_DIR: ${CMAKE_BINARY_DIR}")
message("CMAKE_CURRENT_BINARY_DIR: ${CMAKE_CURRENT_BINARY_DIR}")
```

```
# boo/baz/CMakeLists.txt

message("Processing boo/baz/CMakeLists.txt")
message("CMAKE_BINARY_DIR: ${CMAKE_BINARY_DIR}")
message("CMAKE_CURRENT_BINARY_DIR: ${CMAKE_CURRENT_BINARY_DIR}")
```

```
[cmake-sources]> rm -rf _builds
[cmake-sources]> cmake -Hsimple-tree-binary-vars -B_builds
Top level CMakeLists.txt
CMAKE_BINARY_DIR: /.../cmake-sources/_builds
CMAKE_CURRENT_BINARY_DIR: /.../cmake-sources/_builds
Processing foo/CMakeList.txt
CMAKE_BINARY_DIR: /.../cmake-sources/_builds
CMAKE_CURRENT_BINARY_DIR: /.../cmake-sources/_builds/foo
Processing boo/CMakeList.txt
CMAKE_BINARY_DIR: /.../cmake-sources/_builds
CMAKE_CURRENT_BINARY_DIR: /.../cmake-sources/_builds/boo
Processing boo/baz/CMakeLists.txt
CMAKE_BINARY_DIR: /.../cmake-sources/_builds
CMAKE_CURRENT_BINARY_DIR: /.../cmake-sources/_builds/boo/baz
Processing boo/bar/CMakeLists.txt
CMAKE_BINARY_DIR: /.../cmake-sources/_builds
CMAKE_CURRENT_BINARY_DIR: /.../cmake-sources/_builds/boo/bar
-- Configuring done
-- Generating done
-- Build files have been written to: /.../cmake-sources/_builds
```



See also:

- *Project variables*

CMake documentation

- CMAKE_BINARY_DIR
- CMAKE_CURRENT_BINARY_DIR

3.7.2 Include modules

CMake modules is a common way to reuse code.

Include standard

CMake comes with a set of standard modules:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

include(ProcessorCount)

ProcessorCount(N)
message("Number of processors: ${N}")
```

```
[cmake-sources]> rm -rf _builds
[cmake-sources]> cmake -Hinclude-processor-count -B_builds
Number of processors: 4
-- Configuring done
-- Generating done
-- Build files have been written to: /.../cmake-sources/_builds
```

CMake documentation

- ProcessorCount

Warning: Do not include `Find*.cmake` modules such way. `Find*.cmake` modules designed to be used via find_package.

Include custom

You can modify a `CMAKE_MODULE_PATH` variable to add the path with your custom CMake modules:

```
# Top level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo NONE)

list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_LIST_DIR}/modules")

include(MyModule)
```

```
# modules/MyModule.cmake

message("Hello from MyModule!")
```

```
[cmake-sources]> rm -rf _builds
[cmake-sources]> cmake -Hinclude-users -B_builds
Hello from MyModule!
-- Configuring done
-- Generating done
-- Build files have been written to: /.../cmake-sources/_builds
```

CMake documentation

- CMAKE_MODULE_PATH

Recommendation

To avoid conflicts of your modules with modules from other projects (if they are mixed together by `add_subdirectory`) do "namespace" their names with the project name:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_LIST_DIR}/cmake/Modules")

include(tool_verifier) # BAD! What if a parent project already has 'tool_verifier'?

include(foo_tool_verifier) # Good, includes "./cmake/Modules/foo_tool_verifier.cmake"
```

See also:

- OpenCV modules

See also:

- *Function names*
- *Cache names*

Modify correct

Note that the correct way to set this path is to **append** it to an existing value:

```
# Top level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo NONE)

list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_LIST_DIR}/modules")

include(ProcessorCount)

ProcessorCount(N)
message("Number of processors: ${N}")
```

For example when a user wants to use his own modules instead of standard for any reason:

```
# standard/ProcessorCount.cmake

function(ProcessorCount varname)
  message("Force processor count")
  set("${varname}" 16 PARENT_SCOPE)
endfunction()
```

Works fine:

```
[cmake-sources]> rm -rf _builds
[cmake-sources]> cmake -Hmodify-path -B_builds "-DCMAKE_MODULE_PATH=`pwd`/modify-path/
↪standard"
Force processor count
Number of processors: 16
-- Configuring done
```

```
-- Generating done
-- Build files have been written to: /.../cmake-sources/_builds
```

Modify incorrect

It's not correct to set them ignoring current state:

```
# Top level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(CMAKE_MODULE_PATH "${CMAKE_CURRENT_LIST_DIR}/modules") # WRONG!

include(ProcessorCount)

ProcessorCount(N)
message("Number of processors: ${N}")
```

In this case if user want to use custom modules:

```
# standard/ProcessorCount.cmake

function(ProcessorCount varname)
  message("Force processor count")
  set("${varname}" 16 PARENT_SCOPE)
endfunction()
```

They will **not** be loaded:

```
[cmake-sources]> rm -rf _builds
[cmake-sources]> cmake -Hmodify-incorrect -B_builds "-DCMAKE_MODULE_PATH=`pwd`/modify-
↪incorrect/standard"
Number of processors: 4
-- Configuring done
-- Generating done
-- Build files have been written to: /.../cmake-sources/_builds
```

3.7.3 Common variables

Since every `CMakeLists.txt` is a *listfile* hence the common listfile variables like `CMAKE_CURRENT_LIST_DIR` or `CMAKE_CURRENT_LIST_FILE` are available. For `CMakeLists.txt` added by `add_subdirectory` there will be no difference between `CMAKE_CURRENT_LIST_DIR` and `CMAKE_CURRENT_SOURCE_DIR`, also `CMAKE_CURRENT_LIST_FILE` will be always a full path to `CMakeLists.txt`. However it's not always true for other types of CMake listfiles.

CMake documentation

- CMAKE_CURRENT_LIST_DIR
- CMAKE_CURRENT_LIST_FILE
- CMAKE_CURRENT_LIST_LINE

CMAKE_CURRENT_LIST_*

Information about any kind of listfile can be taken from `CMAKE_CURRENT_LIST_FILE` and `CMAKE_CURRENT_LIST_DIR` variables:

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo NONE)

list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_LIST_DIR}/cmake")

include(mymodule)
```

```
# cmake/mymodule.cmake

message("Full path to module: ${CMAKE_CURRENT_LIST_FILE}")
message("Module located in directory: ${CMAKE_CURRENT_LIST_DIR}")
```

```
[cmake-sources]> rm -rf _builds
[cmake-sources]> cmake -Hpath-to-module -B_builds
Full path to module: /.../cmake-sources/path-to-module/cmake/mymodule.cmake
Module located in directory: /.../cmake-sources/path-to-module/cmake
-- Configuring done
-- Generating done
-- Build files have been written to: /.../cmake-sources/_builds
```

CMAKE_CURRENT_LIST_DIR vs CMAKE_CURRENT_SOURCE_DIR

The difference between those two variables is about type of information they provide. A `CMAKE_CURRENT_SOURCE_DIR` variable describes **a source tree** and should be read as *a current source tree directory*. Here is a list of sibling variables describing source/binary trees:

- CMAKE_SOURCE_DIR
- CMAKE_BINARY_DIR
- PROJECT_SOURCE_DIR
- PROJECT_BINARY_DIR
- **CMAKE_CURRENT_SOURCE_DIR**
- CMAKE_CURRENT_BINARY_DIR

The next files **always** exist:

- `${CMAKE_SOURCE_DIR}/CMakeLists.txt`
- `${CMAKE_BINARY_DIR}/CMakeCache.txt`
- `${PROJECT_SOURCE_DIR}/CMakeLists.txt`
- `${CMAKE_CURRENT_SOURCE_DIR}/CMakeLists.txt`

A `CMAKE_CURRENT_LIST_DIR` variable describes **a current listfile** (it is not necessarily `CMakeLists.txt`, it can be `somemodule.cmake`), and should be read as *a directory of a currently processed listfile*, i.e. directory of `CMAKE_CURRENT_LIST_FILE`. Here is another list of sibling variables:

- CMAKE_CURRENT_LIST_FILE

- CMAKE_CURRENT_LIST_LINE
- **CMAKE_CURRENT_LIST_DIR**
- CMAKE_PARENT_LIST_FILE

Example

Assume we have an external CMake module that calculates SHA1 of CMakeLists.txt and saves it with some custom info to a `sha1` file in a current binary directory:

```
# External module: mymodule.cmake

file(READ "${CMAKE_CURRENT_LIST_DIR}/info/message.txt" _mymodule_message)
file(SHA1 "${CMAKE_CURRENT_SOURCE_DIR}/CMakeLists.txt" _mymodule_cmakelists_sha1)
file(
    WRITE
    "${CMAKE_CURRENT_BINARY_DIR}/sha1"
    "${_mymodule_message}\nsha1(CMakeLists.txt) = ${_mymodule_cmakelists_sha1}\n"
)
```

`mymodule.cmake` uses some resource. Resource `info/message.txt` is a file with content:

```
Message from external module
```

To read this resource we must use `CMAKE_CURRENT_LIST_DIR` because file located **in same external directory** as module:

```
# External module: mymodule.cmake

file(READ "${CMAKE_CURRENT_LIST_DIR}/info/message.txt" _mymodule_message)
file(SHA1 "${CMAKE_CURRENT_SOURCE_DIR}/CMakeLists.txt" _mymodule_cmakelists_sha1)
file(
    WRITE
    "${CMAKE_CURRENT_BINARY_DIR}/sha1"
    "${_mymodule_message}\nsha1(CMakeLists.txt) = ${_mymodule_cmakelists_sha1}\n"
)
```

To read CMakeLists.txt we must use `CMAKE_CURRENT_SOURCE_DIR` because CMakeLists.txt located **in source directory**:

```
# External module: mymodule.cmake

file(READ "${CMAKE_CURRENT_LIST_DIR}/info/message.txt" _mymodule_message)
file(SHA1 "${CMAKE_CURRENT_SOURCE_DIR}/CMakeLists.txt" _mymodule_cmakelists_sha1)
file(
    WRITE
    "${CMAKE_CURRENT_BINARY_DIR}/sha1"
    "${_mymodule_message}\nsha1(CMakeLists.txt) = ${_mymodule_cmakelists_sha1}\n"
)
```

Subdirectory `boo` uses this module:

```
# boo/CMakeLists.txt

message("Processing boo/CMakeList.txt")

add_subdirectory(baz)
```

```
add_subdirectory(bar)

include(mymodule)
```

```
[cmake-sources]> rm -rf _builds
[cmake-sources]> cmake -Hwith-external-module/example -B_builds -DCMAKE_MODULE_
↪PATH=`pwd`/with-external-module/external
Top level CMakeLists.txt
Processing foo/CMakeList.txt
Processing boo/CMakeList.txt
Processing boo/baz/CMakeLists.txt
Processing boo/bar/CMakeLists.txt
-- Configuring done
-- Generating done
-- Build files have been written to: /.../cmake-sources/_builds
```

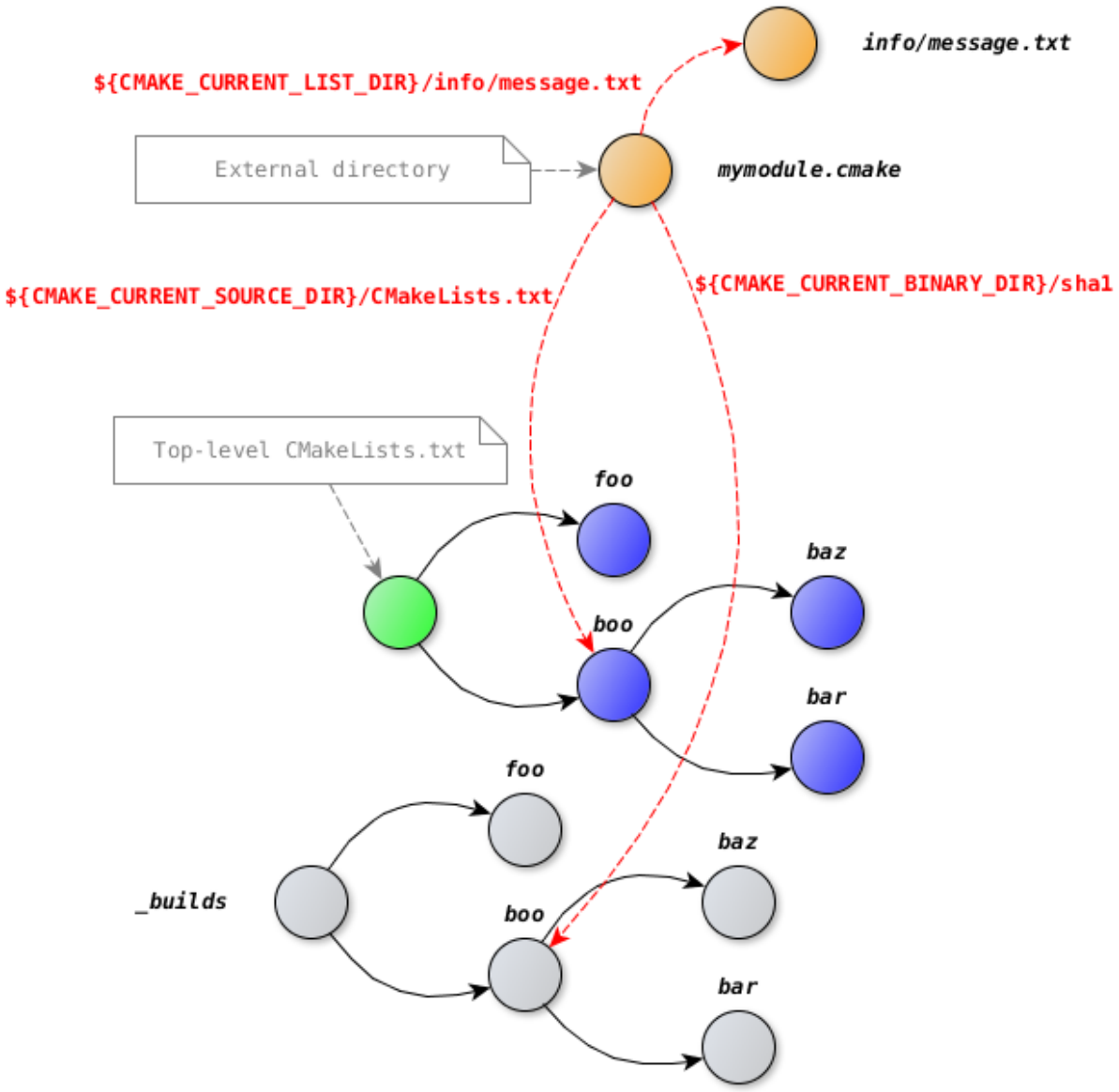
Check a `sha1` file created by the module:

```
[cmake-sources]> cat _builds/boo/sha1
Message from external module

sha1(CMakeLists.txt) = 9f0ceda4ca514a074589fc7591aad0635b6565eb
```

Verify a value manually:

```
[cmake-sources]> openssl sha1 with-external-module/example/boo/CMakeLists.txt
SHA1(with-external-module/example/boo/CMakeLists.txt)=␣
↪9f0ceda4ca514a074589fc7591aad0635b6565eb
```

This diagram will make everything clear:



Recommendation

Instead of keeping in a head all this information you can remember just two variables:

- `CMAKE_CURRENT_LIST_DIR`
- `CMAKE_CURRENT_BINARY_DIR`

Note that *in functions* a `CMAKE_CURRENT_LIST_DIR` variable is set to the directory where a function **is used**, not where a function **is defined** (see *function* for details).

Use `CMAKE_CURRENT_BINARY_DIR` for storing generated files.

> **Warning:** Do not use `CMAKE_CURRENT_BINARY_DIR` for figuring out the full path to objects that was build by native tool, e.g. using `${CMAKE_CURRENT_BINARY_DIR}/foo.exe` is a bad idea since for Linux exe-

cutable will be named `${CMAKE_CURRENT_BINARY_DIR}/foo` and for multi-configuration generators it will be like `${CMAKE_CURRENT_BINARY_DIR}/Debug/foo.exe` and really should be determined on a build step instead of generate step. In such cases *generator expressions* is helpful. For example $<TARGET_FILE:tgt>.

Make sure you **fully understand** what each variable means in other scenarios:

- `CMAKE_SOURCE_DIR`/`CMAKE_BINARY_DIR` these variables point to the root of the source/binary trees. If your project will be added to another project as a subproject by `add_subdirectory`, the locations like `${CMAKE_SOURCE_DIR}/my-resource.txt` will point to `<top-level>/my-resource.txt` instead of `<my-project>/my-resource.txt`
- `PROJECT_SOURCE_DIR`/`PROJECT_BINARY_DIR` these variables are better then previous but still have kind of a global nature. You should change all paths related to `PROJECT_SOURCE_DIR` if you decide to move declaration of your project or decide to detach some part of the code and add new `project` command in the middle of the source tree. Consider using extra variable with clean separate purpose for such job `set(FOO_MY_RESOURCES "${CMAKE_CURRENT_LIST_DIR}/resources")` instead of referring to `${PROJECT_SOURCE_DIR}/resources`.
- `CMAKE_CURRENT_SOURCE_DIR` this is a directory with `CMakeLists.txt`. If you're using this variable internally you can substitute is with `CMAKE_CURRENT_LIST_DIR`. In case you're creating module for external usage consider moving all functionality to `function`.

With this recommendation previous example can be rewritten in next way:

```
# External module: mymodule.cmake

# This is not a part of the function so 'CMAKE_CURRENT_LIST_DIR' is the path
# to the directory with 'mymodule.cmake'.
set(MYMODULE_PATH_TO_INFO "${CMAKE_CURRENT_LIST_DIR}/info/message.txt")

function(mymodule)
  # When we are inside function 'CMAKE_CURRENT_LIST_DIR' is the path to the
  # caller, i.e. path to directory with CMakeLists.txt in our case.
  file(SHA1 "${CMAKE_CURRENT_LIST_DIR}/CMakeLists.txt" sha1)

  file(READ "${MYMODULE_PATH_TO_INFO}" msg)
  file(
      WRITE
      "${CMAKE_CURRENT_BINARY_DIR}/sha1"
      "${msg}\nsha1(CMakeLists.txt) = ${sha1}\n"
  )
endfunction()
```

Note: As you may notice we don't have to use `_long_variable` names since function has it's own scope.

And call a `mymodule` function instead of including a module:

```
# boo/CMakeLists.txt

message("Processing boo/CMakeList.txt")

add_subdirectory(baz)
add_subdirectory(bar)

mymodule()
```

Effect is the same:

```
[cmake-sources]> cat _builds/boo/sha1
Message from external module
sha1(CMakeLists.txt) = 36bcbf5f2f23995661ca4e6349e781160910b71f

[cmake-sources]> openssl sha1 with-external-module-good/example/boo/CMakeLists.txt
SHA1(with-external-module-good/example/boo/CMakeLists.txt)=␣
↪36bcbf5f2f23995661ca4e6349e781160910b71f
```

3.7.4 Scripts

CMake can be used as a cross-platform scripting language.

CMake documentation

- CMake options

Example

Script for creating a file:

```
# create-file.cmake

file(WRITE Hello.txt "Created by script")
```

Run the script by `cmake -P`:

```
[cmake-sources]> rm -f Hello.txt
[cmake-sources]> cmake -P script/create-file.cmake
[cmake-sources]> ls Hello.txt
Hello.txt
[cmake-sources]> cat Hello.txt
Created by script
```

Minimum required (bad)

We should use `cmake_minimum_required` as the first command in a script just like with the *regular CMakeLists.txt*. Lack of `cmake_minimum_required` may lead to problems:

```
# script.cmake

set("Jane Doe" "")
set(MYNAME "Jane Doe")

message("MYNAME: ${MYNAME}")

if("${MYNAME}" STREQUAL "")
  message("MYNAME is empty!")
endif()
```

```
[cmake-sources]> cmake -P minimum-required-bad/script.cmake
MYNAME: Jane Doe
CMake Warning (dev) at minimum-required-bad/script.cmake:6 (if):
  Policy CMP0054 is not set: Only interpret if() arguments as variables or
  keywords when unquoted.  Run "cmake --help-policy CMP0054" for policy
  details.  Use the cmake_policy command to set the policy and suppress this
  warning.

  Quoted variables like "Jane Doe" will no longer be dereferenced when the
  policy is set to NEW.  Since the policy is not set the OLD behavior will be
  used.
This warning is for project developers.  Use -Wno-dev to suppress it.

MYNAME is empty!
```

Minimum required (good)

Same example with `cmake_minimum_required` works correctly and without warning:

```
# script.cmake

cmake_minimum_required(VERSION 3.1)

set("Jane Doe" "")
set(MYNAME "Jane Doe")

message("MYNAME: ${MYNAME}")

if("${MYNAME}" STREQUAL "")
  message("MYNAME is empty!")
endif()
```

```
[cmake-sources]> cmake -P minimum-required-good/script.cmake
MYNAME: Jane Doe
```

cmake -E

Example of using `cmake -E remove_directory` instead of native `rm`/`rmdir` commands:

CMake documentation

- Command-Line Tool Mode

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(dir_to_remove "${CMAKE_CURRENT_BINARY_DIR}/__temp")

if(WIN32)
  # 'rmdir' will exit with error if directory doesn't exist
  # so we have to put 'if' here
  if(EXISTS "${dir_to_remove}")
    # need to convert to windows-style path
```

```
    file(TO_NATIVE_PATH "${dir_to_remove}" native_path)
    execute_process(
        COMMAND cmd /c rmdir "${native_path}" /S /Q
        RESULT_VARIABLE result
    )
  endif()
else()
  # no need to put 'if', 'rm -rf' produce no error if directory doesn't exist
  execute_process(
      COMMAND rm -rf "${dir_to_remove}"
      RESULT_VARIABLE result
  )
endif()

if(NOT result EQUAL 0)
  # Error
endif()
```

Same code with `cmake -E`:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

execute_process(
    COMMAND "${CMAKE_COMMAND}" -E remove_directory "${CMAKE_CURRENT_BINARY_DIR}/__temp
 ↪"
    RESULT_VARIABLE result
)

if(NOT result EQUAL 0)
  # Error
endif()
```

Note: It's easier to use `file(REMOVE_RECURSE ...)` in this particular example

3.8 Control structures

Examples on GitHub

- Repository
- Latest ZIP

3.8.1 Conditional blocks

Simple examples

Example of using an `if` command with `NO`/`YES` constants and variables with `NO`/`YES` values:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

if(YES)
  message("Condition 1")
endif()

if(NO)
  message("Condition 2")
endif()

set(A "YES")
set(B "NO")

if(A)
  message("Condition 3")
endif()

if(B)
  message("Condition 4")
endif()
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hif-simple -B_builds
Condition 1
Condition 3
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

Adding `else`/`elseif`:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(A "TRUE")
set(B "FALSE")

if(A)
  message("Condition 1")
else()
  message("Condition 2")
endif()

if(B)
  message("Condition 3")
else()
  message("Condition 4")
endif()

set(C "OFF")
set(D "ON")

if(C)
  message("Condition 5")
elseif(D)
  message("Condition 6")
else()
```

```
  message("Condition 7")
endif()

set(E "0")
set(F "0")

if(E)
  message("Condition 8")
elseif(F)
  message("Condition 9")
else()
  message("Condition 10")
endif()
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hif-else -B_builds
Condition 1
Condition 4
Condition 6
Condition 10
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

CMP0054

Some of the `if` commands accept `<variable|string>` arguments. This may lead to quite surprising behavior.

For example if we have a variable `A` and it is set to an empty string we can check it with:

```
set(A "")
if(A STREQUAL "")
  message("Value of A is empty string")
endif()
```

You can save the name of the variable in another variable and do the same:

```
set(A "")
set(B "A") # save name of the variable
if(${B} STREQUAL "")
  message("Value of ${B} is an empty string")
endif()
```

If a CMake policy `CMP0054` is set to `OLD` or not present at all (before CMake 3.1), this operation ignores quotes:

```
set(A "")
set(B "A") # save name of the variable
if("${B}" STREQUAL "") # same as 'if(${B} STREQUAL "")'
  message("Value of ${B} is an empty string")
endif()
```

It means an operation depends on the context: is a variable with the name `${B}` present in current scope or not?

```
cmake_minimum_required(VERSION 3.0)
project(foo LANGUAGES NONE)
```

```
set("Jane Doe" "")
set(A "Jane Doe")

message("A = ${A}")

if("${A}" STREQUAL "")
  message("A is empty")
endif()
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hcmp0054-confuse -B_builds
A = Jane Doe
A is empty
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

Try fix

Since CMake accepts any names of the variables you can't filter out `<variable>` from `<variable|string>` by adding "reserved" symbols:

```
cmake_minimum_required(VERSION 3.0)
project(foo LANGUAGES NONE)

set("Jane Doe" "")
set("xJane Doe" "x")
set("!Jane Doe" "!")
set(" Jane Doe" " ")

set(A "Jane Doe")

message("A = ${A}")

if("x${A}" STREQUAL "x")
  message("A is empty (1)")
endif()

if("!${A}" STREQUAL "!")
  message("A is empty (2)")
endif()

if(" ${A}" STREQUAL " ")
  message("A is empty (3)")
endif()
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Htry-fix -B_builds
A = Jane Doe
A is empty (1)
A is empty (2)
A is empty (3)
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

Fix

To avoid such issues you should use CMake 3.1 and a `CMP0054` policy:

```
cmake_minimum_required(VERSION 3.1)
project(foo LANGUAGES NONE)

set("Jane Doe" "")
set("xJane Doe" "x")
set("!Jane Doe" "!")
set(" Jane Doe" " ")

set(A "Jane Doe")

message("A = ${A}")

if("x${A}" STREQUAL "x")
  message("A is empty (1)")
endif()

if("!${A}" STREQUAL "!")
  message("A is empty (2)")
endif()

if(" ${A}" STREQUAL " ")
  message("A is empty (3)")
endif()
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hcmp0054-fix -B_builds
A = Jane Doe
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

Workaround

For CMake before 3.1 as a workaround you can use a `string(COMPARE EQUAL ...)` command:

```
cmake_minimum_required(VERSION 3.0)
project(foo LANGUAGES NONE)

set("Jane Doe" "")
set("xJane Doe" "x")
set("!Jane Doe" "!")
set(" Jane Doe" " ")

set(A "Jane Doe")

message("A = ${A}")

string(COMPARE EQUAL "${A}" "" is_empty)
if(is_empty)
  message("A is empty")
else()
  message("A is not empty")
endif()
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hcmp0054-workaround -B_builds
A = Jane Doe
A is not empty
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

3.8.2 Loops

foreach

CMake documentation

- foreach

Example of a `foreach(<variable> <list>)` command:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

message("Explicit list:")
foreach(item "A" "B" "C")
  message("  ${item}")
endforeach()

message("Dereferenced list:")
set(mylist "foo" "boo" "bar")
foreach(x ${mylist})
  message("  ${x}")
endforeach()

message("Empty list")
foreach(x)
  message("  ${x}")
endforeach()

message("Dereferenced empty list")
set(empty_list)
foreach(x ${empty_list})
  message("  ${x}")
endforeach()

message("List with empty element:")
foreach(i "")
  message("  '${i}'")
endforeach()

message("Separate lists:")
set(mylist a b c)
foreach(x "${mylist}" "x;y;z")
  message("  ${x}")
endforeach()
```

```
message("Combined list:")
set(combined_list "${mylist}" "x;y;z")
foreach(x ${combined_list})
  message("  ${x}")
endforeach()
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hforeach -B_builds
Explicit list:
  A
  B
  C
Dereferenced list:
  foo
  boo
  bar
Empty list
Dereferenced empty list
List with empty element:
  ''
Separate lists:
  a;b;c
  x;y;z
Combined list:
  a
  b
  c
  x
  y
  z
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

As you may notice `foreach(x "${mylist}" "x;y;z")` is not treated as a single list but as a list with two elements: `${mylist}` and `x;y;z`. If you want to merge two lists you should do it explicitly `set(combined_list "${mylist}" "x;y;z")` or use `foreach(x ${mylist} x y z)` form.

foreach with range

Example of usage of a `foreach(... RANGE ...)` command:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

message("Simple range:")
foreach(x RANGE 10)
  message("  ${x}")
endforeach()

message("Range with limits:")
foreach(x RANGE 3 8)
  message("  ${x}")
endforeach()
```

```
message("Range with step:")
foreach(x RANGE 10 14 2)
  message("  ${x}")
endforeach()
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hforeach-range -B_builds
Simple range:
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
  10
Range with limits:
  3
  4
  5
  6
  7
  8
Range with step:
  10
  12
  14
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

while

Example of usage of a `while` command:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

set(a "")
set(condition TRUE)

message("Loop with condition as variable:")
while(condition)
  set(a "${a}x")
  message("  a = ${a}")
  string(COMPARE NOTEQUAL "${a}" "xxxxx" condition)
endwhile()

set(a "")

message("Loop with explicit condition:")
while(NOT a STREQUAL "xxxxx")
  set(a "${a}x")
```

```
  message("  a = ${a}")
endwhile()
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hwhile -B_builds
Loop with condition as variable:
  a = x
  a = xx
  a = xxx
  a = xxxx
  a = xxxxx
Loop with explicit condition:
  a = x
  a = xx
  a = xxx
  a = xxxx
  a = xxxxx
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

break

CMake documentation

- break

Exit from a loop with a `break` command:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

message("Stop 'while' loop:")
set(a "")
while(TRUE)
  set(a "${a}x")
  message("  ${a}")
  string(COMPARE EQUAL "${a}" "xxx" done)
  if(done)
    break()
  endif()
endwhile()

message("Stop 'foreach' loop:")
foreach(x RANGE 10)
  message("  ${x}")
  if(x EQUAL 4)
    break()
  endif()
endforeach()
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hbreak -B_builds
Stop 'while' loop:
```

```
  x
  xx
  xxx
Stop 'foreach' loop:
  0
  1
  2
  3
  4
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

continue

Since CMake 3.2 it's possible to continue the loop:

```
cmake_minimum_required(VERSION 3.2)
project(foo NONE)

message("Loop with 'continue':")
foreach(x RANGE 10)
  if(x EQUAL 2 OR x EQUAL 5)
    message("  skip ${x}")
    continue()
  endif()
  message("  process ${x}")
endforeach()
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hcontinue -B_builds
Loop with 'continue':
  process 0
  process 1
  skip 2
  process 3
  process 4
  skip 5
  process 6
  process 7
  process 8
  process 9
  process 10
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

CMake documentation

- CMake 3.2 release notes

3.8.3 Functions

CMake documentation

- function

Simple

Function without arguments:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

function(foo)
  message("Calling 'foo' function")
endfunction()

foo()
foo()
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hsimple-function -B_builds
Calling 'foo' function
Calling 'foo' function
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

With arguments

Function with arguments and example of `ARGV*`, `ARGC`, `ARGN` usage:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

function(foo x y z)
  message("Calling function 'foo':")
  message("  x = ${x}")
  message("  y = ${y}")
  message("  z = ${z}")
endfunction()

function(boo x y z)
  message("Calling function 'boo':")
  message("  x = ${ARGV0}")
  message("  y = ${ARGV1}")
  message("  z = ${ARGV2}")
  message("  total = ${ARGC}")
endfunction()

function(bar x y z)
  message("Calling function 'bar':")
  message("  All = ${ARGV}")
  message("  Unexpected = ${ARGN}")
endfunction()
```

```
foo("1" "2" "3")
boo("4" "5" "6")
bar("7" "8" "9" "10" "11")
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hfunction-args -B_builds
Calling function 'foo':
  x = 1
  y = 2
  z = 3
Calling function 'boo':
  x = 4
  y = 5
  z = 6
  total = 3
Calling function 'bar':
  All = 7;8;9;10;11
  Unexpected = 10;11
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

CMake style

CMake documentation

- CMakeParseArguments

`cmake_parse_arguments` function can be used for parsing:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

include(CMakeParseArguments) # cmake_parse_arguments

function(foo)
  set(optional FOO BOO)
  set(one X Y Z)
  set(multiple L1 L2)

  # Introduce:
  # * x_FOO
  # * x_BOO
  # * x_X
  # * x_Y
  # * x_Z
  # * x_L1
  # * x_L2
  cmake_parse_arguments(x "${optional}" "${one}" "${multiple}" "${ARGV}")

  string(COMPARE NOTEQUAL "${x_UNPARSED_ARGUMENTS}" "" has_unparsed)
  if(has_unparsed)
    message(FATAL_ERROR "Unparsed arguments: ${x_UNPARSED_ARGUMENTS}")
  endif()
```

```
  message("FOO: ${x_FOO}")
  message("BOO: ${x_BOO}")
  message("X: ${x_X}")
  message("Y: ${x_Y}")
  message("Z: ${x_Z}")

  message("L1:")
  foreach(item ${x_L1})
    message("  ${item}")
  endforeach()

  message("L2:")
  foreach(item ${x_L2})
    message("  ${item}")
  endforeach()
endfunction()

function(boo)
  set(optional "")
  set(one PARAM1 PARAM2)
  set(multiple "")

  # Introduce:
  # * foo_PARAM1
  # * foo_PARAM2
  cmake_parse_arguments(foo "${optional}" "${one}" "${multiple}" "${ARGV}")

  string(COMPARE NOTEQUAL "${foo_UNPARSED_ARGUMENTS}" "" has_unparsed)
  if(has_unparsed)
    message(FATAL_ERROR "Unparsed arguments: ${foo_UNPARSED_ARGUMENTS}")
  endif()

  message("{ param1, param2 } = { ${foo_PARAM1}, ${foo_PARAM2} }")
endfunction()

message("*** Run (1) ***")
foo(L1 item1 item2 item3 X value FOO)

message("*** Run (2) ***")
foo(L2 item1 item3 Y abc Z 123 FOO BOO)

message("*** Run (3) ***")
foo(L1 item1 L1 item2 L1 item3)

message("*** Run (4) ***")
boo(PARAM1 123 PARAM2 888)
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hcmake-style -B_builds
*** Run (1) ***
FOO: TRUE
BOO: FALSE
X: value
Y:
Z:
L1:
  item1
  item2
```

```
  item3
L2:
*** Run (2) ***
FOO: TRUE
BOO: TRUE
X:
Y: abc
Z: 123
L1:
L2:
  item1
  item3
*** Run (3) ***
FOO: FALSE
BOO: FALSE
X:
Y:
Z:
L1:
  item1
  item2
  item3
L2:
*** Run (4) ***
{ param1, param2 } = { 123, 888 }
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

CMake style limitations

Since it's not possible to create a *list with one empty element* and because of internal `CMakeParseArguments` limitations next calls will have equivalent results:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

include(CMakeParseArguments) # cmake_parse_arguments

function(foo)
  set(optional "")
  set(one X)
  set(multiple "")

  # Introduce:
  # * x_X
  cmake_parse_arguments(x "${optional}" "${one}" "${multiple}" "${ARGV}")

  string(COMPARE NOTEQUAL "${x_UNPARSED_ARGUMENTS}" "" has_unparsed)
  if(has_unparsed)
    message(FATAL_ERROR "Unparsed arguments: ${x_UNPARSED_ARGUMENTS}")
  endif()

  if(DEFINED x_X)
    set(is_defined YES)
  else()
```

```
    set(is_defined NO)
  endif()

  message("X is defined: ${is_defined}")
  message("X value: '${x_X}'")
endfunction()

message("*** Run (1) ***")
foo(X "")

message("*** Run (2) ***")
foo(X)

message("*** Run (3) ***")
foo()
```

```
[examples]> rm -rf _builds
[examples]> cmake -Hcontrol-structures/cmake-style-limitations -B_builds
*** Run (1) ***
X is defined: NO
X value: ''
*** Run (2) ***
X is defined: NO
X value: ''
*** Run (3) ***
X is defined: NO
X value: ''
-- Configuring done
-- Generating done
-- Build files have been written to: /.../examples/_builds
```

Return value

There is no special command to return a value from a function. You can set a variable to the *parent scope* instead:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

function(boo)
  set(A "123" PARENT_SCOPE)
endfunction()

set(A "333")
message("Before 'boo': ${A}")
boo()
message("After 'boo': ${A}")

function(bar arg1 result)
  set("${result}" "ABC-${arg1}-XYZ" PARENT_SCOPE)
endfunction()

message("Calling 'bar' with arguments: '123' 'var_out'")
bar("123" var_out)
message("Output: ${var_out}")
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hreturn-value -B_builds
Before 'boo': 333
After 'boo': 123
Calling 'bar' with arguments: '123' 'var_out'
Output: ABC-123-XYZ
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

Return

CMake documentation

- return

You can exit from a function using a `return` command:

```
cmake_minimum_required(VERSION 2.8)
project(foo NONE)

function(foo A B)
  if(A)
    message("Exit on A")
    return()
  endif()

  if(B)
    message("Exit on B")
    return()
  endif()

  message("Exit")
endfunction()

foo(YES NO)
foo(NO YES)
foo(NO NO)
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hreturn -B_builds
Exit on A
Exit on B
Exit
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

CMAKE_CURRENT_LIST_DIR

Value of `CMAKE_CURRENT_LIST_FILE` and `CMAKE_CURRENT_LIST_DIR` is set to the file/directory from where the function **is called**, not the file where the function **is defined**:

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo NONE)

list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_LIST_DIR}/cmake/Modules")

include(foo_run)

foo_run("123")

add_subdirectory(boo)
```

```
# boo/CMakeLists.txt

foo_run("abc")
```

```
# Module cmake/Modules/foo_run.cmake

set(FOO_RUN_FILE_PATH "${CMAKE_CURRENT_LIST_FILE}")
set(FOO_RUN_DIR_PATH "${CMAKE_CURRENT_LIST_DIR}")

function(foo_run value)
  message("foo_run(${value})")

  message("Called from: ${CMAKE_CURRENT_LIST_DIR}")
  message("Defined in file: ${FOO_RUN_FILE_PATH}")
  message("Defined in directory: ${FOO_RUN_DIR_PATH}")
endfunction()
```

```
[control-structures]> rm -rf _builds
[control-structures]> cmake -Hfunction-location -B_builds
foo_run(123)
Called from: /.../control-structures/function-location
Defined in file: /.../control-structures/function-location/cmake/Modules/foo_run.cmake
Defined in directory: /.../control-structures/function-location/cmake/Modules
foo_run(abc)
Called from: /.../control-structures/function-location/boo
Defined in file: /.../control-structures/function-location/cmake/Modules/foo_run.cmake
Defined in directory: /.../control-structures/function-location/cmake/Modules
-- Configuring done
-- Generating done
-- Build files have been written to: /.../control-structures/_builds
```

CMake documentation

- CMAKE_CURRENT_LIST_DIR
- CMAKE_CURRENT_LIST_FILE

Recommendation

To avoid function name clashing with functions from another modules do prefix name with the project name. In case if function name will match name of the module you can verify that module used in your code just by simple in-file

search (and of course delete it if not):

```
include(foo_my_module_1)
include(foo_my_module_2)

foo_my_module_1(INPUT1 "abc" INPUT2 123 RESULT result)
foo_my_module_2(INPUT1 "${result}" INPUT2 "xyz")
```

See also:

- *Module names*
- *Cache names*

3.9 Executables

Examples on GitHub

- Repository
- Latest ZIP

CMake documentation

- add_executable

3.9.1 Simple

Building executable from `main.cpp`:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

add_executable(foo main.cpp)
```

```
[executable-examples]> rm -rf _builds
[executable-examples]> cmake -Hsimple -B_builds
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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: /.../executable-examples/_builds
[executable-examples]> cmake --build _builds
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/main.cpp.o
[100%] Linking CXX executable foo
[100%] Built target foo
```

```
[executable-examples]> ./_builds/foo
Hello from CGold!
```

3.9.2 Duplicates

Targets are global, you can't declare two targets with the same name even if they are declared in different `CMakeLists.txt`:

```
# top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_subdirectory(boo)
add_subdirectory(bar)
```

```
# boo/CMakeLists.txt

add_executable(foo main.cpp)
```

```
# bar/CMakeLists.txt

add_executable(foo main.cpp)
```

```
[examples]> rm -rf _builds
[examples]> cmake -Hexecutable-examples/duplicates -B_builds
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
CMake Error at bar/CMakeLists.txt:1 (add_executable):
  add_executable cannot create target "foo" because another target with the
  same name already exists.  The existing target is an executable created in
  source directory
  "/.../executable-examples/duplicates/boo".
  See documentation for policy CMP0002 for more details.
```

3.10 Tests

In previous section we have checked that executable is working by finding it in binary tree and running it explicitly. If we have several executables or want to run the same executable with different parameters we can organize everything into test suite driven by CTest tool.

CMake documentation

- ctest
- add_test
- enable_testing

Examples on GitHub

- Repository
- Latest ZIP

Creating two executables:

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_executable(boo boo.cpp)
add_executable(bar bar.cpp)

enable_testing()
add_test(NAME boo COMMAND boo)

add_test(NAME bar COMMAND bar)
add_test(NAME bar-with-args COMMAND bar arg1 arg2 arg3)
```

Executable `boo`:

```
#include <iostream> // std::cout

int main() {
  std::cout << "boo" << std::endl;
}
```

Executable `bar`:

```
#include <iostream> // std::cout

int main(int argc, char** argv) {
  std::cout << "bar argc: " << argc << std::endl;
  for (int i=1; i<argc; ++i) {
    std::cout << "argv[" << i << "]: " << argv[i] << std::endl;
  }
}
```

Testing allowed by `enable_testing` directive which must be called in **the root directory**:

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_executable(boo boo.cpp)
add_executable(bar bar.cpp)

enable_testing()
add_test(NAME boo COMMAND boo)

add_test(NAME bar COMMAND bar)
add_test(NAME bar-with-args COMMAND bar arg1 arg2 arg3)
```

Come up with some tests name and specify executable arguments if needed:

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_executable(boo boo.cpp)
add_executable(bar bar.cpp)

enable_testing()
add_test(NAME boo COMMAND boo)

add_test(NAME bar COMMAND bar)
add_test(NAME bar-with-args COMMAND bar arg1 arg2 arg3)
```

Configure and build project:

```
[examples]> rm -rf _builds
[examples]> cmake -Htest-examples/simple -B_builds
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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: /.../examples/_builds
[examples]> cmake --build _builds
Scanning dependencies of target boo
[ 25%] Building CXX object CMakeFiles/boo.dir/boo.cpp.o
[ 50%] Linking CXX executable boo
[ 50%] Built target boo
Scanning dependencies of target bar
[ 75%] Building CXX object CMakeFiles/bar.dir/bar.cpp.o
```

```
[100%] Linking CXX executable bar
[100%] Built target bar
```

Enter `_builds` directory and use `ctest` tool to run all tests:

```
[examples]> cd _builds
[examples/_builds]> ctest
Test project /.../examples/_builds
    Start 1: boo
1/3 Test #1: boo ..............................   Passed    0.00 sec
    Start 2: bar
2/3 Test #2: bar ..............................   Passed    0.00 sec
    Start 3: bar-with-args
3/3 Test #3: bar-with-args ....................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 3

Total Test time (real) =   0.02 sec
```

3.10.1 Multi-config testing

Note that for the *multi-configuration generators* you have to specify build type while running `ctest`. Otherwise no tests will be run. Example of Visual Studio project:

```
[examples\_builds]> ctest
Test project C:/.../examples/_builds
    Start 1: boo
Test not available without configuration.  (Missing "-C <config>"?)
1/3 Test #1: boo ..............................***Not Run   0.00 sec
    Start 2: bar
Test not available without configuration.  (Missing "-C <config>"?)
2/3 Test #2: bar ..............................***Not Run   0.00 sec
    Start 3: bar-with-args
Test not available without configuration.  (Missing "-C <config>"?)
3/3 Test #3: bar-with-args ....................***Not Run   0.00 sec

0% tests passed, 3 tests failed out of 3

Total Test time (real) =   0.02 sec

The following tests FAILED:
          1 - boo (Not Run)
          2 - bar (Not Run)
          3 - bar-with-args (Not Run)
Errors while running CTest
```

Just add `-C Debug` to test with `Debug` build type:

```
[examples\_builds]> ctest -C Debug
Test project C:/.../examples/_builds
    Start 1: boo
1/3 Test #1: boo ..............................   Passed    0.04 sec
    Start 2: bar
2/3 Test #2: bar ..............................   Passed    0.02 sec
    Start 3: bar-with-args
3/3 Test #3: bar-with-args ....................   Passed    0.01 sec
```

```
100% tests passed, 0 tests failed out of 3

Total Test time (real) =   0.09 sec
```

3.10.2 Verbose output

By default only `Passed`/`Failed` information is shown. You can control tests output by `-V`/`-VV` options:

```
[examples/_builds]> ctest -VV
...
test 1
    Start 1: boo

1: Test command: /.../examples/_builds/boo
1: Test timeout computed to be: 9.99988e+06
1: boo
1/3 Test #1: boo ..............................   Passed    0.00 sec
test 2
    Start 2: bar

2: Test command: /.../examples/_builds/bar
2: Test timeout computed to be: 9.99988e+06
2: bar argc: 1
2/3 Test #2: bar ..............................   Passed    0.00 sec
test 3
    Start 3: bar-with-args

3: Test command: /.../examples/_builds/bar "arg1" "arg2" "arg3"
3: Test timeout computed to be: 9.99988e+06
3: bar argc: 4
3: argv[1]: arg1
3: argv[2]: arg2
3: argv[3]: arg3
3/3 Test #3: bar-with-args ....................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 3

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

3.10.3 Subset of tests

It is possible to run only subset of tests instead of all suite. For example running all tests with `bar` pattern in name by using regular expression:

```
[examples/_builds]> ctest -R bar
Test project /.../examples/_builds
    Start 2: bar
1/2 Test #2: bar ..............................   Passed    0.00 sec
    Start 3: bar-with-args
2/2 Test #3: bar-with-args ....................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 2

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

Or only `bar` test:

```
[examples/_builds]> ctest -R '^bar$'
Test project /.../examples/_builds
    Start 2: bar
1/1 Test #2: bar ..............................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 1

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

3.11 Libraries

3.11.1 Static

3.11.2 Shared

3.11.3 Static + shared

Those users who has worked with autotools knows that it's possible to build both static and shared libraries at one go. Here is an overview how it should be done in CMake.

Examples on GitHub

- Repository
- Latest ZIP

Right way

We will start with the right one. Command add_library should be used without `STATIC` or `SHARED` specifier, type of the library will be determined by value of BUILD_SHARED_LIBS variable (default type is static):

```
cmake_minimum_required(VERSION 3.4)
project(foo)

set(CMAKE_WINDOWS_EXPORT_ALL_SYMBOLS YES CACHE BOOL "Export all symbols")

add_library(foo foo.cpp)

install(
    TARGETS foo
    LIBRARY DESTINATION lib
    ARCHIVE DESTINATION lib
    RUNTIME DESTINATION bin
)
```

Note: `STATIC`/`SHARED`/`MODULE` specifiers should be used only in cases when other type of library is by design not possible for any reasons. That's not our case of course since we are trying to build both variants, hence library

designed to be used as static or shared.

Libraries should be installed to separate directories. So there will be **two builds** and **two root directories**. *Out of source* will kindly help us:

```
> cd library-examples
[library-examples]> rm -rf _builds _install
[library-examples]> cmake -Hright-way -B_builds/shared -DBUILD_SHARED_LIBS=ON -DCMAKE_
↪INSTALL_PREFIX="`pwd`/_install/configuration-A"
[library-examples]> cmake --build _builds/shared --target install
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX shared library libfoo.so
[100%] Built target foo
Install the project...
-- Install configuration: ""
-- Installing: /.../library-examples/_install/configuration-A/lib/libfoo.so

[library-examples]> cmake -Hright-way -B_builds/static -DCMAKE_INSTALL_PREFIX="`pwd`/_
↪install/configuration-B"
[library-examples]> cmake --build _builds/static --target install
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX static library libfoo.a
[100%] Built target foo
Install the project...
-- Install configuration: ""
-- Installing: /.../library-examples/_install/configuration-B/lib/libfoo.a
```

Autotools two builds

Note that autotools do build library twice too under the hood, so performance is the same:

```
> mkdir temp
> cd temp
[temp]> wget http://www.x.org/releases/individual/lib/libpciaccess-0.13.4.tar.bz2
[temp]> tar xf libpciaccess-0.13.4.tar.bz2
[temp]> cd libpciaccess-0.13.4
[libpciaccess-0.13.4]> ./configure --enable-shared --enable-static
[libpciaccess-0.13.4]> make V=1
...
libtool: compile:  gcc ... -c linux_devmem.c -fPIC -o .libs/linux_devmem.o
libtool: compile:  gcc ... -c linux_devmem.c -o linux_devmem.o
```

Install to one directory

Another autotools feature is that both libraries will be installed to the one directory. That's works fine on Linux since libraries names will be `libfoo.so` and `libfoo.a`, works fine for OSX since libraries names will be `libfoo.dylib` and `libfoo.a`, but not for Windows. Static build will produce `foo.lib`:

```
> cd library-examples
[library-examples]> rmdir _builds _install /S /Q
[library-examples]> cmake -Hright-way -B_builds\static -G "Visual Studio 14 2015" -
↪DCMAKE_INSTALL_PREFIX=%cd%\_install
```

```
[library-examples]> cmake --build _builds\static --config Release --target install
...
-- Install configuration: "Release"
-- Installing: C:/.../library-examples/_install/lib/foo.lib
```

But shared build will produce **both** `foo.lib` and `foo.dll`, effectively **overwriting** static library and making it **unusable**:

```
[library-examples]> cmake -Hright-way -B_builds\shared -G "Visual Studio 14 2015" -
↪DBUILD_SHARED_LIBS=ON -DCMAKE_INSTALL_PREFIX=%cd%\_install
[library-examples]> cmake --build _builds\shared --config Release --target install
...
-- Install configuration: "Release"
-- Installing: C:/.../library-examples/_install/lib/foo.lib
-- Installing: C:/.../library-examples/_install/bin/foo.dll
```

Configs

Even if libraries doesn't conflict on file level their **configs** will conflict:

```
> cd library-examples
[library-examples]> rm -rf _install _builds
[library-examples]> cmake -Hbar -B_builds/shared -DBUILD_SHARED_LIBS=ON -DCMAKE_BUILD_
↪TYPE=Release -DCMAKE_INSTALL_PREFIX="`pwd`/_install"
[library-examples]> cmake --build _builds/shared --target install
[library-examples]> grep lib/libbar.so -IR _install
_install/lib/cmake/bar/barTargets-release.cmake:  IMPORTED_LOCATION_RELEASE "${_
↪IMPORT_PREFIX}/lib/libbar.so"
_install/lib/cmake/bar/barTargets-release.cmake:list(APPEND _IMPORT_CHECK_FILES_FOR_
↪bar::bar "${_IMPORT_PREFIX}/lib/libbar.so" )
```

Config for static variant will have the same `barTargets-release.cmake` name:

```
[library-examples]> cmake -Hbar -B_builds/static -DCMAKE_BUILD_TYPE=Release -DCMAKE_
↪INSTALL_PREFIX="`pwd`/_install"
[library-examples]> cmake --build _builds/static --target install
[library-examples]> grep lib/libbar.a -IR _install
_install/lib/cmake/bar/barTargets-release.cmake:  IMPORTED_LOCATION_RELEASE "${_
↪IMPORT_PREFIX}/lib/libbar.a"
_install/lib/cmake/bar/barTargets-release.cmake:list(APPEND _IMPORT_CHECK_FILES_FOR_
↪bar::bar "${_IMPORT_PREFIX}/lib/libbar.a" )
```

Now since configuration files for shared variant are overwritten there is no way to load `libbar.so` using `find_package(bar CONFIG REQUIRED)`.

```
[library-examples]> grep lib/libbar.so -IR _install
[library-examples]> echo $?
1
```

Two targets

Problems with two versions of library described in previous section can be solved by using two different targets. This section cover building of two targets simultaneously. One target build at the time is equivalent to this code:

```
add_library(foo foo.cpp)
```

Even if names differs, e.g. by using `option`:

```
option(FOO_STATIC_LIB "Build static library" ON)

if(FOO_STATIC_LIB)
  add_library(foo_static STATIC foo.cpp)
else()
  add_library(foo_shared SHARED foo.cpp)
endif()
```

Warning: This is logically equivalent to the `add_library(foo foo.cpp)` + `BUILD_SHARED_LIBS` functionality so **should not be used**. Use standard CMake features!

So assuming we have code like this:

```
# Don't do that!
add_library(foo_static STATIC foo.cpp)
add_library(foo_shared SHARED foo.cpp)
```

Philosophical

CMake code describe **abstract** configuration. User can choose how this abstraction used on practice. Let's run this example on OSX:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

add_library(foo foo.cpp)
add_executable(boo boo.cpp)

target_link_libraries(boo PUBLIC foo)
```

By default we will build executable and static library:

```
> cd library-examples
[library-examples]> rm -rf _builds
[library-examples]> cmake -Hcustom -B_builds
[library-examples]> cmake --build _builds
[library-examples]> ls _builds/libfoo.a _builds/boo
_builds/libfoo.a
_builds/boo
```

But we are free to switch to shared library:

```
[library-examples]> rm -rf _builds
[library-examples]> cmake -Hcustom -B_builds -DBUILD_SHARED_LIBS=ON
[library-examples]> cmake --build _builds
[library-examples]> ls _builds/libfoo.dylib _builds/boo
_builds/libfoo.dylib
_builds/boo
```

Create bundle:

```
[library-examples]> rm -rf _builds
[library-examples]> cmake -Hcustom -B_builds -DCMAKE_MACOSX_BUNDLE=ON
[library-examples]> cmake --build _builds
[library-examples]> ls -d _builds/libfoo.a _builds/boo.app
_builds/libfoo.a
_builds/boo.app
```

Or do the both:

```
[library-examples]> rm -rf _builds
[library-examples]> cmake -Hcustom -B_builds -DCMAKE_MACOSX_BUNDLE=ON -DBUILD_SHARED_
↪LIBS=ON
[library-examples]> cmake --build _builds
[library-examples]> ls -d _builds/libfoo.dylib _builds/boo.app
_builds/libfoo.dylib
_builds/boo.app
```

Forcing any of this violates customization principle.

Non-default behavior

Let's see how two targets approach will be used on user's side:

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_subdirectory(boo) # 3rd party library

add_executable(foo foo.cpp)
target_link_libraries(foo PUBLIC boo)
```

Targets defined in directory `boo`:

```
# boo/CMakeLists.txt

# Don't do that!
add_library(boo STATIC boo.cpp)
add_library(boo_shared SHARED boo.cpp)
```

User builds library and link by default static `libboo.a` to `foo` executable:

```
> cd library-examples
[library-examples]> rm -rf _builds
[library-examples]> cmake -Hsurprise -B_builds -DCMAKE_VERBOSE_MAKEFILE=ON
[library-examples]> cmake --build _builds
...
/usr/bin/c++ -o foo ... boo/libboo.a
```

User knows that there is `BUILD_SHARED_LIBS` variable that change type of library, so he expects shared in next configuration:

```
[library-examples]> rm -rf _builds
[library-examples]> cmake -Hsurprise -B_builds -DCMAKE_VERBOSE_MAKEFILE=ON -DBUILD_
↪SHARED_LIBS=ON
```

But of course he still got static because type of library is forced:

```
[library-examples]> cmake --build _builds
/usr/bin/c++ -o foo ... boo/libboo.a
```

Build time

Note that in previous example time of compilation of `boo` library is **doubled**. We are building `boo.cpp` **twice** even if we are not planning to use one of the variants:

```
[library-examples]> rm -rf _builds
[library-examples]> cmake -Hsurprise -B_builds
[library-examples]> cmake --build _builds
Scanning dependencies of target boo
[ 16%] Building CXX object boo/CMakeFiles/boo.dir/boo.cpp.o
[ 33%] Linking CXX static library libboo.a
[ 33%] Built target boo
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[ 66%] Linking CXX executable foo
[ 66%] Built target foo
Scanning dependencies of target boo_shared
[ 83%] Building CXX object boo/CMakeFiles/boo_shared.dir/boo.cpp.o
[100%] Linking CXX shared library libboo_shared.so
[100%] Built target boo_shared
```

User of such library pays for something **he doesn't really need**.

PIC conflicts

Assume we want to build everything statically but some part of out code force library to be shared:

```
cmake_minimum_required(VERSION 2.8)
project(use_bar)

find_package(bar CONFIG REQUIRED)

add_library(use_bar_static STATIC use_bar.cpp)
target_link_libraries(use_bar_static PUBLIC bar::bar)

add_library(use_bar_shared SHARED use_bar.cpp)
target_link_libraries(use_bar_shared PUBLIC bar::bar)
```

If `bar` is static we will have problem with target `use_bar_shared` which in fact **we don't really interested in**:

```
> cd library-examples
[library-examples]> rm -rf _builds _install
[library-examples]> cmake -Hbar -B_builds -DCMAKE_INSTALL_PREFIX="`pwd`/_install"
[library-examples]> cmake --build _builds --target install

[library-examples]> rm -rf _builds
[library-examples]> cmake -Huse_bar -B_builds -DCMAKE_PREFIX_PATH="`pwd`/_install"
[library-examples]> cmake --build _builds
Scanning dependencies of target use_bar_shared
[ 25%] Building CXX object CMakeFiles/use_bar_shared.dir/use_bar.cpp.o
```

```
[ 50%] Linking CXX shared library libuse_bar_shared.so
/usr/bin/ld: /.../library-examples/_install/lib/libbar.a(bar.cpp.o):
    relocation R_X86_64_PC32 against symbol `_Z4bar1v' can not be used when
    making a shared object; recompile with -fPIC
```

Note: Such issue **can't be solved** by library usage requirements since library `bar` don't know a priory will it be linked to shared library or not.

Scalability

Two targets approach doesn't scale. If we have `add_library(foo foo.cpp)` we can do control of such code:

```
add_library(foo foo.cpp)
add_executable(boo boo.cpp)
target_link_libraries(boo PUBLIC foo)
```

Using `BUILD_SHARED_LIBS`:

- `ON` - executable linked with shared library
- `OFF` - executable linked with static library

In this code:

```
add_library(foo_static STATIC foo.cpp)
add_library(foo_shared SHARED foo.cpp)
```

What should we do? Create two targets?

```
add_executable(boo_static boo.cpp)
target_link_libraries(boo_static PUBLIC foo_static)

add_executable(boo_shared boo.cpp)
target_link_libraries(boo_shared PUBLIC foo_shared)
```

What if there will be more dependencies?

```
add_library(foo_static STATIC foo.cpp)
add_library(foo_shared SHARED foo.cpp)

add_library(bar_static STATIC foo.cpp)
add_library(bar_shared SHARED foo.cpp)

# 1 - shared, 0 - static
add_executable(boo_0_0 boo.cpp)
add_executable(boo_0_1 boo.cpp)
add_executable(boo_1_0 boo.cpp)
add_executable(boo_1_1 boo.cpp)

target_link_libraries(boo_0_0 PUBLIC foo_static boo_static)
target_link_libraries(boo_0_1 PUBLIC foo_static boo_shared)
target_link_libraries(boo_1_0 PUBLIC foo_shared boo_static)
target_link_libraries(boo_1_1 PUBLIC foo_shared boo_shared)
```

Duplication

Additionally to scalability problems in previous example we have a risk to have same code repeated twice for system with complex dependencies. Assume we have library `bar` in two variants simultaneously:

```
# bar/CMakeLists.txt

# Don't do that!
add_library(bar_static STATIC bar.cpp)
add_library(bar_shared SHARED bar.cpp)
```

And target `baz` that for some reason decide that shared variant of linkage is preferable:

```
# baz/CMakeLists.txt

add_library(baz SHARED baz.cpp)
target_link_libraries(baz PUBLIC bar_shared)
```

Our executable links to both libraries. Probably we don't know/not interested in fact that `baz` use `bar` too. We decide that static linkage is preferable for any reason:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

add_subdirectory(bar)
add_subdirectory(baz)

add_executable(foo foo.cpp)
target_link_libraries(foo PUBLIC bar_static baz)
```

Let's build it:

```
[library-examples]> rm -rf _builds
[library-examples]> cmake -Hdup -B_builds
[library-examples]> cmake --build _builds
```

We are linked to the `libbaz.so` and we **do linked** to `libbar_shared.so` because it's dependency of `baz`:

```
> ldd _builds/foo
  ...
  libbaz.so => /.../library-examples/_builds/baz/libbaz.so (0x00007f6d2f2a4000)
  libbar_shared.so => /.../library-examples/_builds/bar/libbar_shared.so␣
↪(0x00007f6d2e927000)
```

At the same time we have `bar` linked statically:

```
> objdump -d _builds/foo | grep -A5 'barv.*:'
0000000000400c12 <_Z3barv>:
  400c12:       55                      push   %rbp
  400c13:       48 89 e5                mov    %rsp,%rbp
  400c16:       b8 42 00 00 00          mov    $0x42,%eax
  400c1b:       5d                      pop    %rbp
  400c1c:       c3                      retq
```

So effectively code of function `bar` present in our dependencies twice! First time in executable and second time in linked shared library:

```
> objdump -d _builds/bar/libbar_shared.so | grep -A5 'barv.*:'
0000000000000610 <_Z3barv>:
 610:   55                      push   %rbp
 611:   48 89 e5                mov    %rsp,%rbp
 614:   b8 42 00 00 00          mov    $0x42,%eax
 619:   5d                      pop    %rbp
 61a:   c3                      retq
```

Summary

- Use `STATIC`/`SHARED`/`MODULE` only if **library designed** to have no other types
- Use **no specifiers** if library designed to be used as static or shared. Respect `BUILD_SHARED_LIBS` variable
- Install static and shared libraries to **separate directories**

CMake mailing list

- Static & shared library

3.11.4 Symbols

In case of diagnosing linker errors or hiding some functions from public usage it may be helpful to know the table of symbols of library.

Tools

The tool for listing symbols differs for different platforms.

Examples on GitHub

- Repository
- Latest ZIP

Example

Here is an example of library which has both defined and undefined symbols:

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_library(boo Boo.hpp Boo.cpp Foo.hpp)
```

Method `Boo::boo` declared and will be defined:

```
// Boo.hpp

#ifndef BOO_HPP_
#define BOO_HPP_

class Boo {
 public:
  int boo(int, char);
};

#endif // BOO_HPP_
```

```
// Boo.cpp

#include "Boo.hpp"

#include "Foo.hpp"

int Boo::boo(int x, char a) {
  Foo foo;

  return foo.foo(a, 1.0 + x);
}
```

Method `Foo::foo` declared, will be used but **will not be** defined:

```
// Foo.hpp

#ifndef FOO_HPP_
#define FOO_HPP_

class Foo {
 public:
  int foo(char, double);
};

#endif // FOO_HPP_
```

```
// Boo.cpp

#include "Boo.hpp"

#include "Foo.hpp"

int Boo::boo(int x, char a) {
  Foo foo;

  return foo.foo(a, 1.0 + x);
}
```

Build library:

```
[library-examples]> rm -rf _builds
[library-examples]> cmake -Hlibrary-symbols -B_builds
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
```

```
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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: /.../library-examples/_builds

[library-examples]> cmake --build _builds
Scanning dependencies of target boo
[ 50%] Building CXX object CMakeFiles/boo.dir/Boo.cpp.o
[100%] Linking CXX static library libboo.a
[100%] Built target boo

[library-examples]> ls _builds/libboo.a
_builds/libboo.a
```

Linux

Use `nm` for Linux:

```
> which nm
/usr/bin/nm
```

Install instructions for Ubuntu:

```
> sudo apt-get install binutils
```

`nm --defined-only` will show symbols defined by current module. Add `--demangle` to beautify output:

```
[library-examples]> nm --defined-only --demangle _builds/libboo.a

Boo.cpp.o:
0000000000000000 T Boo::boo(int, char)
```

`nm --undefined-only` will show undefined:

```
[library-examples]> nm --undefined-only --demangle _builds/libboo.a

Boo.cpp.o:
                 U __stack_chk_fail
                 U Foo::foo(char, double)
```

OSX

Same `nm` tool with `--defined-only`/`--undefined-only` options can be used on `OSX` platform. However `--demangle` is not available, `c++filt` can be used instead:

```
> which nm
/usr/bin/nm

> which c++filt
/usr/bin/c++filt
```

Defined symbols:

```
> nm --defined-only _builds/libboo.a | c++filt

_builds/libboo.a(Boo.cpp.o):
0000000000000000 T Boo::boo(int, char)
```

Undefined symbols:

```
> nm --undefined-only _builds/libboo.a | c++filt

_builds/libboo.a(Boo.cpp.o):
Foo::foo(char, double)
```

Windows

`DUMPBIN` tool can help to discover symbols on Windows platform. It's available via *Developer Command Prompt*:

```
> where dumpbin
...\msvc\2015\VC\bin\dumpbin.exe
```

Add `/SYMBOLS` to see the table. Defined symbols can be filtered by `External` + `SECT`:

```
[library-examples]> dumpbin /symbols _builds\Debug\boo.lib | findstr "External" |␣
↪findstr "SECT"
00A 00000000 SECT4  notype ()    External     | ?boo@Boo@@QAEHHD@Z (public: int __
↪thiscall Boo::boo(int,char))
01C 00000000 SECT7  notype       External     | __real@3ff0000000000000
```

Undefined by `External` + `UNDEF`:

```
[library-examples]> dumpbin /symbols _builds\Debug\boo.lib | findstr "External" |␣
↪findstr "UNDEF"
00B 00000000 UNDEF  notype ()    External     | ?foo@Foo@@QAEHDN@Z (public: int __
↪thiscall Foo::foo(char,double))
00C 00000000 UNDEF  notype ()    External     | @_RTC_CheckStackVars@8
00D 00000000 UNDEF  notype ()    External     | __RTC_CheckEsp
00E 00000000 UNDEF  notype ()    External     | __RTC_InitBase
00F 00000000 UNDEF  notype ()    External     | __RTC_Shutdown
019 00000000 UNDEF  notype       External     | __fltused
```

See also:

- DUMPBIN reference
- DUMPBIN /SYMBOLS

Use `/EXPORTS` if you want to see the symbols available in DLL:

```
[library-examples]> dumpbin /exports _builds\Release\boo.dll | findstr "Boo"
  1    0 00001000 ?boo@Boo@@QAEHHD@Z
```

Use `undname` to demangle:

```
[library-examples]> undname ?boo@Boo@@QAEHHD@Z
Microsoft (R) C++ Name Undecorator
Copyright (C) Microsoft Corporation. All rights reserved.

Undecoration of :- "?boo@Boo@@QAEHHD@Z"
is :- "public: int __thiscall Boo::boo(int,char)"
```

See also:

- DUMPBIN /EXPORTS
- Viewing Decorated Names

Simple error

Examples on GitHub

- Repository
- Latest ZIP

Here is an example of trivial "undefined reference" error with diagnostic and, of course, fix instructions.

Library `boo`:

```
# boo/CMakeLists.txt

add_library(boo Boo.hpp Boo.cpp)
```

```
// boo/Boo.hpp

#ifndef BOO_HPP_
#define BOO_HPP_

class Boo {
 public:
  int boo(int, int);
};

#endif // BOO_HPP_
```

```
// boo/Boo.cpp

#include "boo/Boo.hpp"

int Boo::boo(int, int) {
  return 0x42;
}
```

Library `foo` use library `boo` but since we are trying to trigger an error the `target_link_libraries` directive is intentionally missing:

```
# foo/CMakeLists.txt

add_library(foo Foo.cpp Foo.hpp)
```

```
// foo/Foo.hpp

#ifndef FOO_HPP_
#define FOO_HPP_

class Foo {
 public:
  int foo(int, char);
};

#endif // FOO_HPP_
```

```
// foo/Foo.cpp

#include "foo/Foo.hpp"
#include "boo/Boo.hpp"

int Foo::foo(int, char) {
  Boo boo;
  return boo.boo(14, 15);
}
```

Final `baz` executable:

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(baz)

include_directories(${CMAKE_CURRENT_LIST_DIR}) # for '#include <boo/Boo.hpp>'

add_subdirectory(boo)
add_subdirectory(foo)

add_executable(baz main.cpp)
target_link_libraries(baz foo)
```

```
#include "foo/Foo.hpp"

int main() {
  Foo foo;
  return foo.foo(144, 'x');
}
```

Generate project:

```
[library-examples]> rm -rf _builds
[library-examples]> cmake -Hlink-error -B_builds
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
```

```
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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: /.../library-examples/_builds
```

First let's build library `boo`:

```
[library-examples]> cmake --build _builds --target boo
-- Configuring done
-- Generating done
-- Build files have been written to: /.../library-examples/_builds
Scanning dependencies of target boo
[ 50%] Building CXX object boo/CMakeFiles/boo.dir/Boo.cpp.o
[100%] Linking CXX static library libboo.a
[100%] Built target boo
```

An attempt to build executable `baz` will fail with link error:

```
> cmake --build _builds --target baz
Scanning dependencies of target foo
[ 25%] Building CXX object foo/CMakeFiles/foo.dir/Foo.cpp.o
[ 50%] Linking CXX static library libfoo.a
[ 50%] Built target foo
Scanning dependencies of target baz
[ 75%] Building CXX object CMakeFiles/baz.dir/main.cpp.o
[100%] Linking CXX executable baz
foo/libfoo.a(Foo.cpp.o): In function `Foo::foo(int, char)':
Foo.cpp:(.text+0x35): undefined reference to `Boo::boo(int, int)'
collect2: error: ld returned 1 exit status
CMakeFiles/baz.dir/build.make:95: recipe for target 'baz' failed
make[3]: *** [baz] Error 1
CMakeFiles/Makefile2:67: recipe for target 'CMakeFiles/baz.dir/all' failed
make[2]: *** [CMakeFiles/baz.dir/all] Error 2
CMakeFiles/Makefile2:79: recipe for target 'CMakeFiles/baz.dir/rule' failed
make[1]: *** [CMakeFiles/baz.dir/rule] Error 2
Makefile:118: recipe for target 'baz' failed
make: *** [baz] Error 2
```

Use `nm` tool to verify that symbol is indeed undefined:

```
> nm --undefined-only --demangle _builds/foo/libfoo.a

Foo.cpp.o:
                 U __stack_chk_fail
                 U Boo::boo(int, int)
```

Library `boo` has it:

```
> nm --defined-only --demangle _builds/boo/libboo.a

Boo.cpp.o:
0000000000000000 T Boo::boo(int, int)
```

So library `foo` depends on library `boo`, every time we are linking `foo` we have to link `boo` too. This can be expressed by `target_link_libraries` command. Fix:

```
--- /examples/library-examples/link-error/foo/CMakeLists.txt
+++ /examples/library-examples/link-error-fix/foo/CMakeLists.txt
@@ -1,3 +1,4 @@
 # foo/CMakeLists.txt

 add_library(foo Foo.cpp Foo.hpp)
+target_link_libraries(foo PUBLIC boo)
```

Should work now:

```
[library-examples]> rm -rf _builds
[library-examples]> cmake -Hlink-error-fix -B_builds
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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: /.../library-examples/_builds

[library-examples]> cmake --build _builds
Scanning dependencies of target boo
[ 16%] Building CXX object boo/CMakeFiles/boo.dir/Boo.cpp.o
[ 33%] Linking CXX static library libboo.a
[ 33%] Built target boo
Scanning dependencies of target foo
[ 50%] Building CXX object foo/CMakeFiles/foo.dir/Foo.cpp.o
[ 66%] Linking CXX static library libfoo.a
[ 66%] Built target foo
Scanning dependencies of target baz
[ 83%] Building CXX object CMakeFiles/baz.dir/main.cpp.o
[100%] Linking CXX executable baz
[100%] Built target baz
```

ODR violation (local)

Examples on GitHub

- Repository
- Latest ZIP

The next example is about scenario when badly written CMake code leads to *ODR* violation.

Assume we have library `boo`:

```
# boo/CMakeLists.txt

add_definitions(-DBOO_USE_SHORT_INT) # This is wrong!
add_library(boo Boo.hpp Boo.cpp)
```

```
// boo/Boo.hpp

#ifndef BOO_HPP_
#define BOO_HPP_

class Boo {
 public:
#ifdef BOO_USE_SHORT_INT
  typedef short int value_type;
#else
  typedef unsigned long long value_type;
#endif

  static void boo(int, value_type);
};

#endif // BOO_HPP_
```

```
// boo/Boo.cpp

#include "boo/Boo.hpp"

void Boo::boo(int, value_type) {
}
```

Methods of `boo` used in library `foo`:

```
// foo/Foo.hpp

#ifndef FOO_HPP_
#define FOO_HPP_

class Foo {
 public:
  static void foo(int, int);
};

#endif // FOO_HPP_
```

```
// foo/Foo.cpp

#include "foo/Foo.hpp"
#include "boo/Boo.hpp"

void Foo::foo(int, int) {
  Boo::value_type x(2);
  return Boo::boo(1, x);
}
```

```
# foo/CMakeLists.txt

add_library(foo Foo.hpp Foo.cpp)
target_link_libraries(foo PUBLIC boo)
```

And final executable `baz`:

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(baz)

include_directories(${CMAKE_CURRENT_LIST_DIR}) # for '#include <boo/Boo.hpp>'

add_subdirectory(boo)
add_subdirectory(foo)

add_executable(baz main.cpp)
target_link_libraries(baz foo)
```

```
#include "foo/Foo.hpp"

int main() {
  Foo::foo(0, 0);
}
```

Let's build the project now:

```
[library-examples]> rm -rf _builds
[library-examples]> cmake -Hlink-error-odr-local -B_builds -DCMAKE_VERBOSE_MAKEFILE=ON
...
[library-examples]> cmake --build _builds
```

Link will fail with "undefined reference" error:

```
/usr/bin/c++ -DBOO_USE_SHORT_INT /.../Boo.cpp
...
/usr/bin/c++ /.../Foo.cpp
...
/usr/bin/c++ -rdynamic CMakeFiles/baz.dir/main.cpp.o -o baz foo/libfoo.a boo/libboo.a
foo/libfoo.a(Foo.cpp.o): In function `Foo::foo(int, int)':
Foo.cpp:(.text+0x23): undefined reference to `Boo::boo(int, unsigned long long)'
collect2: error: ld returned 1 exit status
CMakeFiles/baz.dir/build.make:99: recipe for target 'baz' failed
make[2]: *** [baz] Error 1
```

Check symbols we need:

```
[library-examples]> nm --defined-only --demangle _builds/boo/libboo.a

Boo.cpp.o:
0000000000000000 T Boo::boo(int, short)
```

Indeed that's not what we are looking for:

```
[library-examples]> nm --undefined-only --demangle _builds/foo/libfoo.a

```

```
Foo.cpp.o:
    U Boo::boo(int, unsigned long long)
```

The reason of the failure is that we use `BOO_USE_SHORT_INT` while building `boo` library and not using it while building library `foo`. Since in both cases we are loading `boo/Boo.hpp` header (which depends on `BOO_USE_SHORT_INT`) we should define `BOO_USE_SHORT_INT` in both cases too. target_compile_definitions can help us to solve the issue:

```
--- /examples/library-examples/link-error-odr-local/boo/CMakeLists.txt
+++ /examples/library-examples/link-error-odr-local-fix/boo/CMakeLists.txt
@@ -1,4 +1,4 @@
 # boo/CMakeLists.txt

-add_definitions(-DBOO_USE_SHORT_INT) # This is wrong!
 add_library(boo Boo.hpp Boo.cpp)
+target_compile_definitions(boo PUBLIC "BOO_USE_SHORT_INT")
```

Links fine now:

```
[library-examples]> rm -rf _builds
[library-examples]> cmake -Hlink-error-odr-local-fix -B_builds -DCMAKE_VERBOSE_
↪MAKEFILE=ON
...
[library-examples]> cmake --build _builds
/usr/bin/c++ -DBOO_USE_SHORT_INT /.../Boo.cpp
...
/usr/bin/c++ -DBOO_USE_SHORT_INT /.../Foo.cpp
...
/usr/bin/c++ -DBOO_USE_SHORT_INT /.../main.cpp
...
/usr/bin/c++ -rdynamic CMakeFiles/baz.dir/main.cpp.o -o baz foo/libfoo.a boo/libboo.a
...
> nm --defined-only --demangle _builds/boo/libboo.a
Boo.cpp.o:
0000000000000000 T Boo::boo(int, short)
> nm --undefined-only --demangle _builds/foo/libfoo.a
Foo.cpp.o:
    U Boo::boo(int, short)
```

ODR violation (global)

Examples on GitHub

- Repository
- Latest ZIP

Next code shows the ODR violation example based on the same `#ifdef` technique but the reason and solution will be different.

Assume we have library `boo` which can be used with both C++98 and C++11 standards:

```
// boo/Boo.hpp

#ifndef BOO_HPP_
```

```
#define BOO_HPP_

#if __cplusplus >= 201103L
# include <thread> // std::thread
#endif

class Boo {
 public:
#if __cplusplus >= 201103L
  typedef std::thread thread_type;
#else
  class InternalThread {
  };
  typedef InternalThread thread_type;
#endif
  static void boo(thread_type&);
};

#endif // BOO_HPP_
```

```
// boo/Boo.cpp

#include "boo/Boo.hpp"

#include <iostream> // std::cout

void Boo::boo(thread_type&) {
#if __cplusplus >= 201103L
  std::cout << "Boo: 2011" << std::endl;
#else
  std::cout << "Boo: 1998" << std::endl;
#endif
}
```

```
# boo/CMakeLists.txt

add_library(boo Boo.hpp Boo.cpp)
```

Library `foo` depends on `boo`:

```
// foo/Foo.hpp

#ifndef FOO_HPP_
#define FOO_HPP_

class Foo {
 public:
  static int foo();
};

#endif // FOO_HPP_
```

```
// foo/Foo.cpp

#include <foo/Foo.hpp>

#include <boo/Boo.hpp>
```

```
int Foo::foo() {
  Boo::thread_type t;
  Boo::boo(t);
  return 0;
}
```

Assuming that library `foo` use some C++11 features (this fact is not reflected in C++ code though) first that came to mind is to modify `CXX_STANDARD` property:

```
# foo/CMakeLists.txt

add_library(foo Foo.cpp Foo.hpp)
target_link_libraries(foo PUBLIC boo)

set_target_properties(foo PROPERTIES CXX_STANDARD 11)
```

Final executable:

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

include_directories("${CMAKE_CURRENT_LIST_DIR}") # for '#include <boo/Boo.hpp>'

add_subdirectory(boo)
add_subdirectory(foo)

add_executable(baz baz.cpp)
target_link_libraries(baz PUBLIC foo)
```

```
// baz.cpp

#include <iostream> // std::cout
#include <foo/Foo.hpp>

int main() {
  std::cout << "Foo: " << Foo::foo() << std::endl;
}
```

Link will fail for the same reason as with previous example. We are not using C++11 flags while building `boo` library but using C++11 flags while building `foo` and C++11 flag is analyzed in `boo/Boo.hpp` which is loaded by both targets:

```
[examples]> rm -rf _builds
[examples]> cmake -Hlibrary-examples/link-error-odr-global -B_builds
...
[examples]> cmake --build _builds
...
[100%] Linking CXX executable baz
foo/libfoo.a(Foo.cpp.o): In function `Foo::foo()':
Foo.cpp:(.text+0x52): undefined reference to `Boo::boo(std::thread&)'
collect2: error: ld returned 1 exit status
CMakeFiles/baz.dir/build.make:96: recipe for target 'baz' failed
make[2]: *** [baz] Error 1
```

Can this issue be fixed using the same approach as `target_compile_definitions(boo PUBLIC "BOO_USE_SHORT_INT")`? Note that if we set `set_target_properties(boo PROPERTIES CXX_STANDARD 11)` we can't use `boo` with the C++98 targets for the exact same reason, even if `boo` is designed to work with both standards.

The main difference here is that `BOO_USE_SHORT_INT` is **local** to the library `boo` and hence should be controlled locally (as shown before in `CMakeLists.txt` of `boo` library). Meanwhile C++98/C++11 flags are **global** and hence should be declared globally somewhere. In our simple case where all targets connected together in one project, we can add `CMAKE_CXX_STANDARD` to the configure step.

Removing local modification of `CXX_STANDARD`:

```
--- /examples/library-examples/link-error-odr-global/foo/CMakeLists.txt
+++ /examples/library-examples/link-error-odr-global-fix/foo/CMakeLists.txt
@@ -2,5 +2,3 @@

 add_library(foo Foo.cpp Foo.hpp)
 target_link_libraries(foo PUBLIC boo)
-
-set_target_properties(foo PROPERTIES CXX_STANDARD 11)
```

Building C++11 variant:

```
[examples]> rm -rf _builds
[examples]> cmake -Hlibrary-examples/link-error-odr-global-fix -B_builds -DCMAKE_CXX_
↪STANDARD=11
...
[examples]> cmake --build _builds
...
[examples]> ./_builds/baz
Boo: 2011
```

Building C++98 variant:

```
[examples]> rm -rf _builds
[examples]> cmake -Hlibrary-examples/link-error-odr-global-fix -B_builds -DCMAKE_CXX_
↪STANDARD=98
...
[examples]> cmake --build _builds
...
[examples]> ./_builds/baz
Boo: 1998
```

If we have more complex hierarchy of targets which are sequentially build/installed, we have to use same `CMAKE_CXX_STANDARD` value for each participating project. `CMAKE_CXX_STANDARD` is not the only property with global nature, it might be helpful to set all such properties/flags in one place - *toolchain*.

If you still want to set global flags locally for any reason then at least put the code under `if` condition. For example let's set C++11 for all targets in the project and C++14 for target `boo`:

```
if(NOT EXISTS "${CMAKE_TOOLCHAIN_FILE}")
  set(CMAKE_CXX_STANDARD 11) # set a global minimum standard
  set_target_properties(boo PROPERTIES CXX_STANDARD 14) # set a standard for a target
  # ...
endif()
```

Link order

GNU linker

This problem occurs only when you're using GNU linker. From `man ld` on Linux:

```
The linker will search an archive only once, at the location where it is
specified on the command line.  If the archive defines a symbol which was
undefined in some object which appeared before the archive on the command
line, the linker will include the appropriate file(s) from the archive.
However, an undefined symbol in an object appearing later on the command
line will not cause the linker to search the archive again.
```

There is no such issue on OSX for example, quote from `man ld`:

```
ld will only pull .o files out of a static library if needed to resolve
some symbol reference.  Unlike traditional linkers, ld will continually
search a static library while linking. There is no need to specify a
static library multiple times on the command line.
```

Example tested on Linux with GCC compiler and standard `ld` linker:

```
> ld --version
GNU ld (GNU Binutils for Ubuntu) 2.26.1
Copyright (C) 2015 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or (at your option) a later version.
This program has absolutely no warranty.

> gcc --version
gcc (Ubuntu 5.4.1-2ubuntu1~16.04) 5.4.1 20160904
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Problem

Example with two libraries `bar`, `boo` and executable `foo`:

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

add_library(bar bar.cpp)
add_library(boo boo.cpp)

add_executable(foo foo.cpp)
target_link_libraries(foo PUBLIC bar boo)
```

Library `bar` doesn't depend on anything and define function `int bar()`:

```
// bar.cpp

int bar() {
  return 0x42;
}
```

Library `boo` depends on `bar` and define function `int boo()`:

```
// boo.cpp

int bar();

int boo() {
  return bar();
}
```

Executable `foo` depends on `boo`:

```
// foo.cpp

int boo();

int main() {
  return boo();
}
```

Build will fail with linker error:

```
[examples]> rm -rf _builds
[examples]> cmake -Hlibrary-examples/link-order-bad -B_builds -DCMAKE_VERBOSE_
 ↪MAKEFILE=ON
-- The C compiler identification is GNU 5.4.1
-- The CXX compiler identification is GNU 5.4.1
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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: /.../examples/_builds
[examples]> cmake --build _builds
...
[ 16%] Building CXX object CMakeFiles/bar.dir/bar.cpp.o
/usr/bin/c++     -o CMakeFiles/bar.dir/bar.cpp.o -c /.../examples/library-examples/
 ↪link-order-bad/bar.cpp
[ 33%] Linking CXX static library libbar.a
...
/usr/bin/ar qc libbar.a  CMakeFiles/bar.dir/bar.cpp.o
/usr/bin/ranlib libbar.a
[ 33%] Built target bar
...
[ 50%] Building CXX object CMakeFiles/boo.dir/boo.cpp.o
/usr/bin/c++     -o CMakeFiles/boo.dir/boo.cpp.o -c /.../examples/library-examples/
 ↪link-order-bad/boo.cpp
[ 66%] Linking CXX static library libboo.a
...
/usr/bin/ar qc libboo.a  CMakeFiles/boo.dir/boo.cpp.o
```

```
/usr/bin/ranlib libboo.a
[ 66%] Built target boo
...
[ 83%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
/usr/bin/c++     -o CMakeFiles/foo.dir/foo.cpp.o -c /.../examples/library-examples/
↪link-order-bad/foo.cpp
[100%] Linking CXX executable foo
...
/usr/bin/c++    -rdynamic CMakeFiles/foo.dir/foo.cpp.o  -o foo libbar.a libboo.a
libboo.a(boo.cpp.o): In function `boo()':
boo.cpp:(.text+0x5): undefined reference to `bar()'
collect2: error: ld returned 1 exit status
...
```

Note that linker can't find symbol `int bar()` from `bar` library even if `libbar.a` is present in command line.

To understand the reason of error you have to understand how linker works:

- All files passed to linker processed from **left to right**
- Linker **collects undefined symbols** from files to the pool of undefined symbols
- If object from archive doesn't resolve any symbols from pool of undefined symbols, then **it dropped**

Next thing happens in example above:

- 3 files passed to linker to create final `foo` executable:
  - object `CMakeFiles/foo.dir/foo.cpp.o`
  - archive `libbar.a`
  - archive `libboo.a`
- `CMakeFiles/foo.dir/foo.cpp.o` has undefined symbol `int boo()`. Current pool of undefined symbols is `int boo()`
- Archive `libbar.a` defines `int bar()`, doesn't have any undefined symbols and doesn't resolve any symbols from pool. Hence **we drop it**. Current pool of undefined symbols is `int boo()`
- Archive `libboo.a` defines `int boo()` and has undefined symbol `int bar()`. `int boo()` removed from pool and `int bar()` added. Current pool of undefined symbols is `int bar()`
- No files left. Pool of undefined symbols is not empty and error about unresolved `int bar()` symbol reported.

Fix

To fix this you should declare dependency between `boo` and `bar`:

```
--- /examples/library-examples/link-order-bad/CMakeLists.txt
+++ /examples/library-examples/link-order-fix/CMakeLists.txt
@@ -5,6 +5,7 @@

 add_library(bar bar.cpp)
 add_library(boo boo.cpp)
+target_link_libraries(boo PUBLIC bar)

 add_executable(foo foo.cpp)
-target_link_libraries(foo PUBLIC bar boo)
+target_link_libraries(foo PUBLIC boo)
```

This approach both clean (`foo` doesn't explicitly depends on `bar`, why `target_link_libraries(foo PUBLIC bar)` used?) and correct - CMake will control the right order of files:

```
[examples]> rm -rf _builds
[examples]> cmake -Hlibrary-examples/link-order-fix -B_builds -DCMAKE_VERBOSE_
↪MAKEFILE=ON
-- The C compiler identification is GNU 5.4.1
-- The CXX compiler identification is GNU 5.4.1
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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: /.../examples/_builds
[examples]> cmake --build _builds
...
/usr/bin/c++    -rdynamic CMakeFiles/foo.dir/foo.cpp.o  -o foo libboo.a libbar.a
make[2]: Leaving directory '/.../examples/_builds'
[100%] Built target foo
make[1]: Leaving directory '/.../examples/_builds'
/.../bin/cmake -E cmake_progress_start /.../examples/_builds/CMakeFiles 0
```

Summary

- If one library depends on symbols from other library you have to express it by `target_link_libraries` command. Even if you may not have problems in current setup they may appear later or on another platform.
- If you have "undefined reference" error even if library with missing symbols is present in command line, then it may means that the order is not correct. Fix it by adding `target_link_libraries(boo PUBLIC bar)`, where `boo` is library with unresolved symbols and `bar` is library which defines those symbols.

3.12 Pseudo targets

3.12.1 Imported targets

3.12.2 Alias targets

3.12.3 Interface targets

3.13 Collecting sources

3.13.1 Avoid globbing

3.13.2 Project layout

Examples on GitHub

- Repository
- Latest ZIP

lib/	*<project>/*	**<project>.hpp**	
		<target>/	**CMakeLists.txt** with target `<project>_<target>`
			<target>.hpp
app/	*<project>/*	*<target>/*	**CMakeLists.txt** with target `<project>_<target>`
test/	*<project>/*	*<target>/*	**CMakeLists.txt** with target `<project>_<target>`
example/	*<project>/*	*<target>/*	**CMakeLists.txt** with target `<project>_<target>`
cmake/	module/	**<project>_<module>.cmake**	
	template/	***.cmake.in**	
	script/	***.cmake**	
	include/	***.cmake**	
	try_compile/	***.cpp**	

See also:

- *Install layout*

```
-- CMakeLists.txt
-- lib/
|   -- CMakeLists.txt
|   -- fruits/
|       -- CMakeLists.txt
|       -- fruits.hpp
|       -- rosaceae/
|       |   -- CMakeLists.txt
|       |   -- rosaceae.hpp
|       |   -- Pear.cpp
|       |   -- Pear.hpp
|       |   -- Plum.cpp
|       |   -- Plum.hpp
|       |   -- unittest/
|       |       -- Pear.cpp
|       -- tropical/
|           -- CMakeLists.txt
```

```
|           -- tropical.hpp
|           -- Avocado.cpp
|           -- Avocado.hpp
|           -- Pineapple.cpp
|           -- Pineapple.hpp
|           -- unittest/
|               -- Avocado.cpp
|               -- Pineapple.cpp
-- app/
|   -- CMakeLists.txt
|   -- fruits/
|       -- CMakeLists.txt
|       -- breakfast/
|       |   -- CMakeLists.txt
|       |   -- flatware/
|       |   |   -- Teaspoon.cpp
|       |   |   -- Teaspoon.hpp
|       |   -- main.cpp
|       -- dinner/
|           -- CMakeLists.txt
|           -- main.cpp
-- example/
|   -- CMakeLists.txt
|   -- fruits/
|       -- CMakeLists.txt
|       -- quick_meal/
|       |   -- CMakeLists.txt
|       |   -- main.cpp
|       -- vegan_party/
|           -- CMakeLists.txt
|           -- main.cpp
-- test/
    -- CMakeLists.txt
    -- fruits/
        -- CMakeLists.txt
        -- check_tropical/
        |   -- CMakeLists.txt
        |   -- data/
        |       -- avocado.ini
        -- skin_off/
            -- CMakeLists.txt
```

3.14 Usage requirements

3.14.1 Compile definitions

3.14.2 Include directories

3.14.3 Link libraries

3.15 Build types

3.15.1 Detect Multi/Single

```
string(COMPARE EQUAL "${CMAKE_CFG_INTDIR}" "." is_single)
if(is_single)
  message("Single-configuration generator")
else()
  message("Multi-configuration generator")
endif()
```

CMake documentation

- CMAKE_CFG_INTDIR

Warning: `if(XCODE OR MSVC)` condition doesn't work because `MSVC` **defined** for NMake single-configuration generator too.

Warning: `if(XCODE OR MSVC_IDE)` condition doesn't work because `MSVC_IDE` is **not defined** for Visual Studio MDD toolchain.

3.16 configure_file

3.17 Install

The next step in chain of *Configure* → *Generate* → *Build* → *Test* stages is **install**: final step of development process which often require privilege escalation (`make` vs `sudo make install`). Installation is an important part of the ecosystem: results of the project installation allows to integrate it into another project using `find_package` and unlike `add_subdirectory` doesn't pollute current scope with unnecessary targets and variables. *Packing* use install procedure under the hood.

See also:

- *CMake stages*
- *Stages diagram*

Examples on GitHub

- Repository
- Latest ZIP

3.17.1 Library

TODO

ALIAS: Unify interface for both `find_package` and `add_subdirectory`

3.17.2 Header-only library

TODO

INTERFACE

3.17.3 Library with dependencies

TODO

`find_dependency` in `Config.cmake.in`

3.17.4 Optional dependencies

TODO

`find_dependency(baz CONFIG)` under condition `if("@FOO_WITH_BAZ@")`

3.17.5 CMake modules

3.17.6 Export header

CMake documentation

- GenerateExportHeader

3.17.7 RPATH

CMake wiki

- RPATH handling

Wikipedia

- RPATH

3.17.8 Version

CMake documentation

- write_basic_package_version_file

3.17.9 CMAKE_INSTALL_PREFIX

CMake documentation

- CMAKE_INSTALL_PREFIX

CMAKE_INSTALL_PREFIX variable can be used to control destination directory of install procedure:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

add_library(foo foo.cpp)

install(TARGETS foo DESTINATION lib)
```

```
[install-examples]> rm -rf _builds
[install-examples]> cmake -Hsimple -B_builds -DCMAKE_INSTALL_PREFIX=_install/config-A
[install-examples]> cmake --build _builds --target install
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX static library libfoo.a
[100%] Built target foo
Install the project...
-- Install configuration: ""
-- Installing: /.../install-examples/_install/config-A/lib/libfoo.a

[install-examples]> cmake -Hsimple -B_builds -DCMAKE_INSTALL_PREFIX=_install/config-B
[install-examples]> cmake --build _builds --target install
[100%] Built target foo
Install the project...
-- Install configuration: ""
-- Installing: /.../install-examples/_install/config-B/lib/libfoo.a
```

Modify

This variable is designed to be modified on user side. Do not force it in code!

```
cmake_minimum_required(VERSION 2.8)
project(foo)

set(CMAKE_INSTALL_PREFIX "${CMAKE_CURRENT_BINARY_DIR}/3rdParty/root") # BAD CODE!

add_library(foo foo.cpp)

install(TARGETS foo DESTINATION lib)
```

```
[install-examples]> rm -rf _builds
[install-examples]> cmake -Hmodify-bad -B_builds -DCMAKE_INSTALL_PREFIX="`pwd`/_
↪install"
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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: /.../install-examples/_builds
```

Library unexpectedly installed to `3rdparty/root` instead of `_install`:

```
[install-examples]> cmake --build _builds --target install
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX static library libfoo.a
[100%] Built target foo
Install the project...
-- Install configuration: ""
-- Installing: /.../install-examples/_builds/3rdParty/root/lib/libfoo.a
```

Note: Use *CACHE* in such case

On the fly

`Make` do support changing of install directory on the fly by `DESTDIR`:

```
[install-examples]> rm -rf _builds
[install-examples]> cmake -Hsimple -B_builds -DCMAKE_INSTALL_PREFIX=""
[install-examples]> make -C _builds DESTDIR="`pwd`/_install/config-A" install
...
```

```
Install the project...
-- Install configuration: ""
-- Installing: /.../install-examples/_install/config-A/lib/libfoo.a
make: Leaving directory '/.../install-examples/_builds'

[install-examples]> make -C _builds DESTDIR="`pwd`/_install/config-B" install
...
Install the project...
-- Install configuration: ""
-- Installing: /.../install-examples/_install/config-B/lib/libfoo.a
make: Leaving directory '/.../install-examples/_builds'
```

Read

Because of the `DESTDIR` feature, CPack functionality, different nature of build and install stages often usage of `CMAKE_INSTALL_PREFIX` variable on configure step is an indicator of wrongly written code:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

add_library(foo foo.cpp)

install(TARGETS foo DESTINATION lib)

# BAD CODE!
file(
    COPY
    "${CMAKE_CURRENT_LIST_DIR}/README"
    DESTINATION
    "${CMAKE_INSTALL_PREFIX}/share/foo"
)

include(CPack)
```

User may not want to install such project at all, so copying of file to root is something unintended and quite surprising. If you're lucky you will get problems with permissions on configure step instead of a silent copy:

```
[install-examples]> rm -rf _builds
[install-examples]> cmake -Hwrong-usage -B_builds
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
CMake Error at CMakeLists.txt:9 (file):
  file COPY cannot copy file
  "/.../install-examples/wrong-usage/README"
```

```
  to "/usr/local/share/foo/README".

-- Configuring incomplete, errors occurred!
See also "/.../install-examples/_builds/CMakeFiles/CMakeOutput.log".
```

CPack will use separate directory for install so `README` will not be included in archive:

```
[install-examples]> rm -rf _builds _install
[install-examples]> cmake -Hwrong-usage -B_builds -DCMAKE_INSTALL_PREFIX="`pwd`/_
↪install"
[install-examples]> (cd _builds && cpack -G TGZ)
CPack: Create package using TGZ
CPack: Install projects
CPack: - Run preinstall target for: foo
CPack: - Install project: foo
CPack: Create package
CPack: - package: /.../install-examples/_builds/foo-0.1.1-Linux.tar.gz generated.
[install-examples]> tar xf _builds/foo-0.1.1-Linux.tar.gz
[install-examples]> find foo-0.1.1-Linux -type f
foo-0.1.1-Linux/lib/libfoo.a
```

Implicit read

All work should be delegated to `install` command instead, in such case `CMAKE_INSTALL_PREFIX` will be read implicitly:

```
cmake_minimum_required(VERSION 2.8)
project(foo)

add_library(foo foo.cpp)

install(TARGETS foo DESTINATION lib)
install(FILES README DESTINATION share/foo)

include(CPack)
```

```
[install-examples]> rm -rf _builds _install
[install-examples]> cmake -Hright-usage -B_builds -DCMAKE_INSTALL_PREFIX="`pwd`/_
↪install"
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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: /.../install-examples/_builds
```

Correct install directory:

```
[install-examples]> cmake --build _builds --target install
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX static library libfoo.a
[100%] Built target foo
Install the project...
-- Install configuration: ""
-- Installing: /.../install-examples/_install/lib/libfoo.a
-- Installing: /.../install-examples/_install/share/foo/README
```

Correct packing:

```
[install-examples]> (cd _builds && cpack -G TGZ)
CPack: Create package using TGZ
CPack: Install projects
CPack: - Run preinstall target for: foo
CPack: - Install project: foo
CPack: Create package
CPack: - package: /.../install-examples/_builds/foo-0.1.1-Linux.tar.gz generated.
[install-examples]> tar xf _builds/foo-0.1.1-Linux.tar.gz
[install-examples]> find foo-0.1.1-Linux -type f
foo-0.1.1-Linux/share/foo/README
foo-0.1.1-Linux/lib/libfoo.a
```

Install script

Same logic can be applied if `CMAKE_INSTALL_PREFIX` used in script created by `configure_file` command:

```
# Top-level CMakeLists.txt

cmake_minimum_required(VERSION 2.8)
project(foo)

set(script "${CMAKE_CURRENT_BINARY_DIR}/script.cmake")
configure_file(script.cmake.in "${script}" @ONLY)

install(SCRIPT "${script}")

include(CPack)
```

```
# script.cmake.in

cmake_minimum_required(VERSION 2.8)

set(correct "$ENV{DESTDIR}${CMAKE_INSTALL_PREFIX}")

message("Incorrect value: '@CMAKE_INSTALL_PREFIX@'")
message("Correct value: '${correct}'")

file(WRITE "${correct}/share/foo/info" "Some info")
```

Configure for `DESTDIR` usage:

```
[install-examples]> rm -rf _builds _install foo-0.1.1-Linux
[install-examples]> cmake -Hconfigure -B_builds -DCMAKE_INSTALL_PREFIX=""
```

```
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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: /.../install-examples/_builds
```

`DESTDIR` read correctly:

```
[install-examples]> make DESTDIR="`pwd`/_install/config-A" -C _builds install
make: Entering directory '/.../install-examples/_builds'
Install the project...
-- Install configuration: ""
Incorrect value: ''
Correct value: '/.../install-examples/_install/config-A'
make: Leaving directory '/.../install-examples/_builds'
[install-examples]> find _install/config-A -type f
_install/config-A/share/foo/info
```

Changing directory on the fly:

```
[install-examples]> make DESTDIR="`pwd`/_install/config-B" -C _builds install
make: Entering directory '/.../install-examples/_builds'
Install the project...
-- Install configuration: ""
Incorrect value: ''
Correct value: '/.../install-examples/_install/config-B'
make: Leaving directory '/.../install-examples/_builds'
[install-examples]> find _install/config-B -type f
_install/config-B/share/foo/info
```

Regular install:

```
[install-examples]> rm -rf _builds _install
[install-examples]> cmake -Hconfigure -B_builds -DCMAKE_INSTALL_PREFIX="`pwd`/_install
↪"
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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: /.../install-examples/_builds
[install-examples]> cmake --build _builds --target install
Install the project...
-- Install configuration: ""
Incorrect value: '/.../install-examples/_install'
Correct value: '/.../install-examples/_install'
[install-examples]> find _install -type f
_install/share/foo/info
```

Packing:

```
[install-examples]> (cd _builds && cpack -G TGZ)
CPack: Create package using TGZ
CPack: Install projects
CPack: - Run preinstall target for: foo
CPack: - Install project: foo
Incorrect value: '/.../install-examples/_install'
Correct value: '/.../install-examples/_builds/_CPack_Packages/Linux/TGZ/foo-0.1.1-
↪Linux'
CPack: Create package
CPack: - package: /.../install-examples/_builds/foo-0.1.1-Linux.tar.gz generated.
[install-examples]> tar xf _builds/foo-0.1.1-Linux.tar.gz
[install-examples]> find foo-0.1.1-Linux -type f
foo-0.1.1-Linux/share/foo/info
```

Summary

- Do not **force** value of `CMAKE_INSTALL_PREFIX`
- Use of `CMAKE_INSTALL_PREFIX` on configure, generate, build steps is an indicator of badly designed code
- Use `install` instead of `CMAKE_INSTALL_PREFIX`
- Respect `DESTDIR`

3.17.10 Layout

include/	*<project>/*	**<project>.hpp**	
lib*/	**<project>_<target>**		
	cmake/	*<project>/*	**<project>Config.cmake**
bin/	**<project>_<target>**		
cmake/	module/	**<project>_<module>.cmake**	
	template/	*<project>/*	***.cmake.in**
	script/	*<project>/*	***.cmake**
	include/	*<project>/*	***.cmake**

```
include(GNUInstallDirs)

install(
    TARGETS <project>_<target>_1 <project>_<target>_2
```

```
    EXPORT <project>Targets
    LIBRARY DESTINATION "${CMAKE_INSTALL_LIBDIR}"
    ARCHIVE DESTINATION "${CMAKE_INSTALL_LIBDIR}"
    RUNTIME DESTINATION "${CMAKE_INSTALL_BINDIR}"
    INCLUDES DESTINATION "${CMAKE_INSTALL_INCLUDEDIR}"
)
```

See also:

- *Project layout*

CMake documentation

- GNUInstallDirs

Linux layout after installation of example project:

```
-- bin
|   -- fruits_breakfast*
|   -- fruits_dinner*
-- include
|   -- fruits
|       -- fruits.hpp
|       -- FRUITS_ROSACEAE_EXPORT.h
|       -- FRUITS_TROPICAL_EXPORT.h
|       -- rosaceae
|       |   -- Pear.hpp
|       |   -- Plum.hpp
|       |   -- rosaceae.hpp
|       -- tropical
|           -- Avocado.hpp
|           -- Pineapple.hpp
|           -- tropical.hpp
-- lib
    -- cmake
    |   -- fruits
    |       -- fruitsConfig.cmake
    |       -- fruitsConfigVersion.cmake
    |       -- fruitsTargets.cmake
    |       -- fruitsTargets-release.cmake
    -- libfruits_rosaceae.a
    -- libfruits_tropical.a
```

Windows layout after installation of example project:

```
-- bin
|   -- fruits_breakfast.exe
|   -- fruits_dinner.exe
-- include
|   -- fruits
|       -- fruits.hpp
|       -- FRUITS_ROSACEAE_EXPORT.h
|       -- FRUITS_TROPICAL_EXPORT.h
|       -- rosaceae
|       |   -- Pear.hpp
|       |   -- Plum.hpp
|       |   -- rosaceae.hpp
```

```
|       -- tropical
|           -- Avocado.hpp
|           -- Pineapple.hpp
|           -- tropical.hpp
-- lib
    -- cmake
    |   -- fruits
    |       -- fruitsConfig.cmake
    |       -- fruitsConfigVersion.cmake
    |       -- fruitsTargets.cmake
    |       -- fruitsTargets-release.cmake
    -- fruits_rosaceae.lib
    -- fruits_tropical.lib
```

Windows Debug + DLL:

```
-- bin
|   -- fruits_breakfast.exe
|   -- fruits_breakfast.pdb
|   -- fruits_dinner.exe
|   -- fruits_dinner.pdb
|   -- fruits_rosaceaed.dll
|   -- fruits_rosaceaed.pdb
|   -- fruits_tropicald.dll
|   -- fruits_tropicald.pdb
-- include
|   -- fruits
|       -- fruits.hpp
|       -- FRUITS_ROSACEAE_EXPORT.h
|       -- FRUITS_TROPICAL_EXPORT.h
|       -- rosaceae
|       |   -- Pear.hpp
|       |   -- Plum.hpp
|       |   -- rosaceae.hpp
|       -- tropical
|           -- Avocado.hpp
|           -- Pineapple.hpp
|           -- tropical.hpp
-- lib
    -- cmake
    |   -- fruits
    |       -- fruitsConfig.cmake
    |       -- fruitsConfigVersion.cmake
    |       -- fruitsTargets.cmake
    |       -- fruitsTargets-debug.cmake
    -- fruits_rosaceaed.lib
    -- fruits_tropicald.lib
```

3.17.11 Samples

- Install library. TODO: adapt https://github.com/forexample/package-example
- Header-only library
- Install library, optional dependencies (system ZLIB)
- https://github.com/cgold-examples/fruits

- optional dependencies (Hunter)
- version
- CMake modules

3.17.12 Managing dependencies

There are a lot of different ways to deal with dependencies. We start with widely used anti-patterns and explain why they are anti-patterns. The second part will contain an examples of good approaches and list of requirements that any other package manager should satisfy.

Bad way

Merge sources

Assume we have library `foo`:

```
src
-- foo
    -- foo.cpp
    -- foo.hpp
```

Library `foo` depends on library `boo`:

```
src
-- boo
    -- boo.cpp
    -- boo.hpp
```

The worst thing you can do is to merge both sources by copying `boo` to the directory with `foo`:

```
src
-- boo
|   -- boo.cpp
|   -- boo.hpp
-- foo
    -- foo.cpp
    -- foo.hpp
```

C++ directive `#include <foo/foo.hpp>` means that directory `src/` should be in list of paths to headers (in terms of compilers like GCC: `-I/.../src`). We want our local directory to have **highest priority** if there are several paths (e.g. if there are system wide paths and another copy of library `foo` installed system wide, then we want local copy to take a priority). Hence whatever the rest of header paths is, the `#include <boo/boo.hpp>` of dependent library `boo` will **always fall to our local copy**. If somebody want to try to use another version of `boo` the only choice you left to him is to remove `src/boo` directory.

Copy to "third_party" directory

Instead of copying to the same directory you can copy dependent library to `third_party` directory:

```
src
-- foo
    -- foo.cpp
```

```
    -- foo.hpp
third_party
-- boo
    -- boo.cpp
    -- boo.hpp
```

There will be two independent paths to headers (at least): `-I/.../src` and `-I/.../third_party` hence if somebody want to try to use another version of `boo` it's enough to modify CMake code without changing project structure.

Assuming that both libraries are under *VCS* control, by doing plain copy operation you're **losing information about version** of `boo`. Also if you want to modify `boo` sources locally, then merging them with update of `boo` from upstream might be not a trivial operation.

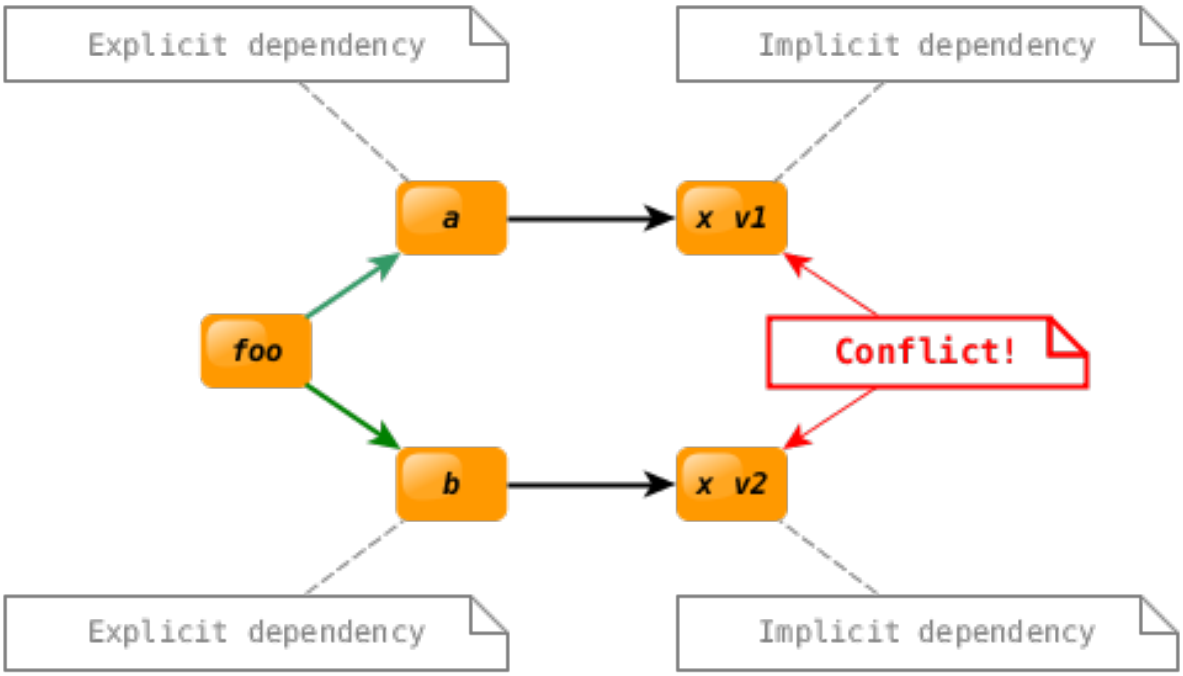
Git submodule

Using same structure you can keep information about version of `boo` by adding it as a Git submodule instead of raw copying. Git functionality will help to track modification of third party code and merging it with future releases.

Git submodules will work well for:

- Adding sources that are extension of your project, hence it makes no sense to add these sources to another project. I.e. submodule is used as a **dependency exactly for one node in dependency tree**.
- Managing dependencies in the project that can't be used as another dependency, i.e. **your project is the leaf in dependency tree**, like executable. Though this still may raise the question of size optimization when package manager is used, it will be better to use dependencies as a shared libraries.
- **Short period** of development when you're actively modifying third party code.

Submodule is not a good long term solution for managing dependencies that can be reused. It leads to conflicts like this:



Note: If we are talking about reusable library then you can't control final structure of dependency tree. If you are not experiencing such issue it doesn't mean the same will be true for somebody else.

At first you will simply get *target names conflict*:

```
[examples]> rm -rf _builds
[examples]> cmake -Hdep-examples/deps-submodule-conflict -B_builds
-- The C compiler identification is GNU 5.4.1
-- The CXX compiler identification is GNU 5.4.1
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
CMake Error at third_party/b/third_party/x/CMakeLists.txt:4 (add_library):
  add_library cannot create target "x" because another target with the same
  name already exists.  The existing target is a static library created in
  source directory "/.../third_party/a/third_party/x".
```

You can protect including of third party code by condition:

```
if(NOT TARGET x::x)
  add_subdirectory(third_party/x)
endif()
```

It will solve target name conflict however it may lead to tricky behavior. Effectively it will introduce "first win" strategy while doing dependency resolution, mixing two separate concepts:

- Project structure
  - `foo` depends on `a` and `b`
  - `a` depends on `x`
  - `b` depends on `x`
- Versions
  - Some `a` version available
  - Some `b` version available
  - Two versions of `x` available: `v1.0` and `v2.0`

Options is a common way to customize your CMake code, often it's involve the change of used dependencies and change of project structure. Let's add option `FOO_WITH_A` to the example to control optional dependency `foo` -> `a`:

```
cmake_minimum_required(VERSION 3.2)
project(foo)

option(FOO_WITH_A "Use 'a' module" ON)

add_executable(foo foo.cpp)
```

```
if(FOO_WITH_A)
  add_subdirectory(third_party/a)
  target_link_libraries(foo PUBLIC a::a)
  target_compile_definitions(foo PUBLIC FOO_WITH_A)
endif()

add_subdirectory(third_party/b)
target_link_libraries(foo PUBLIC b::b)

enable_testing()
add_test(NAME foo COMMAND foo)
```

When option `FOO_WITH_A` is enabled the `x` dependency from `a` subdirectory will proceed first, hence `v1.0` will be used. And if `FOO_WITH_A` is disabled the `x` dependency from `b` subdirectory will proceed first, hence `v2.0` will be used.

From user perspective it can be quite surprising and may look like some `a` functionality is leaking into module `b`:

```
[examples]> rm -rf _builds
[examples]> cmake -Hdep-examples/deps-submodule-option -B_builds -DFOO_WITH_A=ON
...
[examples]> cmake --build _builds
...
[examples]> cd _builds
[examples/_builds]> ctest -V
1: Test command: /.../examples/_builds/foo
1: Test timeout computed to be: 9.99988e+06
1: Running 'a' module
1: x say: nice
1: Running 'b' module
1: x say: nice
1/1 Test #1: foo ..............................   Passed    0.00 sec
```

Disable module `a` and behavior of `b` changed!

```
[examples]> rm -rf _builds
[examples]> cmake -Hdep-examples/deps-submodule-option -B_builds -DFOO_WITH_A=OFF
...
[examples]> cmake --build _builds
...
[examples]> cd _builds
[examples/_builds]> ctest -V
1: Test command: /.../examples/_builds/foo
1: Test timeout computed to be: 9.99988e+06
1: Running 'a' module
1: (Module 'a' disabled)
1: Running 'b' module
1: x say: good
1/1 Test #1: foo ..............................   Passed    0.00 sec
```

Note: Such behavior can be "stabilized" by adding `foo` -> `x` dependency:

```
# before 'a' and 'b'
add_subdirectory(third_party/x)

if(FOO_WITH_A)
```

```
  add_subdirectory(third_party/a)
endif()
add_subdirectory(third_party/b)
```

But this obviously doesn't scale well since `x` is an implicit dependency and we have no control over whether it will be used in future `a`/`b` releases or more dependencies will be introduced or on which options/platforms they depends, etc.

Since version of `x` tied to project structure every time you switch option `FOO_WITH_A` the whole project will rebuild:

```
[examples]> rm -rf _builds
[examples]> cmake -Hdep-examples/deps-submodule-option -B_builds -DFOO_WITH_A=ON
```

Build everything from scratch first time:

```
[examples]> cmake --build _builds
Scanning dependencies of target x
[ 12%] Building CXX object third_party/a/third_party/x/CMakeFiles/x.dir/x/x.cpp.o
[ 25%] Linking CXX static library libx.a
[ 25%] Built target x
Scanning dependencies of target b
[ 37%] Building CXX object third_party/b/CMakeFiles/b.dir/b/b.cpp.o
[ 50%] Linking CXX static library libb.a
[ 50%] Built target b
Scanning dependencies of target a
[ 62%] Building CXX object third_party/a/CMakeFiles/a.dir/a/a.cpp.o
[ 75%] Linking CXX static library liba.a
[ 75%] Built target a
Scanning dependencies of target foo
[ 87%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX executable foo
[100%] Built target foo
```

Second time just checking:

```
[examples]> cmake --build _builds
[ 25%] Built target x
[ 50%] Built target b
[ 75%] Built target a
[100%] Built target foo
```

Disable component `a`:

```
[examples]> cmake -Hdep-examples/deps-submodule-option -B_builds -DFOO_WITH_A=OFF
```

Whole project rebuild!

```
[examples]> cmake --build _builds
Scanning dependencies of target x
[ 16%] Building CXX object third_party/b/third_party/x/CMakeFiles/x.dir/x/x.cpp.o
[ 33%] Linking CXX static library libx.a
[ 33%] Built target x
Scanning dependencies of target b
[ 50%] Building CXX object third_party/b/CMakeFiles/b.dir/b/b.cpp.o
[ 66%] Linking CXX static library libb.a
[ 66%] Built target b
Scanning dependencies of target foo
[ 83%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
```

```
[100%] Linking CXX executable foo
[100%] Built target foo
```

Summary

The best way to introduce bundled dependency is to add it to the separate directory like `third_party` as a submodule.

The downsides of such approach:

- `add_subdirectory` is **not a shareable solution**. Each `add_subdirectory(third_party/x)` block from different projects has it's own copy of `x` artifacts. Every time you start new project and add `add_subdirectory(third_party/x)` you're building `x` from scratch. It's not convenient if such build takes a long time.
- Dependency resolution **is not option friendly**.

See also:

- Why not bundle dependencies

Good way

Package manager

Use system package manager to manage `a` and `b` dependencies. Install them to your system and then integrate into CMake using find_package:

```
cmake_minimum_required(VERSION 3.2)
project(foo)

option(FOO_WITH_A "Use 'a' module" ON)

add_executable(foo foo.cpp)

if(FOO_WITH_A)
  find_package(a CONFIG REQUIRED)
  target_link_libraries(foo PUBLIC a::a)
  target_compile_definitions(foo PUBLIC FOO_WITH_A)
endif()

find_package(b CONFIG REQUIRED)
target_link_libraries(foo PUBLIC b::b)

enable_testing()
add_test(NAME foo COMMAND foo)
```

```
[examples]> rm -rf _builds
[examples]> cmake -Hdep-examples/deps-find-package -B_builds -DFOO_WITH_A=ON
[examples]> cmake --build _builds
```

Result of running test with module `a` enabled:

```
[examples]> cd _builds
[examples/_builds]> ctest -V
```

```
1: Test command: /.../_builds/foo
1: Test timeout computed to be: 9.99988e+06
1: Running 'a' module
1: x say: nice
1: Running 'b' module
1: x say: nice
1/1 Test #1: foo ..............................   Passed    0.00 sec
```

With module `a` disabled:

```
[examples]> cmake -Hdep-examples/deps-find-package -B_builds -DFOO_WITH_A=OFF
```

Third parties remains the same of course, only `foo` executable rebuild:

```
[examples]> cmake --build _builds
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX executable foo
[100%] Built target foo
```

Behavior of module `b` is the same:

```
[examples]> cd _builds
[examples/_builds]> ctest -V
1: Test command: /.../_builds/foo
1: Test timeout computed to be: 9.99988e+06
1: Running 'a' module
1: (Module 'a' disabled)
1: Running 'b' module
1: x say: nice
1/1 Test #1: foo ..............................   Passed    0.00 sec
```

Pros:

- **Locally shareable**. Root directory with libraries can be reused by any number of local project.
- **Globally shareable**. Usually dependencies distributed as binaries shared across many local machines. You don't have to build all dependencies from sources.
- **Option friendly**. Whatever options you've enabled the same set of third parties will be used.

Cons:

- **Not much customization** over third party dependencies
- Different system package managers have **different set of packages** and available versions
- Usually only **one root** directory

ExternalProject_Add

With the help of ExternalProject_Add module you can create so-called "super-build" project with dependencies:

```
cmake_minimum_required(VERSION 3.2)
project(super-build-example)

include(ExternalProject)

ExternalProject_Add(
```

```
    x
    URL https://github.com/cgold-examples/x/archive/v1.0.tar.gz
    URL_HASH SHA1=3c15777fddee4fbf41a57241effc59a821562f65
    CMAKE_ARGS -DCMAKE_INSTALL_PREFIX=${CMAKE_INSTALL_PREFIX}
)

ExternalProject_Add(
    a
    URL https://github.com/cgold-examples/a/archive/v1.0.tar.gz
    URL_HASH SHA1=9adb3574369cf3c186b4984eb6778fca5866e347
    CMAKE_ARGS -DCMAKE_INSTALL_PREFIX=${CMAKE_INSTALL_PREFIX}
    DEPENDS x
)

ExternalProject_Add(
    b
    URL https://github.com/cgold-examples/b/archive/v1.0.tar.gz
    URL_HASH SHA1=7ef127ddc31d6a9b510d9cdc318c68c7709a8204
    CMAKE_ARGS -DCMAKE_INSTALL_PREFIX=${CMAKE_INSTALL_PREFIX}
    DEPENDS x
)
```

Using such project you can install all dependencies to some custom root `_ep_install` directory:

```
[examples]> rm -rf _ep_build
[examples]> cmake -Hdep-examples/deps-super-build -B_ep_build -DCMAKE_INSTALL_PREFIX=_
↪ep_install
[examples]> cmake --build _ep_build
...
-- Downloading...
   dst='/.../examples/_ep_build/x-prefix/src/v1.0.tar.gz'
   timeout='none'
-- Using src='https://github.com/cgold-examples/x/archive/v1.0.tar.gz'
...
Install the project...
-- Install configuration: ""
-- Installing: /.../_ep_install/lib/libx.a
-- Installing: /.../_ep_install/include/x/x.hpp
-- Installing: /.../_ep_install/lib/cmake/x/xConfig.cmake
-- Installing: /.../_ep_install/lib/cmake/x/xTargets.cmake
-- Installing: /.../_ep_install/lib/cmake/x/xTargets-noconfig.cmake
...
-- Downloading...
   dst='/.../examples/_ep_build/a-prefix/src/v1.0.tar.gz'
   timeout='none'
-- Using src='https://github.com/cgold-examples/a/archive/v1.0.tar.gz'
...
Install the project...
-- Install configuration: ""
-- Installing: /.../_ep_install/lib/liba.a
-- Installing: /.../_ep_install/include/a/a.hpp
-- Installing: /.../_ep_install/lib/cmake/a/aConfig.cmake
-- Installing: /.../_ep_install/lib/cmake/a/aTargets.cmake
-- Installing: /.../_ep_install/lib/cmake/a/aTargets-noconfig.cmake
...
-- Downloading...
   dst='/.../examples/_ep_build/b-prefix/src/v1.0.tar.gz'
   timeout='none'
```

```
-- Using src='https://github.com/cgold-examples/b/archive/v1.0.tar.gz'
...
Install the project...
-- Install configuration: ""
-- Installing: /.../_ep_install/lib/libb.a
-- Installing: /.../_ep_install/include/b/b.hpp
-- Installing: /.../_ep_install/lib/cmake/b/bConfig.cmake
-- Installing: /.../_ep_install/lib/cmake/b/bTargets.cmake
-- Installing: /.../_ep_install/lib/cmake/b/bTargets-noconfig.cmake
```

Now you can use same `deps-find-package` example and inject `_ep_install` root directory **with your custom dependencies** instead of system dependencies:

```
[examples]> rm -rf _builds
[examples]> cmake -Hdep-examples/deps-find-package -B_builds -DCMAKE_PREFIX_PATH=/.../
↪examples/_ep_install -DCMAKE_VERBOSE_MAKEFILE=ON
[examples]> cmake --build _builds
/usr/bin/c++ ... -o foo
    /.../_ep_install/lib/liba.a
    /.../_ep_install/lib/libb.a
    /.../_ep_install/lib/libx.a
```

Pros:

- **Locally shareable**. Root directory with libraries can be reused by any number of local project.
- **Option friendly**. Whatever options you've enabled the same set of third parties will be used.
- **Third party customization**. You have full control over your dependencies.
- **Same set of packages** across all platforms.
- You can create **as many independent root directories as you need**.

Cons:

- Only **build from sources**. There is no built-in mechanism for supporting distribution of binaries and meta information. Usually user have to build everything from scratch on new machine.
- You have to know everything about your dependencies and carefully manage the build order, including implicit dependencies. For example if project `a` depends on `x` optionally you have to do something like this:

```
option(EP_A_WITH_X "Enable A_WITH_X for 'a' package" ON)

if(EP_A_WITH_X)
  # We need 'x' project
  ExternalProject_Add(
      x
      ...
  )
  set(a_dependencies x)
endif()

ExternalProject_Add(
    a
    ...
    CMAKE_ARGS -DA_WITH_X=${EP_A_WITH_X}
    DEPENDS ${a_dependencies}
)
```

  If dependency tree is complex it can be hard to maintain such super-build.

- Writing correct customizable `ExternalProject_Add` rules is not a trivial task. Take a look at real working code for example (and it really is just a tip of the iceberg).

Requirements

Good dependency management system should satisfy next requirements:

- **Locally shareable**. Root directory with libraries should be easily reused by any number of local project. CMake has find_package facility for injecting code into project and semi-automatic generation of `XXXConfig.cmake` configs for consumer (see Creating packages). Dependency management system should be friendly to this approach.
- **Globally shareable**. For the performance purposes there should be an ability to reuse binaries without building them from sources.
- **Option friendly**. Whatever options you've enabled the same set of third parties should be used.
- **Third party customization**. You should have an ability to control the way how third party code built: CMake options, CMake build types, compiler flags, etc.

3.18 Toolchain

CMake documentation

- CMake toolchains

3.18.1 Globals

Even if toolchain is originally designed to help with cross-compiling and usually containing fancy variables like CMAKE_SYSTEM_NAME or CMAKE_FIND_ROOT_PATH in practice it can help you with holding compiler settings that logically doesn't belong to some particular local `CMakeLists.txt` but rather should be shared across various projects.

C++ standard

C++ standard flags should be set globally. You should avoid using any commands that set it locally for target or project.

Note: Example tested with GCC 5.4.1 on Linux. Different compilers may work with C++ standards differently.

Examples on GitHub

- Repository
- Latest ZIP

Example

Let's assume we have header-only library `boo` implemented by `Boo.hpp` which can work with both C++98 and C++11:

```
// Boo.hpp

#ifndef BOO_HPP_
#define BOO_HPP_

#if __cplusplus >= 201103L
# include <thread>
#endif

class Boo {
 public:
#if __cplusplus >= 201103L
  typedef std::thread thread_type;
  static void call(thread_type&) {
  }
#else
  class InternalThread {
  };
  typedef InternalThread thread_type;
  static void call(thread_type&) {
  }
#endif
};

#endif // BOO_HPP_
```

Library `foo` that depends on `boo` and use C++11 **internally**:

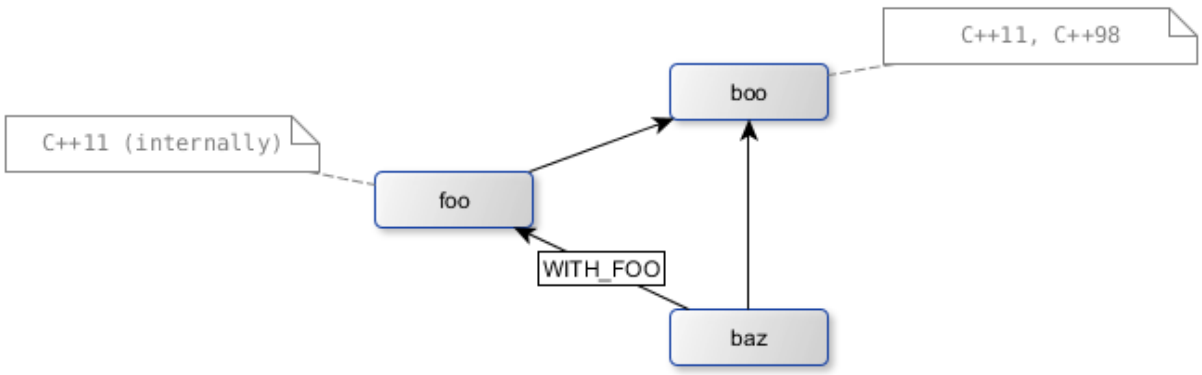
```
// Foo.hpp

#ifndef FOO_HPP_
#define FOO_HPP_

#include <Boo.hpp>

class Foo {
 public:
  static int optimize(Boo::thread_type&);
};

#endif // FOO_HPP_
```

```
// Foo.cpp

#include <Foo.hpp>

constexpr int foo_helper_value() {
  return 0x42;
}

int Foo::optimize(Boo::thread_type&) {
  return foo_helper_value();
}
```

Executable `baz` knows nothing about standards and just use API of `Boo` and `Foo` classes, `Foo` is optional:

```
// main.cpp

#include <iostream> // std::cout

#include <Boo.hpp>

#ifdef WITH_FOO
# include <Foo.hpp>
#endif

int main() {
  Boo::thread_type t;

  std::cout << "C++ standard: " << __cplusplus << std::endl;

#ifdef WITH_FOO
  std::cout << "With Foo support" << std::endl;
  Foo::optimize(t);
#endif

  Boo::call(t);
}
```

Graphically it will look like this:



CMake project :

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.1)
project(foo)

add_library(boo INTERFACE)
target_include_directories(boo INTERFACE "$<BUILD_INTERFACE:${CMAKE_CURRENT_LIST_DIR}>
↪")

add_executable(baz main.cpp)
target_link_libraries(baz PUBLIC boo)

if(WITH_FOO)
```

```
  add_library(foo Foo.cpp Foo.hpp)
  target_link_libraries(foo PUBLIC boo)
  target_link_libraries(baz PUBLIC foo)
  target_compile_definitions(baz PUBLIC WITH_FOO)
endif()
```

Overview:

- `boo` provide same API for both C++11 and C++98 configuration so user don't have to worry about standards.
- `foo` use some C++11 feature but only internally.
- `baz` don't know anything about used standards, interested only in `boo` or `foo` API.

Imagine that `baz` for the long time relies only on `boo`, it's important to keep this functionality even for old C++98 configuration. But there is `foo` library that use C++11 and allow us to introduce some optimization.

We want:

- C++11 with `foo`
- C++11 without `foo`
- C++98 with `foo` should produce error message as soon as possible
- C++98 without `foo`

Bad

The first thing that comes to mind after looking at C++ code is that since `foo` use `constexpr` feature internally we should do:

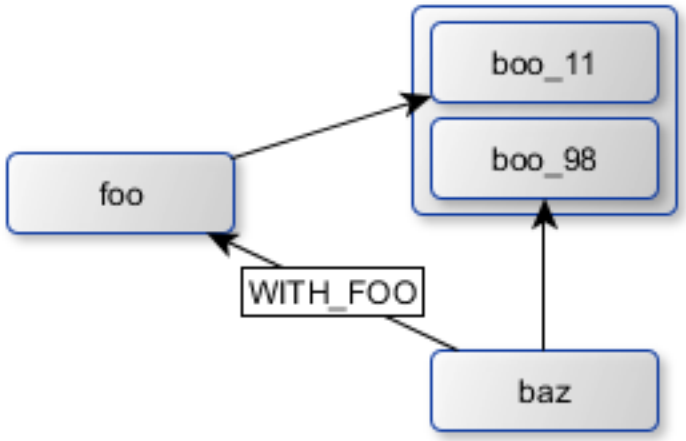
```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.1)
project(foo)

add_library(boo INTERFACE)
target_include_directories(boo INTERFACE "$<BUILD_INTERFACE:${CMAKE_CURRENT_LIST_DIR}>
↪")

add_executable(baz main.cpp)
target_link_libraries(baz PUBLIC boo)

if(WITH_FOO)
  add_library(foo Foo.cpp Foo.hpp)
  target_compile_features(foo PRIVATE cxx_constexpr)
  target_link_libraries(foo PUBLIC boo)
  target_link_libraries(baz PUBLIC foo)
  target_compile_definitions(baz PUBLIC WITH_FOO)
endif()
```

This is not correct and will end with error on link stage after successful generation and compilation:

```
[examples]> rm -rf _builds
[examples]> cmake -Htoolchain-usage-examples/globals/cxx-standard/bad -B_builds -
↪DWITH_FOO=ON
-- The C compiler identification is GNU 5.4.1
-- The CXX compiler identification is GNU 5.4.1
...
```

```
-- Configuring done
-- Generating done
-- Build files have been written to: /.../examples/_builds
[examples]> cmake --build _builds
Scanning dependencies of target foo
[ 25%] Building CXX object CMakeFiles/foo.dir/Foo.cpp.o
[ 50%] Linking CXX static library libfoo.a
[ 50%] Built target foo
Scanning dependencies of target baz
[ 75%] Building CXX object CMakeFiles/baz.dir/main.cpp.o
[100%] Linking CXX executable baz
CMakeFiles/baz.dir/main.cpp.o: In function `main':
main.cpp:(.text+0x64): undefined reference to `Foo::optimize(Boo::InternalThread&)'
collect2: error: ld returned 1 exit status
CMakeFiles/baz.dir/build.make:95: recipe for target 'baz' failed
make[2]: *** [baz] Error 1
CMakeFiles/Makefile2:104: recipe for target 'CMakeFiles/baz.dir/all' failed
make[1]: *** [CMakeFiles/baz.dir/all] Error 2
Makefile:83: recipe for target 'all' failed
make: *** [all] Error 2
```

The reason is violation of ODR rule, similar example have been described *before*. Effectively we are having two different libraries `boo_11` and `boo_98` with the same symbols:



Toolchain

Let's create toolchain file `cxx11.cmake` instead so we can use it to set standard globally for all targets in project:

```
# cxx11.cmake

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED YES)
```

You can add it with `-DCMAKE_TOOLCHAIN_FILE=/path/to/cxx11.cmake`:

```
[examples]> rm -rf _builds
[examples]> cmake -Htoolchain-usage-examples/globals/cxx-standard/toolchain -B_builds␣
↪-DCMAKE_TOOLCHAIN_FILE=cxx11.cmake -DWITH_FOO=YES
```

```
-- The C compiler identification is GNU 5.4.1
-- The CXX compiler identification is GNU 5.4.1
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- 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: /.../examples/_builds
[examples]> cmake --build _builds
Scanning dependencies of target foo
[ 25%] Building CXX object CMakeFiles/foo.dir/Foo.cpp.o
[ 50%] Linking CXX static library libfoo.a
[ 50%] Built target foo
Scanning dependencies of target baz
[ 75%] Building CXX object CMakeFiles/baz.dir/main.cpp.o
[100%] Linking CXX executable baz
[100%] Built target baz
[examples]> ./_builds/baz
C++ standard: 201103
With Foo support
```

Looks better now!

try_compile

The next thing we need to improve is early error detection. Note that now if we try to specify `WITH_FOO=ON` with C++98 there will be no errors reported on generation stage. Build will failed while trying to compile `foo` target.

To do this you can create C++ file and add few samples of features you are planning to use:

```cpp
// features_used_by_foo.cpp

constexpr int value() {
  return 0x42;
}

int main() {
  return value();
}
```

Use CMake module with `try_compile` to test this code:

```
# features_used_by_foo.cmake

set(bindir "${CMAKE_CURRENT_BINARY_DIR}/foo/try_compile")
set(saved_output "${bindir}/output.txt")
set(srcfile "${CMAKE_CURRENT_LIST_DIR}/features_used_by_foo.cpp")
try_compile(
```

```
    FOO_IS_FINE
    "${bindir}"
    "${srcfile}"
    OUTPUT_VARIABLE output
)
if(NOT FOO_IS_FINE)
  file(WRITE "${saved_output}" "${output}")
  message(
      FATAL_ERROR
      "Can't compile test file:\n"
      " ${srcfile}\n"
      "Error log:\n"
      " ${saved_output}\n"
      "Please check that your compiler supports C++11 features and C++11 standard␣
↪enabled."
  )
endif()
```

Include this check before creating target `foo`:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.1)
project(foo)

add_library(boo INTERFACE)
target_include_directories(boo INTERFACE "$<BUILD_INTERFACE:${CMAKE_CURRENT_LIST_DIR}>
↪")

add_executable(baz main.cpp)
target_link_libraries(baz PUBLIC boo)

if(WITH_FOO)
  include("${CMAKE_CURRENT_LIST_DIR}/features_used_by_foo.cmake")
  add_library(foo Foo.cpp Foo.hpp)
  target_link_libraries(foo PUBLIC boo)
  target_link_libraries(baz PUBLIC foo)
  target_compile_definitions(baz PUBLIC WITH_FOO)
endif()
```

Defaults

As usual cache variables allow us to set default values if needed:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.1)

set(
    CMAKE_TOOLCHAIN_FILE
    "${CMAKE_CURRENT_LIST_DIR}/cxx11.cmake"
    CACHE
    FILEPATH
    "Default toolchain"
)
```

```
project(foo)

option(WITH_FOO "Enable Foo optimization" ON)

add_library(boo INTERFACE)
target_include_directories(boo INTERFACE "$<BUILD_INTERFACE:${CMAKE_CURRENT_LIST_DIR}>
↪")

add_executable(baz main.cpp)
target_link_libraries(baz PUBLIC boo)

if(WITH_FOO)
  include("${CMAKE_CURRENT_LIST_DIR}/features_used_by_foo.cmake")
  add_library(foo Foo.cpp Foo.hpp)
  target_link_libraries(foo PUBLIC boo)
  target_link_libraries(baz PUBLIC foo)
  target_compile_definitions(baz PUBLIC WITH_FOO)
endif()
```

Note:

- Toolchain should be set before first `project` command, see *Project: Tools discovering*

See also:

- *Cache variables: Use case*

Scalability

If this example looks simple and used approach look like an overkill just imagine next situation:

- `boo` is external library that supports C++98, C++11, C++14, etc. standards and consists of 1000+ source files
- `foo` is external library that supports only few modern standards and tested with C++11 and C++17. Consist of 1000+ source files and non-trivially interacts with `boo`
- Your project `baz` has `boo` requirement and optional `foo`, should works correctly in all possibles variations

The worst that may happen if you will use toolchain approach is that `foo` will fail with **compile** error instead of error on generation stage. The error will be plain, such as `Can't use 'auto',-std=c++11 is missing?`. This can be easily improved with `try_compile`.

If you will keep using locally specified standard like modifying `CXX_STANDARD` property and conflict will occur:

- there will be **no warning** messages on generate step
- there will be **no warning** messages on compile step
- link will fail with opaque error pointing to some **implementation details** inside `boo` library while your usage of `boo` API will look completely fine

When you will try to find error elsewhere:

- stand-alone version of `boo` will work correctly with all examples and standards
- stand-alone version of `foo` will interact correctly with `boo` with all examples and supported standards
- your project `baz` will work correctly with `boo` if you will use configuration without `foo`

Summary

- Use toolchain if you need to specify standard, set default toolchain if needed
- Avoid using `CXX_STANDARD` in your code
- Avoid using `CMAKE_CXX_STANDARD` anywhere except toolchain
- Avoid using `target_compile_features` module
- If you have to use them for any reason at least protect it with `if`:

```
if(NOT EXISTS "${CMAKE_TOOLCHAIN_FILE}")
  set_target_properties(boo PROPERTIES CXX_STANDARD 14)
  target_compile_features(foo PUBLIC cxx_constexpr)
endif()
```

3.19 Generator expressions

3.20 Properties

3.21 Packing

3.22 Continuous integration

3.22.1 Travis

3.22.2 AppVeyor

CHAPTER 4

Platforms

4.1 iOS

4.1.1 Errors

Validate

Stackoverflow

- Missing iOS Distribution signing identity for <username>

Upload to App Store

Stackoverflow

- Getting ITMS-4238 "Redundant Binary Upload" (in terms of CMake you have to change CFBundleVersionString, e.g. 1.0 to 1.1)

4.1.2 Universal binaries

4.2 Android

4.2.1 General Hints

Prepare device

You have to prepare your device for debugging. For Android 4.2+ tap `Build number` seven times:

- *Settings* → *About phone* → *Build number*

`Developer options` appears now:

- *Settings* → *Developer options*

See also:

- Enabling On-device Developer Options

Note:

- On practice instructions may differ for different devices. E.g. it may be `Android version` or `MIUI version` instead of `Build number` (http://en.miui.com/thread-24025-1-1.html)

Go to `Developer options` and turn it `ON`:

- *Settings* → *Developer options* → *Developer options*

Also turn `ON` debug mode when USB is connected. Otherwise `adb` will not be able to discover the device:

- *Settings* → *Developer options* → *USB debugging*

Get Android NDK

Polly

- Script install-ci-dependencies.py will install Android NDK if environment variable `TOOLCHAIN` set to `android-*` (.travis.yml example).

Android NDK contains compilers and other tools for C++ development.

Get Android SDK

Hunter

- Android SDK will be downloaded automatically, no need to install it.

Android SDK tools used for development on Android platform: adb, android, emulator, etc.

Verify

Connect device with USB and verify it's visible by `adb` service:

```
> adb devices
List of devices attached
MTPxxx device
```

If service is not started there will be extra messages:

```
> adb devices
List of devices attached
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
MTPxxx device
```

SDK version on device

The needed version of SDK can be get by reading `ro.build.version.sdk`:

```
> adb -d shell getprop ro.build.version.sdk
19
```

Means you need to use API 19.

Note:

- `-d` is for real device
- `-e` is for emulator

CPU architecture

Run next command to determine CPU architecture of emulator:

```
> adb -e shell getprop ro.product.cpu.abi
x86
```

And this one for device:

```
> adb -d shell getprop ro.product.cpu.abi
armeabi-v7a
```

Log

See also:

- logcat

Clear log:

```
> adb logcat -c
```

Filter only Info (`I`) messages from `SimpleApp`, ignore others and exit:

```
> adb logcat -d SimpleApp:I '*:S'
--------- beginning of /dev/log/main
--------- beginning of /dev/log/system
I/SimpleApp( 9015): Hello from Android! (Not debug)
>
```

Any messages from `SimpleApp`, ignore others:

```
> adb logcat -d 'SimpleApp:*' '*:S'
--------- beginning of /dev/log/main
--------- beginning of /dev/log/system
I/SimpleApp( 9015): Hello from Android! (Not debug)
>
```

4.2.2 Windows Host

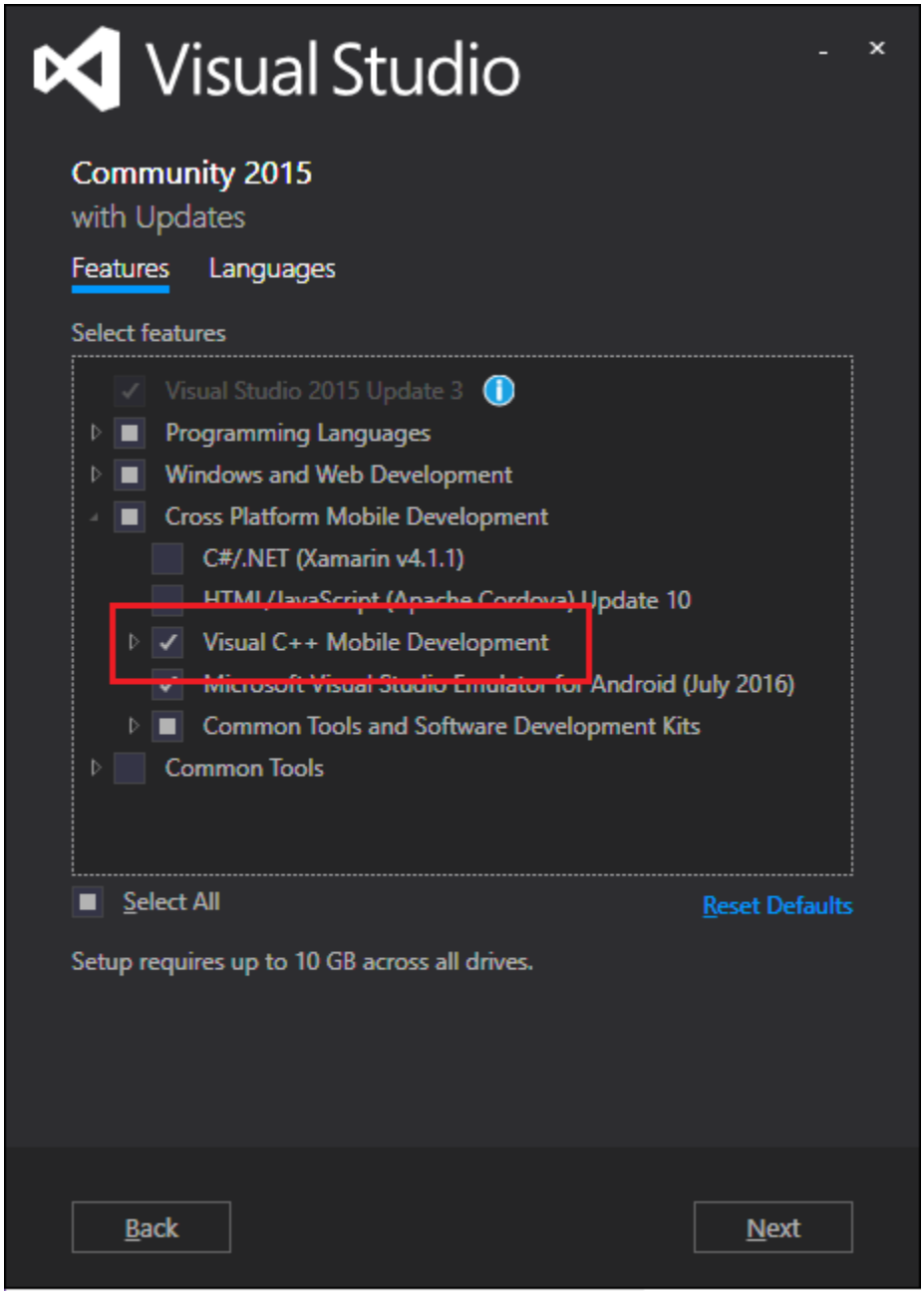
See also:

- Support for Android CMake projects in Visual Studio

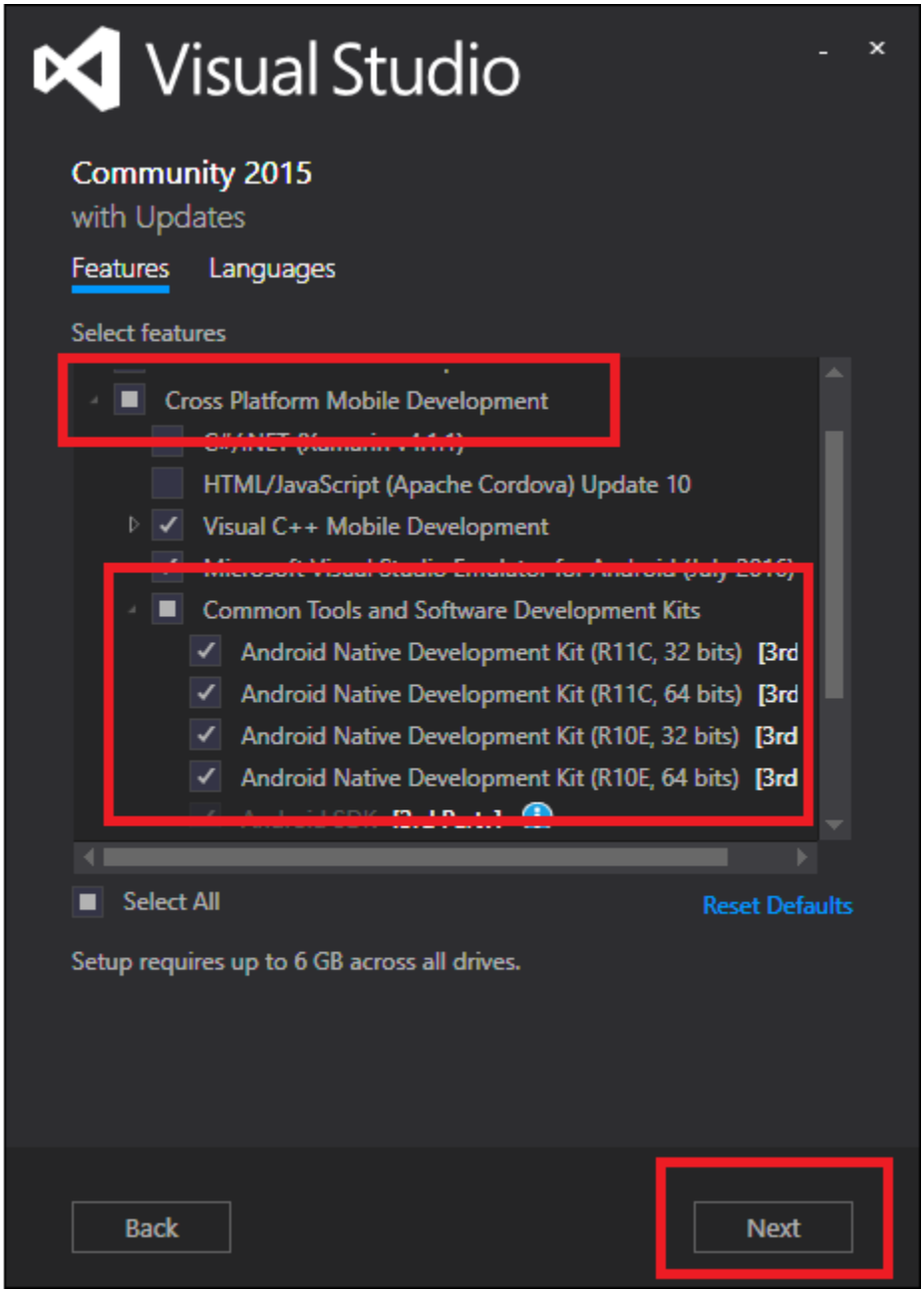
Required components

Install *additional components* first.

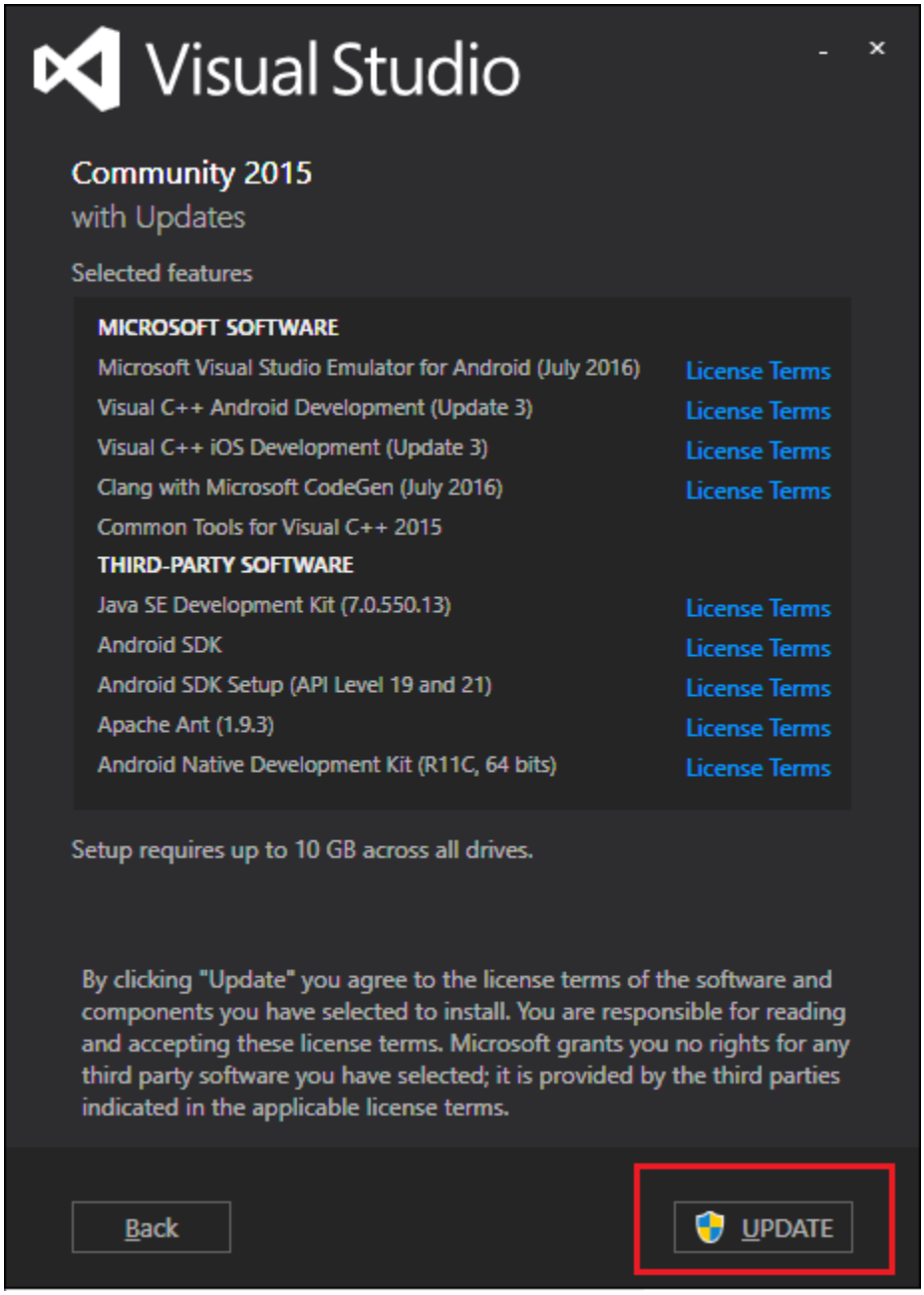
Cross Platform Mobile Development → *Visual C++ Mobile Development*:



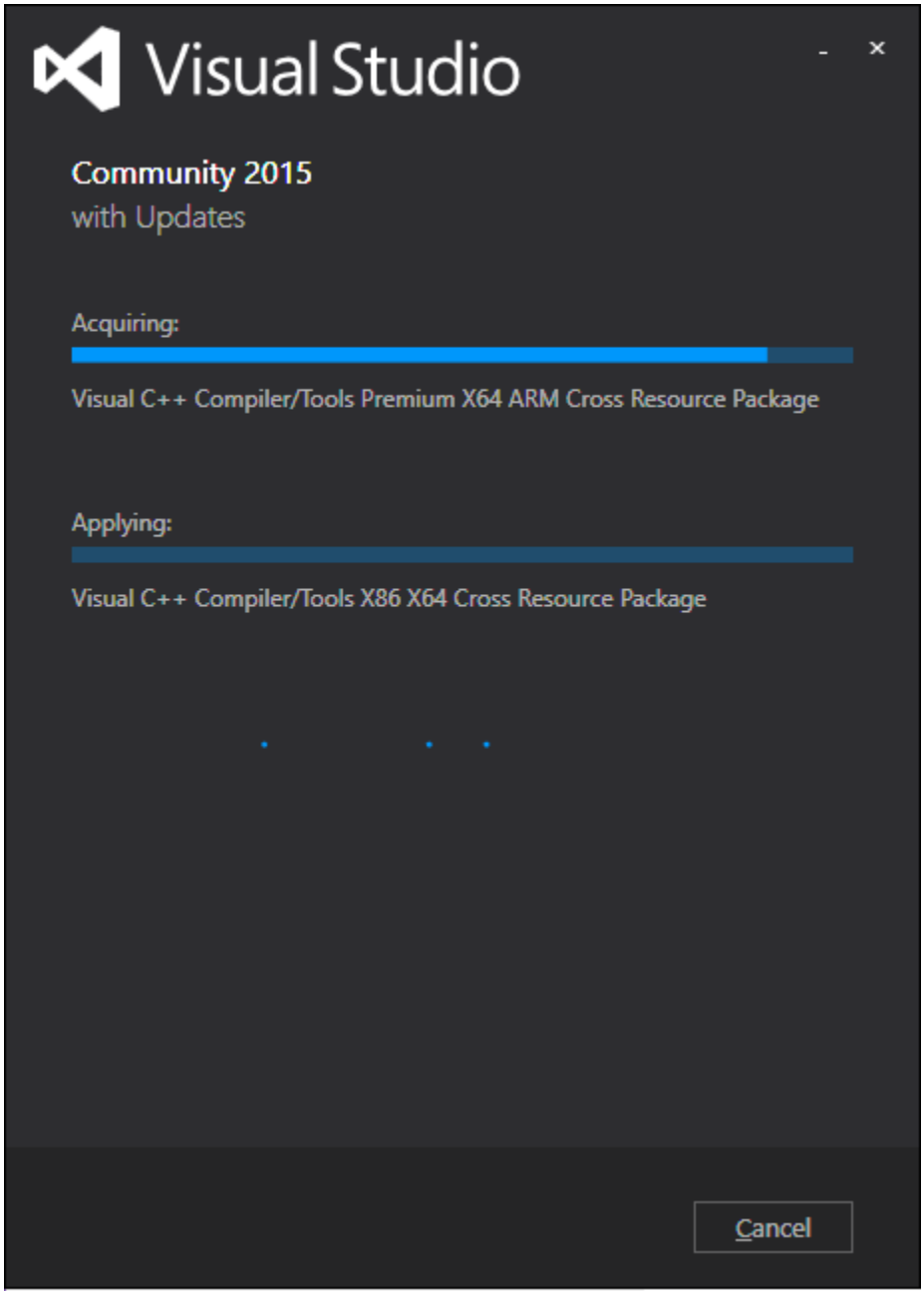
And *Cross Platform Mobile Development → Common Tools and Software Development Kits → Android Native Development Kit **:



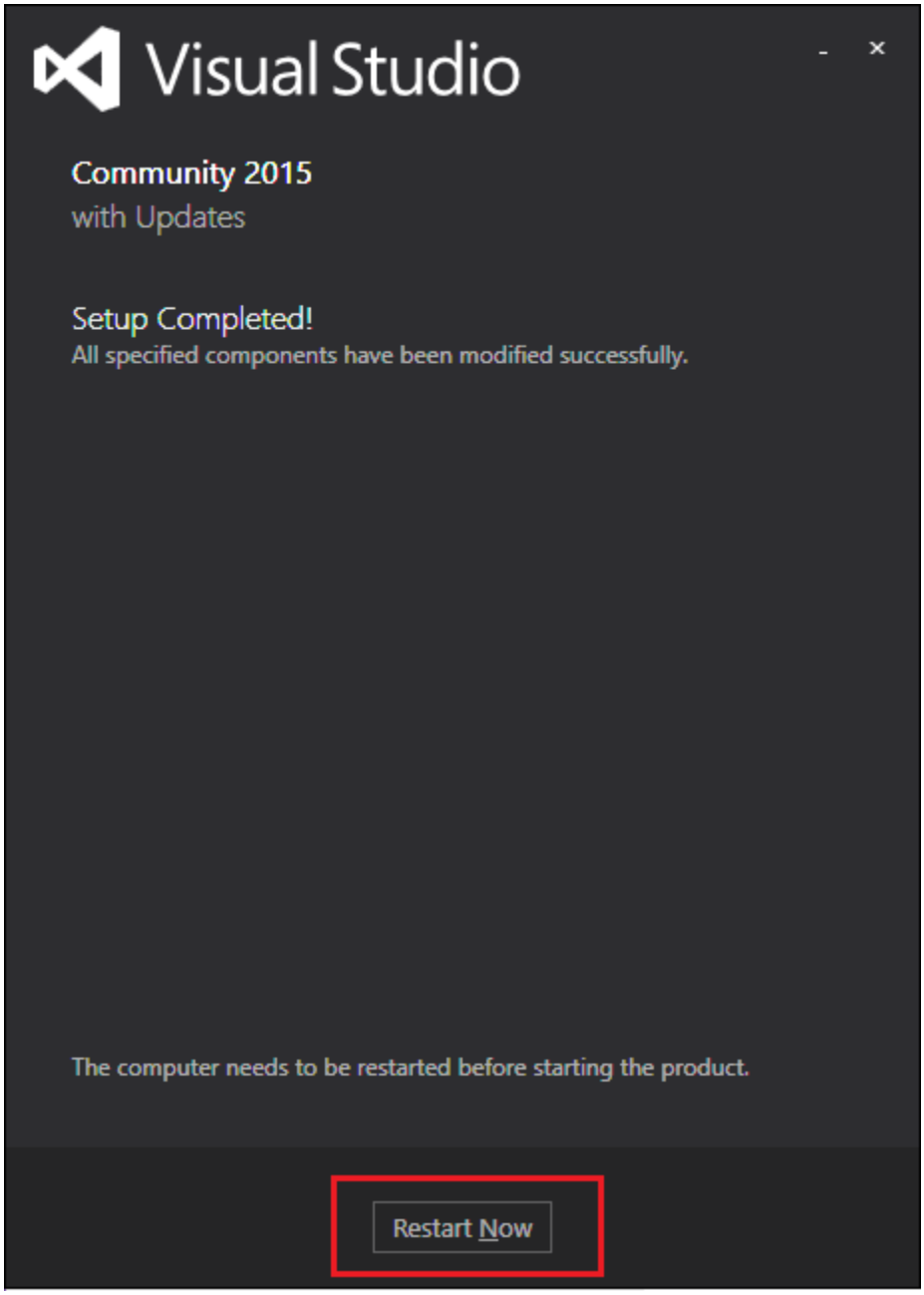
Click `Update`:



Wait until finished:



Restart may required:



Experimental CMake

Build experimental CMake version from `feature/VCMDDAndroid` branch of `Microsoft/CMake` repository: https://github.com/Microsoft/CMake/tree/feature/VCMDDAndroid.

You can download ZIP archive with binaries from here.

Unpack archive and add `bin` directory to PATH environment variable. Verify by `where cmake` that you're using experimental version.

Example

Examples on GitHub

- Repository
- Latest ZIP

```
cmake_minimum_required(VERSION 3.4)

### Emulate toolchain {
set(_expected_platform_module "${CMAKE_ROOT}/Modules/Platform/VCMDDAndroid.cmake")

if(NOT EXISTS "${_expected_platform_module}")
  message(
      FATAL_ERROR
      "File not found:\n  ${_expected_platform_module}\n"
      "You are using non-patched CMake version!"
  )
endif()

set(CMAKE_SYSTEM_NAME "VCMDDAndroid")
### }

project(foo)

add_library(foo foo.cpp)
```

Generate:

```
[platforms-android-on-windows]> cmake -H. -B_builds "-GVisual Studio 14 2015 ARM"
-- The C compiler identification is Clang 3.6.0
-- The CXX compiler identification is Clang 3.6.0
-- Check for working C compiler using: Visual Studio 14 2015 ARM
-- Check for working C compiler using: Visual Studio 14 2015 ARM -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler using: Visual Studio 14 2015 ARM
-- Check for working CXX compiler using: Visual Studio 14 2015 ARM -- 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: C:/.../platforms-android-on-windows/_builds
```

Build:

```
[platforms-android-on-windows]> cmake --build _builds --config Debug
Microsoft (R) Build Engine version 14.0.25420.1
Copyright (C) Microsoft Corporation. All rights reserved.

...

CustomBuild:
  Building Custom Rule C:/.../platforms-android-on-windows/CMakeLists.txt
```

```
  CMake does not need to re-run because C:\...\platforms-android-on-windows\_
↪builds\CMakeFiles\generate.stamp is up-to-date.
ClCompile:

  C:\ProgramData\Microsoft\AndroidNDK64\android-ndk-r10e\toolchains\llvm-3.
↪6\prebuilt\windows-x86_64\bin\clang.exe
  ...
  --sysroot="C:\ProgramData\Microsoft\AndroidNDK64\android-ndk-r10e\platforms\android-
↪19\arch-arm"
  -I "C:\ProgramData\Microsoft\AndroidNDK64\android-ndk-r10e\platforms\android-
↪19\arch-arm\usr\include"
  -I "C:\ProgramData\Microsoft\AndroidNDK64\android-ndk-r10e\sources\cxx-stl\gnu-
↪libstdc++\4.9\include"
  -I "C:\ProgramData\Microsoft\AndroidNDK64\android-ndk-r10e\sources\cxx-stl\gnu-
↪libstdc++\4.9\libs\armeabi-v7a\include"
  -I "C:\ProgramData\Microsoft\AndroidNDK64\android-ndk-r10e\sources\cxx-stl\gnu-
↪libstdc++\4.9\include\backward"
  -I "C:\ProgramData\Microsoft\AndroidNDK64\android-ndk-r10e\toolchains\llvm-3.
↪6\prebuilt\windows-x86_64\lib\clang\3.6\include"
  ...
  "C:\...\platforms-android-on-windows\foo.cpp"

foo.cpp
Lib:

...

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.04
```

Errors

Platform not found

If you see error like that:

```
[platforms-android-on-windows]> cmake -H. -B_builds "-GVisual Studio 14 2015 ARM"
CMake Error at CMakeLists.txt:7 (message):
  File not found:

    C:/.../Modules/Platform/VCMDDAndroid.cmake

  You are using non-patched CMake version!
```

It means you are not using experimental version. Check your `PATH` variable.

MDD not installed

If you haven't install *Cross Platform Mobile Development → Visual C++ Mobile Development* you will see this error:

```
[platforms-android-on-windows]> cmake -H. -B_builds "-GVisual Studio 14 2015 ARM"
CMake Error at CMakeLists.txt:17 (project):
  CMAKE_SYSTEM_NAME is 'VCMDDAndroid' but 'Visual C++ for Mobile Development
  (Android support)' is not installed.
```

Compiler not found

Check for *Cross Platform Mobile Development → Common Tools and Software Development Kits → Android Native Development Kit* * if you see this error:

```
[platforms-android-on-windows]> cmake -H. -B_builds "-GVisual Studio 14 2015 ARM"
-- The C compiler identification is unknown
-- The CXX compiler identification is unknown
CMake Error at CMakeLists.txt:17 (project):
  No CMAKE_C_COMPILER could be found.

CMake Error at CMakeLists.txt:17 (project):
  No CMAKE_CXX_COMPILER could be found.
```

Polly

- Android toolchains

Debugging with Visual Studio

Warning:

- Android with CMake support is in preliminary state so everything looks hacky.

Prepare

Install Polly if not already installed and add to PATH:

```
> git clone https://github.com/ruslo/polly
> set PATH=%cd%\polly\bin;%PATH%
> where polly.py
C:\...\polly\bin\polly.py
```

Check VC MDD version of CMake used (see notes above):

```
> where cmake
C:\...\cmake-3.4.2-win32-x86\bin\cmake.exe
```

Clone projects:

```
> git clone https://github.com/forexample/android-cmake
> cd android-cmake\09-vc-mdd-debug
[09-vc-mdd-debug]>
```

Verify that your device connected and visible, check CPU and API version fit toolchain that will be used (see *General Hints*).

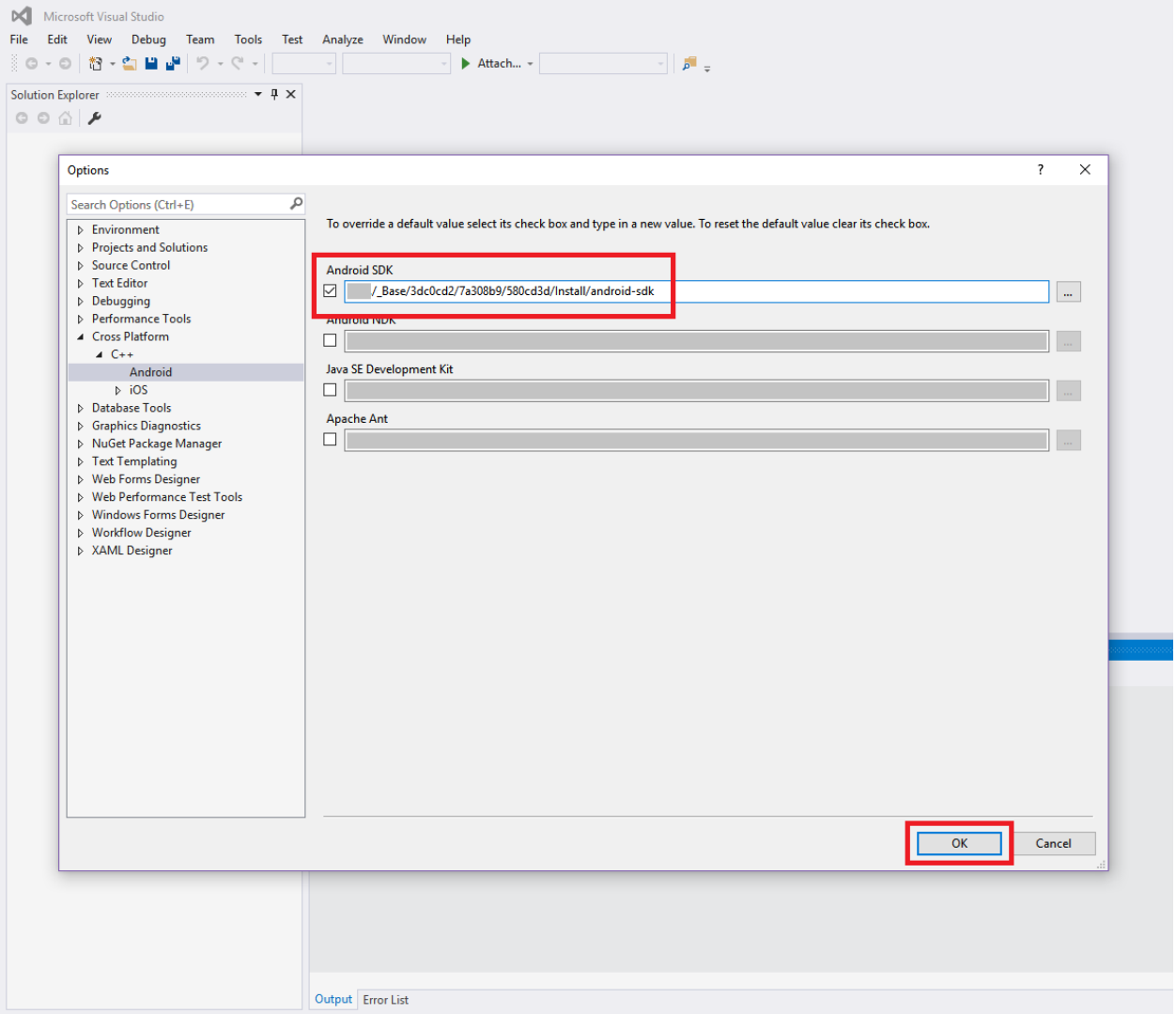
Android SDK

Now we will run CMake. This will trigger Hunter instructions that will download all dependencies. Do not run build for now and don't open Visual Studio project, we are interested in Android SDK path first:

```
[09-vc-mdd-debug]> build.py --toolchain android-vc-ndk-r10e-api-19-arm-clang-3-6 --
↪verbose --nobuild
...
-- [hunter *** DEBUG *** ...] Package already installed: Android-SDK
-- [hunter *** DEBUG *** ...] load: C:/.../cmake/projects/Android-SDK/hunter.cmake ...
↪ end
-- Path to `android`: C:/.../Install/android-sdk/tools/android
-- Path to `emulator`: C:/.../Install/android-sdk/tools/emulator
-- Path to `adb`: C:/.../Install/android-sdk/platform-tools/adb
-- Tools -> Options -> Cross Platform -> C++ -> Android: C:/.../Install/android-sdk
-- [hunter *** DEBUG *** ...] load: C:/.../cmake/projects/Android-Modules/hunter.cmake
-- [hunter *** DEBUG *** ...] Android-Modules versions available: [1.0.0]
```

Copy path `C:/.../Install/android-sdk` to clipboard and open Visual Studio. There is no way to control Android SDK path by CMake code so we have to add this path manually to

- *Tools* → *Options* → *Cross Platform* → *C++* → *Android*



Click OK and exit.

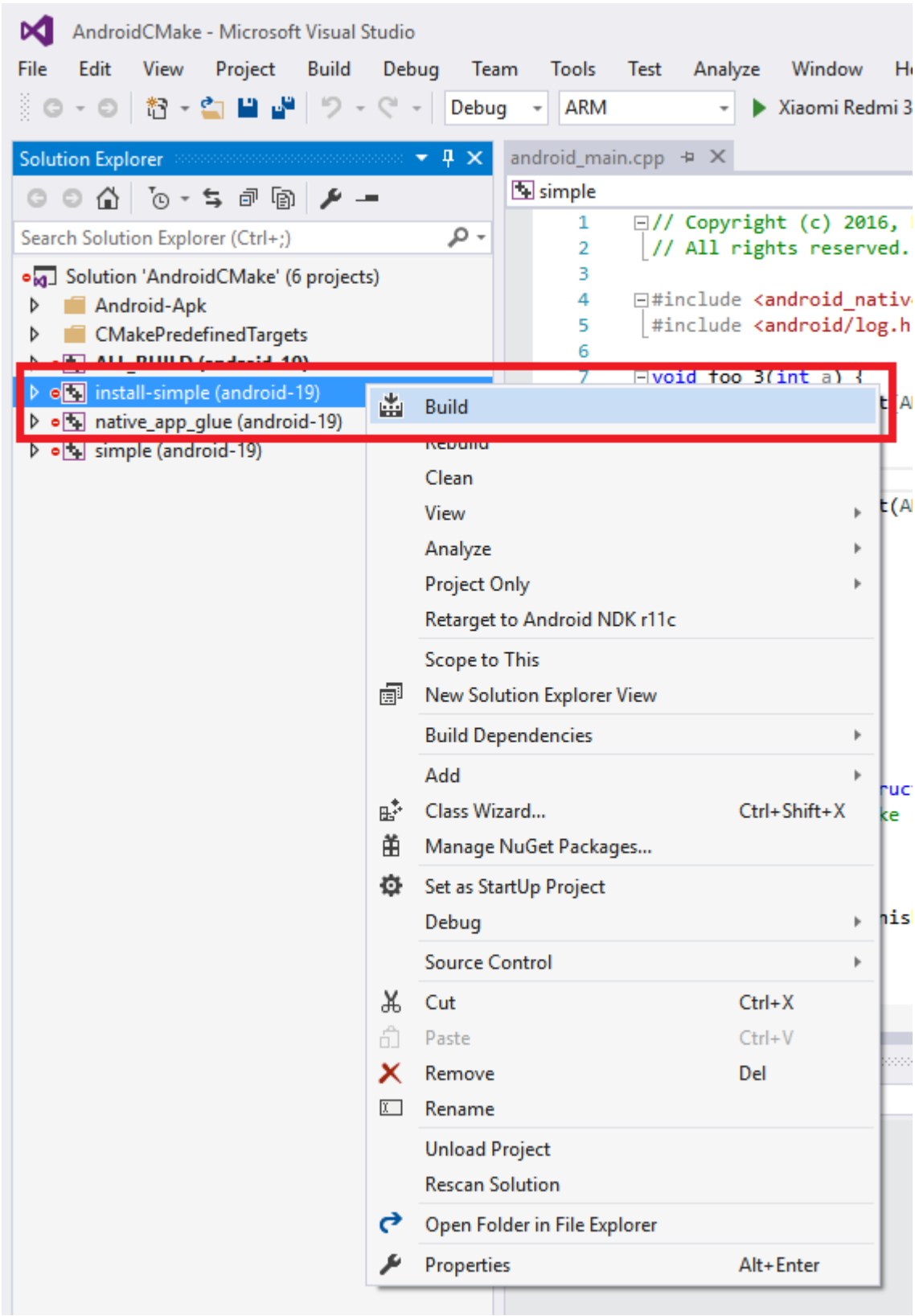
Note: It need to be done only once. Note that this setting is global so it will be used in other Visual Studio projects (they may not use Hunter or CMake).

APK Create

Run CMake again, this time use `--open` to open generated Visual Studio solution:

```
[09-vc-mdd-debug]> build.py --toolchain android-vc-ndk-r10e-api-19-arm-clang-3-6 --
↪verbose --nobuild --open
```

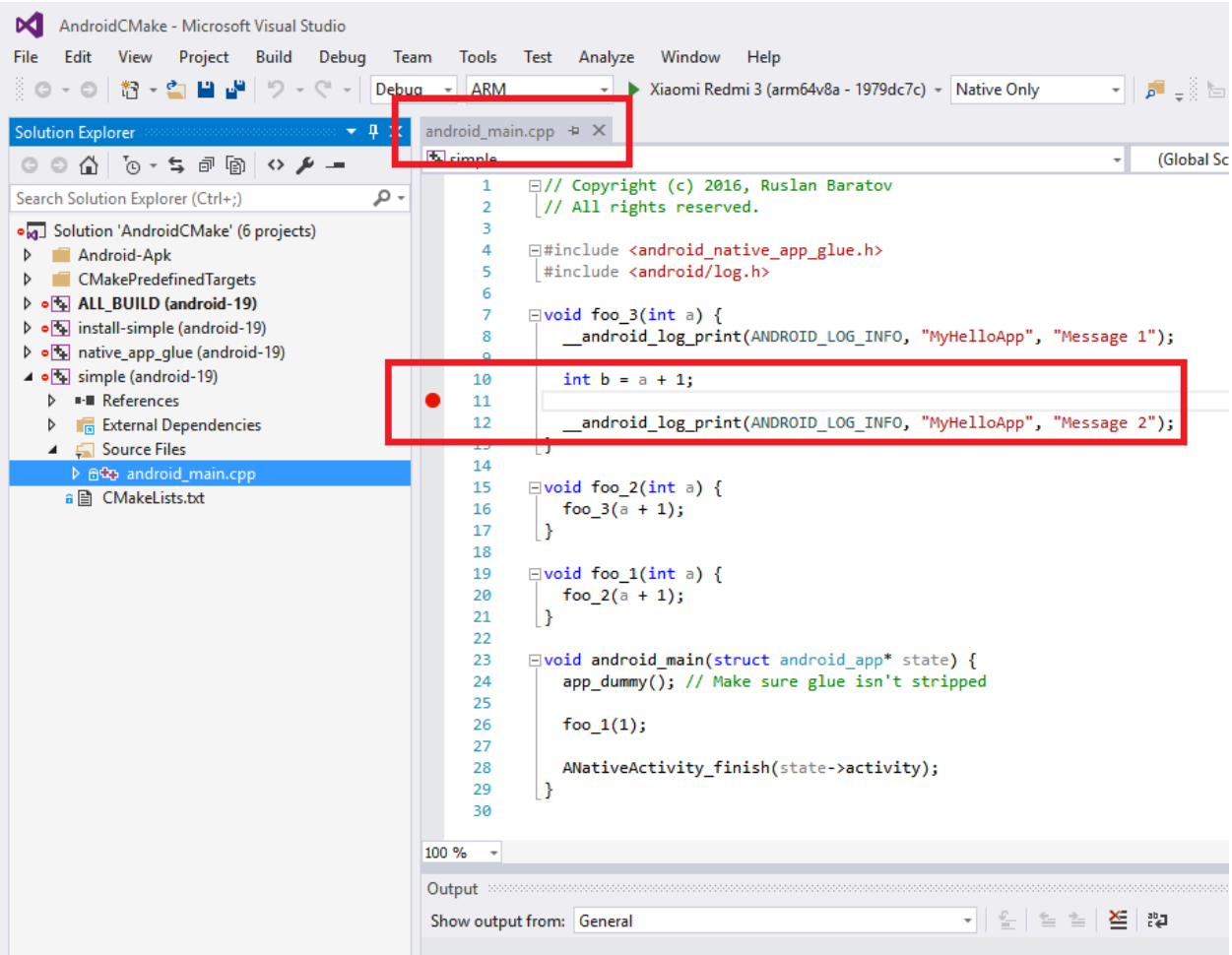
Find `install-simple` target and build it. This will trigger creating APK and installing it on device:


AndroidCMake - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window
Debug
ARM
Xiaomi Redmi 3
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'AndroidCMake' (6 projects)
Android-Apk
CMakePredefinedTargets
install-simple (android-19)
native_app_glue (android-19)
simple (android-19)
android_main.cpp
simple
1 // Copyright (c) 2016,
2 // All rights reserved.
3
4 #include <android_nativ
5 #include <android/log.h
6
7 void foo 3(int a) {
Build
Clean
View
Analyze
Project Only
Retarget to Android NDK r11c
Scope to This
New Solution Explorer View
Build Dependencies
Add
Class Wizard... Ctrl+Shift+X
Manage NuGet Packages...
Set as StartUp Project
Debug
Source Control
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename
Unload Project
Rescan Solution
Open Folder in File Explorer
Properties Alt+Enter

Warning: This step is necessary **always** before running application on device, otherwise old APK will be used and new updates will not be visible!

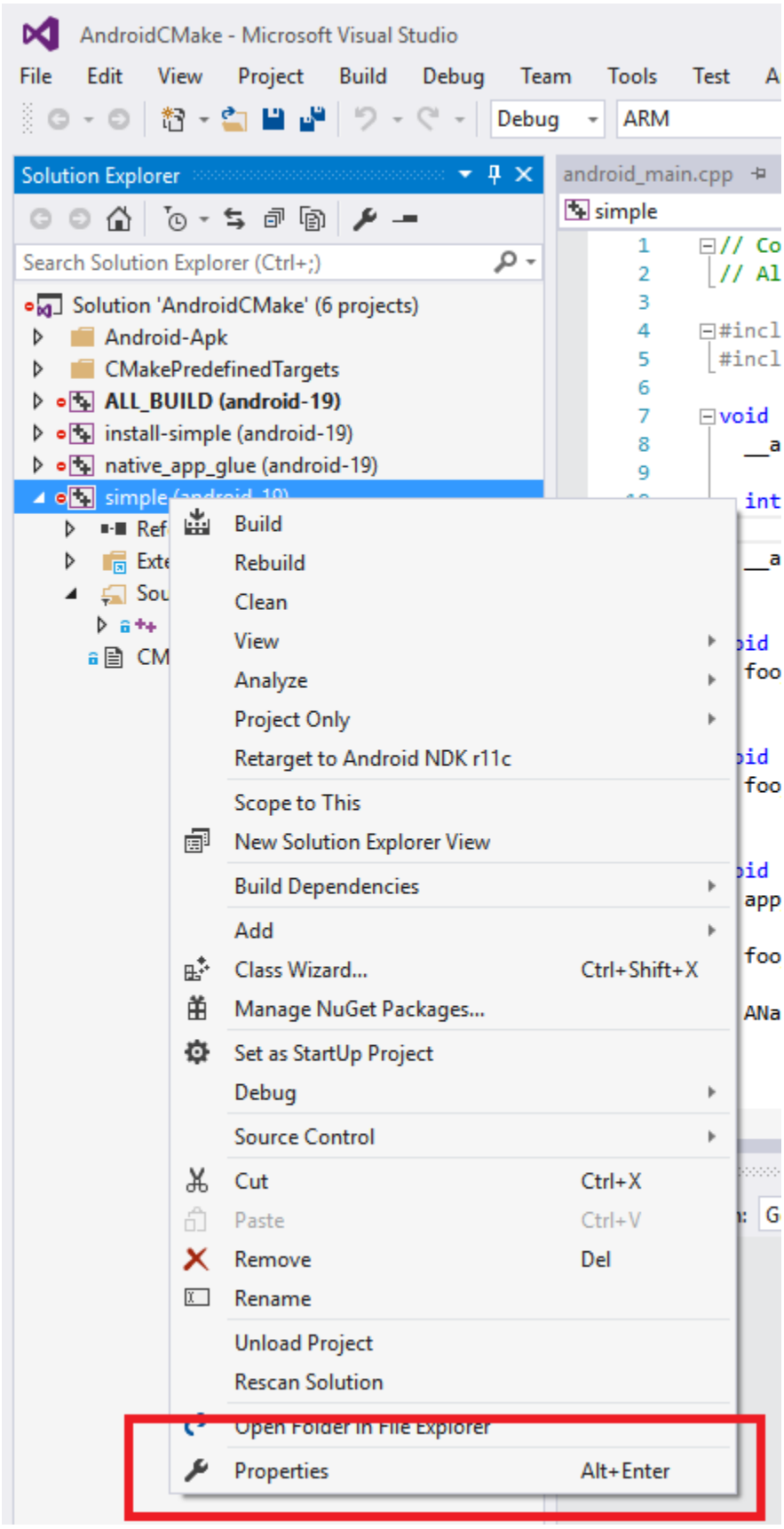
Breakpoints

Let's find `android_main.cpp` source file, open it and set breakpoint to the function `foo_3` before printing `Message 2`:

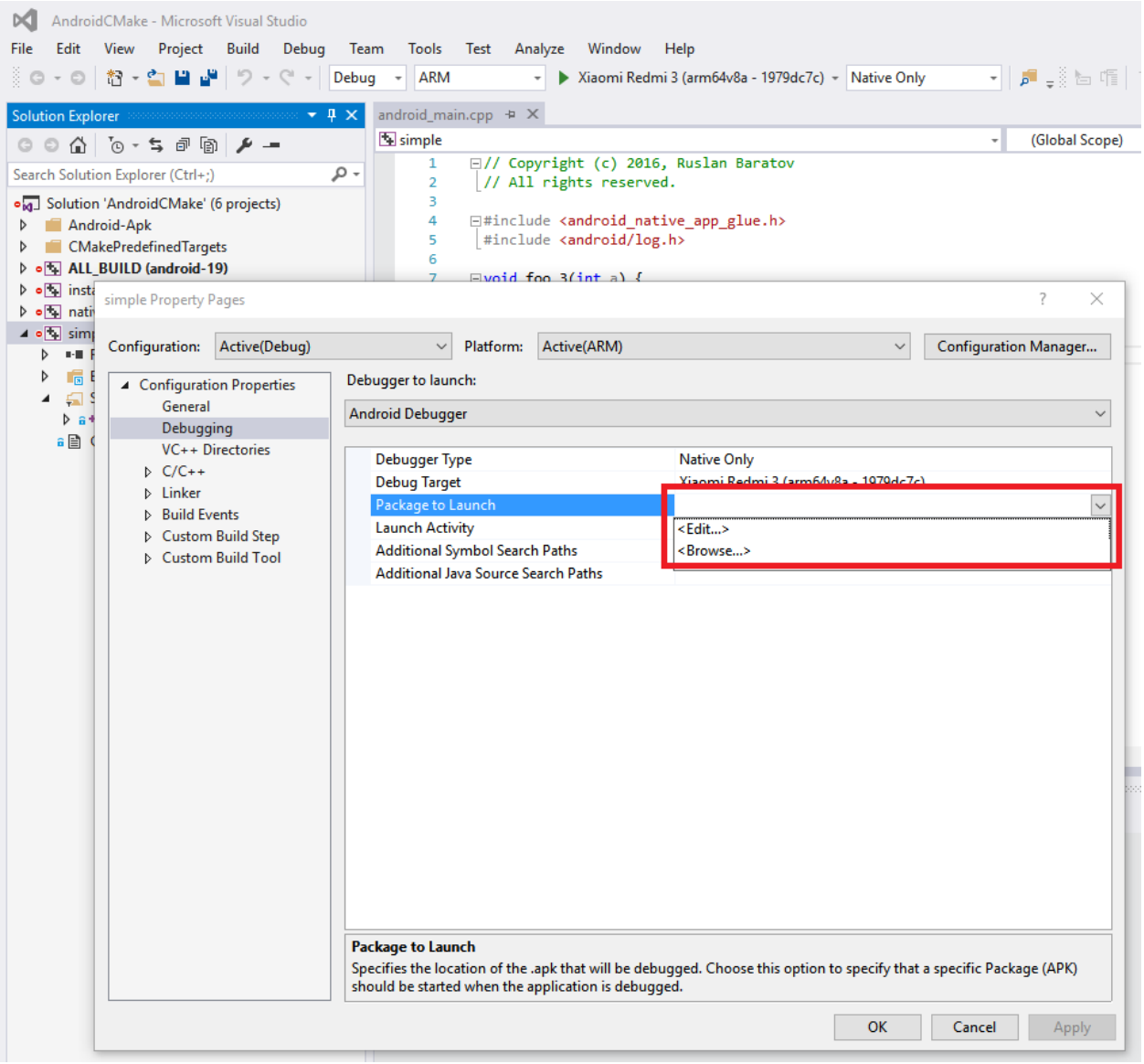


APK path

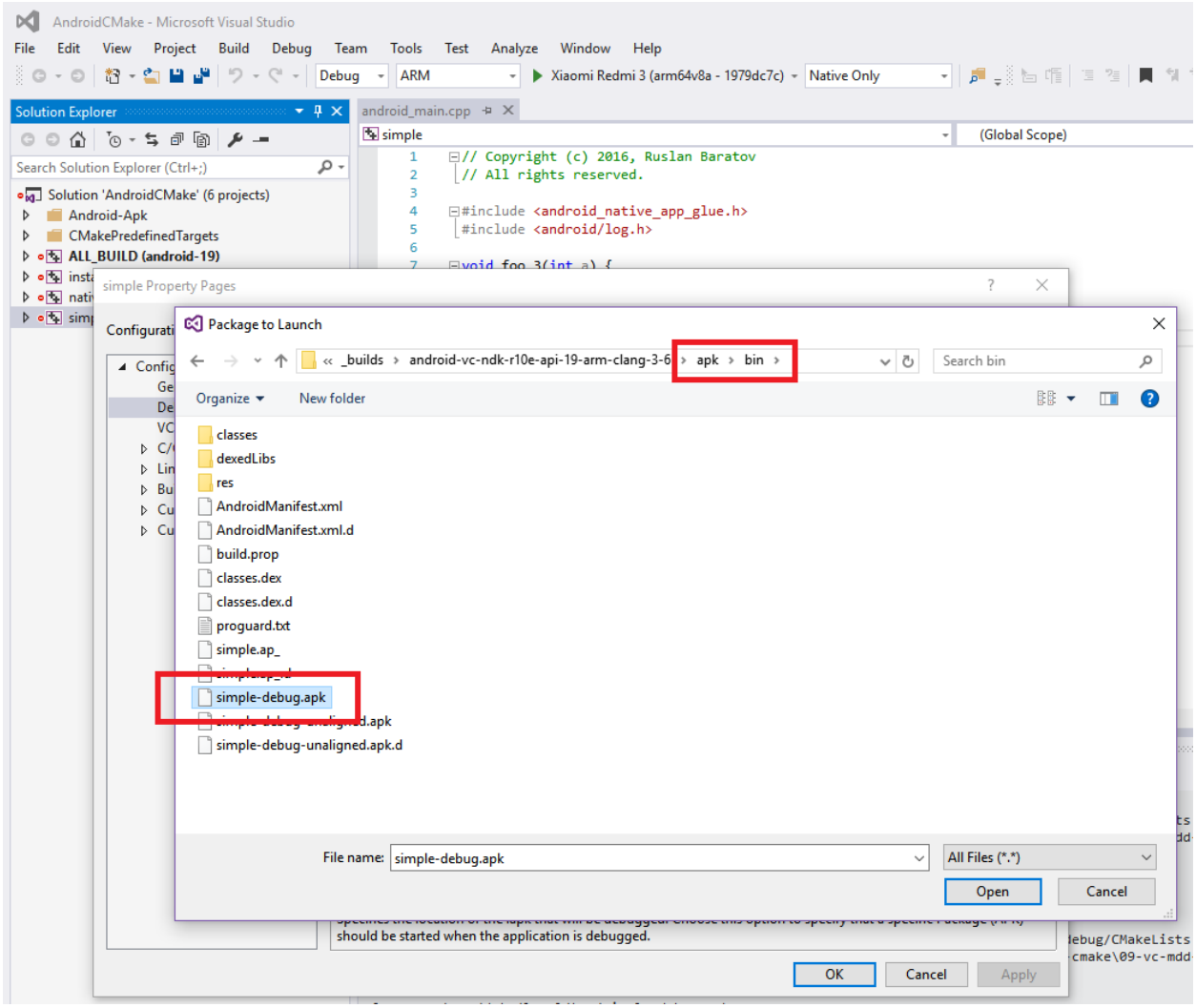
We have to tell Visual Studio which APK file we are debugging. Go to the `Properties`:


AndroidCMake - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test A
Debug
ARM
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'AndroidCMake' (6 projects)
Android-Apk
CMakePredefinedTargets
ALL_BUILD (android-19)
install-simple (android-19)
native_app_glue (android-19)
android_main.cpp
simple
Build
Rebuild
Clean
View
Analyze
Project Only
Retarget to Android NDK r11c
Scope to This
New Solution Explorer View
Build Dependencies
Add
Class Wizard... Ctrl+Shift+X
Manage NuGet Packages...
Set as StartUp Project
Debug
Source Control
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename
Unload Project
Rescan Solution
Open Folder in File Explorer
Properties Alt+Enter

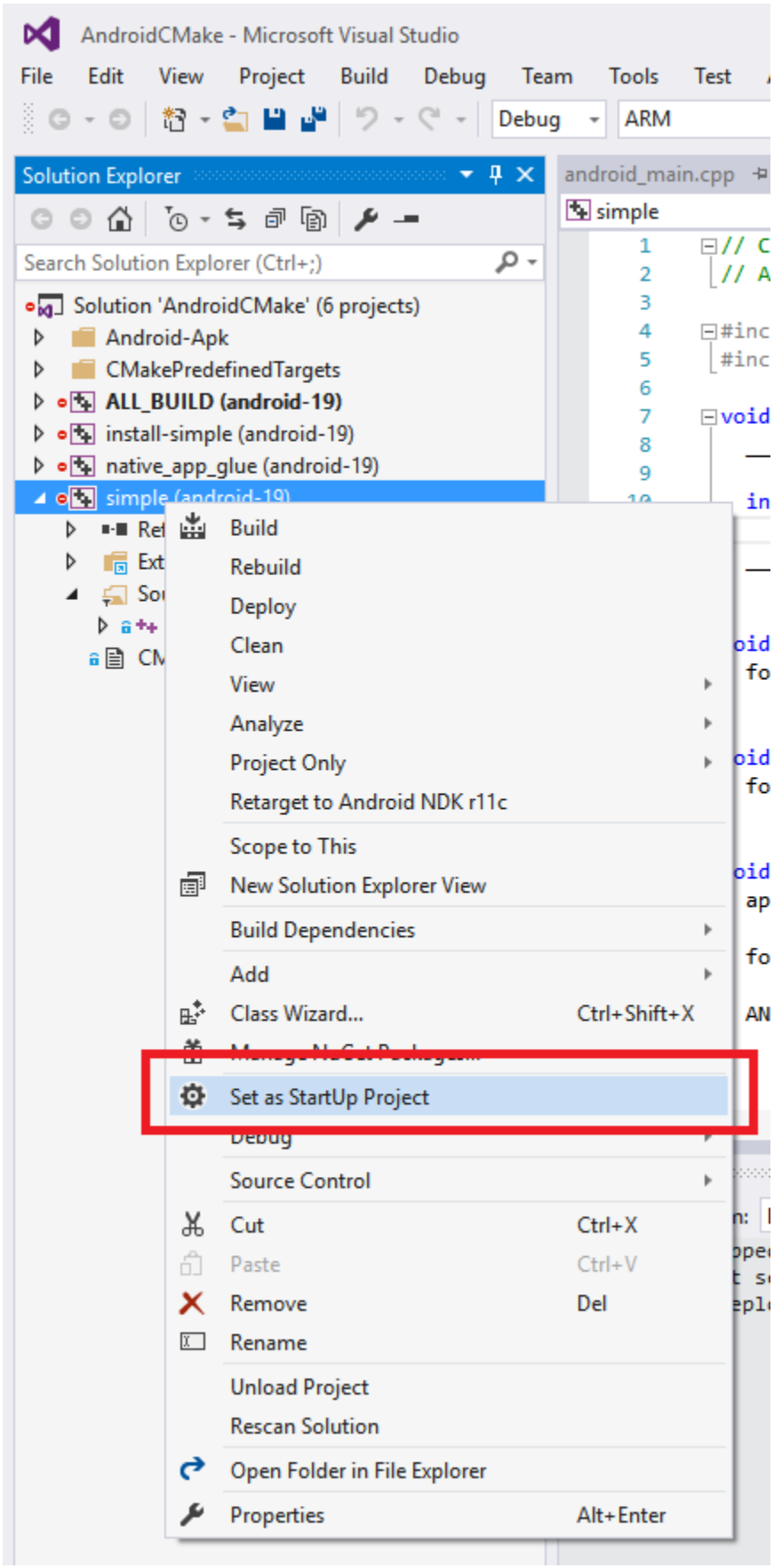
Find `Package to Launch` in `Debugging`:



And find `simple-debug.apk` in `apk/bin` folder:

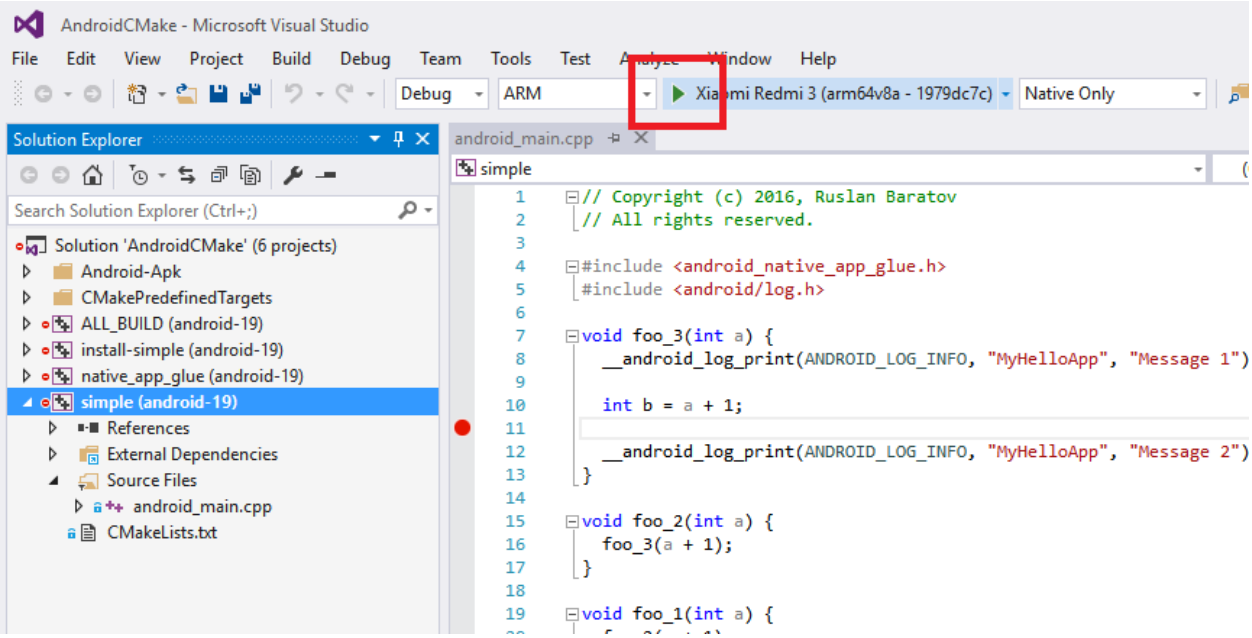


Now set this project as StartUp:

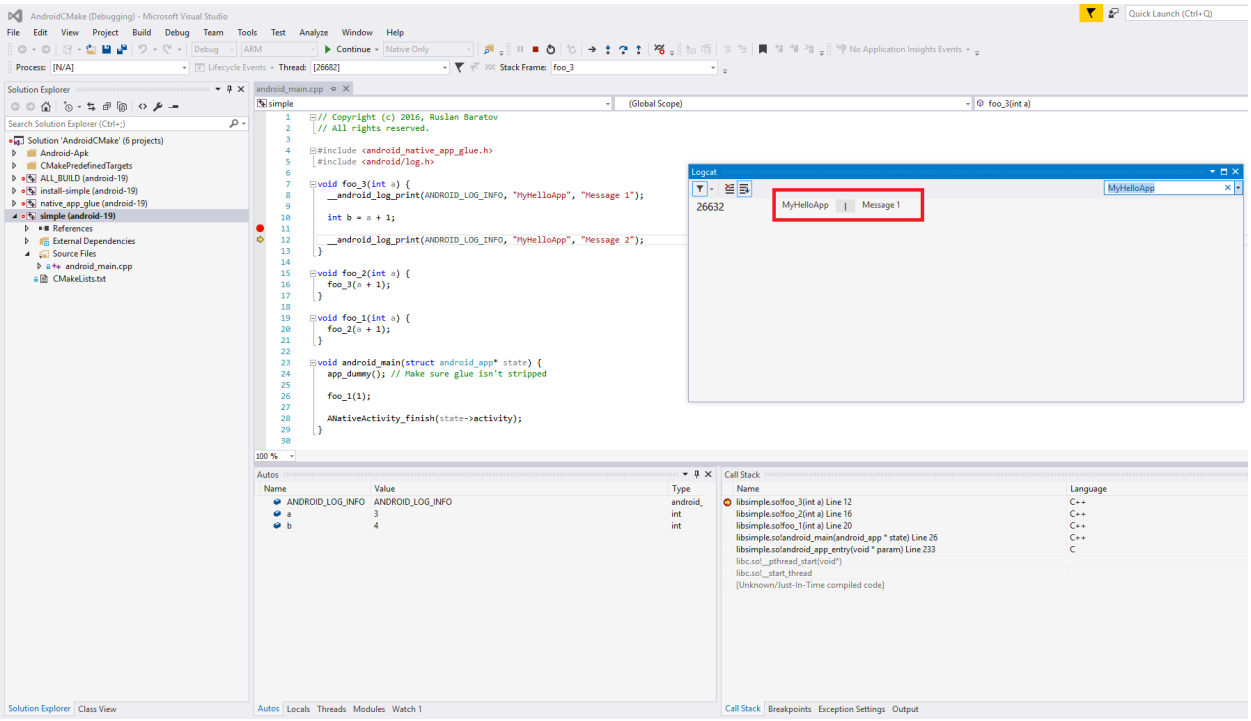

AndroidCMake - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test
Debug
ARM
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'AndroidCMake' (6 projects)
Android-Apk
CMakePredefinedTargets
ALL_BUILD (android-19)
install-simple (android-19)
native_app_glue (android-19)
simple (android-19)
android_main.cpp
simple
Build
Rebuild
Deploy
Clean
View
Analyze
Project Only
Retarget to Android NDK r11c
Scope to This
New Solution Explorer View
Build Dependencies
Add
Class Wizard... Ctrl+Shift+X
Set as StartUp Project
Debug
Source Control
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename
Unload Project
Rescan Solution
Open Folder in File Explorer
Properties Alt+Enter

Run

We are ready to run!



Debugger stops right after `Message 1` and before `Message 2`. Screenshot with call stack, logcat and values of local variables `a` and `b`:



CHAPTER 5

Generators

5.1 Ninja

- CMake option: `-G Ninja`
- Website
- Sources on GitHub

CMake documentation

- Ninja

5.1.1 Installation

Ubuntu

```
> sudo apt-get install ninja-build
> ninja -h
usage: ninja [options] [targets...]
...
> ninja --version
1.3.4
```

CHAPTER 6

Compilers

CHAPTER 7

Contacts

7.1 Public

- Feel free to open a new issue if you want to ask a question

7.2 Private

- Write me at ruslan_baratov@yahoo.com
- Private chat room on Gitter: https://gitter.im/ruslo

7.3 Hire

I'm available for hire as a freelance developer for all types of CGold development (adding details to existing articles, adding new tutorials/examples, etc.) or CMake development (introduce configuration from scratch, refactor existing code, etc.).

7.4 Donations

If you like CGold and its goals, consider supporting it by making a donation. Thanks! :)

CHAPTER 8

Rejected

There are topics that will be intentionally not covered by this document. Some features are obsolete - there are better clean and modern approaches. Other features lead to error-prone code and should not be used. Also I want to keep document straight/focused and avoid creating too broad tutorial.

8.1 ExternalProject_Add

There will be no hints about writing super-build project using ExternalProject because same can be done nicely with *Hunter package manager*.

CMake documentation

- ExternalProject

See also:

- Hunter features: do not repeat yourself

8.2 FindXXX.cmake

There are no instructions for writing `FindXXX.cmake` files like FindZLIB.cmake because it's easier to add some code to generate ZLIBConfig.cmake automatically.

Quote from CMake wiki:

```
If creating a Find* module for a library that already uses CMake as its build
system, please create a *Config.cmake instead, and submit it upstream. This
solution is much more robust.
```

CMake documentation

- Creating packages

Examples on GitHub

- Package example

8.3 macro

Unlike `function` command the `macro` command doesn't create scope so it does *modify variables* from scope where it called.

Note:

- Use *function* instead

CMake documentation

- macro

8.4 Object libraries

CMake documentation

- Object Libraries
- add_library(... OBJECT ...)

As documentation states `OBJECT` library is a non-archival collection of object files. `OBJECT` libraries have few limitations which makes them harder to use.

8.4.1 target_link_libraries

`OBJECT` library can't be used on the right hand side of `target_link_libraries` command. In practice it means that you will not be able to make a hierarchy of targets as you do with regular `add_library` command.

Example:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.2)
project(foo)

add_library(boo OBJECT boo.cpp)

add_library(foo OBJECT foo.cpp)
target_link_libraries(foo PUBLIC boo)

add_executable(baz $<TARGET_OBJECTS:foo> baz.cpp)
```

Will produce an error:

```
CMake Error at CMakeLists.txt:8 (target_link_libraries):
  Object library target "foo" may not link to anything.
```

You should put all dependent components to `add_executable` explicitly:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.2)
project(foo)

add_library(boo OBJECT boo.cpp)

add_library(foo OBJECT foo.cpp)

add_executable(
    baz
    $<TARGET_OBJECTS:foo>
    # List all 'foo' dependencies explicitly
    $<TARGET_OBJECTS:boo>
    # ...
    baz.cpp
)
```

If this component is optional:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.2)
project(foo)

option(FOO_WITH_BOO "With 'boo' component" ON)

if(FOO_WITH_BOO)
  add_library(boo OBJECT boo.cpp)
  set(boo_objects $<TARGET_OBJECTS:boo>)
else()
  set(boo_objects "")
endif()

add_library(foo OBJECT foo.cpp)

add_executable(
    baz
    $<TARGET_OBJECTS:foo>
    ${boo_objects}
    baz.cpp
)
```

8.4.2 Target name

Even if an `OBJECT` library is not a "real" target you will still have to name it carefully as a regular target since it will occupy slot in pool of names. As a result you can't use it as a local temporary helper tool:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.2)
project(foo)

add_subdirectory(boo)
add_subdirectory(bar)
```

```
# boo/CMakeLists.txt

add_library(core OBJECT x1.cpp x2.cpp)
add_executable(boo $<TARGET_OBJECTS:core> boo.cpp)
```

```
# bar/CMakeLists.txt

add_library(core OBJECT y1.cpp y2.cpp)
add_executable(bar $<TARGET_OBJECTS:core> bar.cpp)
```

Error:

```
CMake Error at bar/CMakeLists.txt:1 (add_library):
  add_library cannot create target "core" because another target with the
  same name already exists.  The existing target is created in source
  directory "/.../boo".  See documentation
  for policy CMP0002 for more details.
```

8.4.3 Usage requirements

Usage requirements will not be propagated:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.2)
project(foo)

include_directories("${CMAKE_CURRENT_LIST_DIR}")

add_library(boo OBJECT boo.cpp boo.hpp)
target_compile_definitions(boo PUBLIC FOO_WITH_BOO)

add_executable(baz $<TARGET_OBJECTS:boo> baz.cpp)
```

```
// boo.hpp

#ifndef BOO_HPP_
#define BOO_HPP_

#if !defined(FOO_WITH_BOO)
# error "FOO_WITH_BOO is not defined!"
#endif

#endif // BOO_HPP_
```

```
// baz.cpp

#include <boo.hpp>

int main() {
}
```

`boo.cpp` source will compile fine because `FOO_WITH_BOO` will be added:

```
/usr/bin/g++ -DFOO_WITH_BOO ... -o CMakeFiles/boo.dir/boo.cpp.o -c /.../boo.cpp
```

But not `baz.cpp`:

```
/usr/bin/g++ ... -o CMakeFiles/baz.dir/baz.cpp.o -c /.../baz.cpp
In file included from /.../baz.cpp:3:0:
/.../boo.hpp:7:3: error: #error "FOO_WITH_BOO is not defined!"
 # error "FOO_WITH_BOO is not defined!"
   ^
```

8.4.4 No real sources

As mentioned in documentation you can't have target with only `OBJECT` files. E.g. this code will not work with Xcode:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.2)
project(foo)

add_library(boo OBJECT boo.cpp)
add_executable(foo $<TARGET_OBJECTS:boo>)

enable_testing()
add_test(NAME foo COMMAND foo)
```

No warnings or build errors but when you will try to test it:

```
1: Test command:
Unable to find executable: /.../_builds/Release/foo
1/1 Test #1: foo ..............................***Not Run   0.00 sec
```

Note: As a workaround you can add dummy source file to the executable.

8.4.5 Name conflict

You can't have two source files with the same names even if they are located in different directories. This code will not work with Xcode generator:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.2)
project(foo)

add_library(boo OBJECT x.cpp boo/x.cpp)
add_executable(foo foo.cpp $<TARGET_OBJECTS:boo>)
```

8.5 target_compile_features

CMake documentation

- CMake compile features
- target_compile_features

This function sets locally something that belongs to global settings. Such behavior can lead to nontrivial errors. See for details:

- *ODR violation (global)*
- *C++ standard*

8.6 write_compiler_detection_header

CMake documentation

- WriteCompilerDetectionHeader

This module doesn't provide enough abstraction. You have to specify supported compilers explicitly. From documentation:

```
Compilers which are known to CMake, but not specified are detected and a
preprocessor #error is generated for them.
```

Meaning that this code:

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.10)
project(foo)

include(WriteCompilerDetectionHeader)

set(gen_include "${CMAKE_CURRENT_BINARY_DIR}/generated/")
write_compiler_detection_header(
    FILE "${gen_include}/${PROJECT_NAME}/detection.hpp"
    PREFIX ${PROJECT_NAME}
    COMPILERS Clang MSVC
    FEATURES cxx_variadic_templates
    VERSION 3.10
)

add_executable(foo foo.cpp)
target_include_directories(
    foo PUBLIC "$<BUILD_INTERFACE:${gen_include}>"
)
```

```
// foo.cpp
#include <foo/detection.hpp>

int main() {
}
```

Will return error while compiling with GCC:

```
/usr/bin/g++ ... -c /.../foo.cpp
In file included from /.../foo.cpp:2:0:
/.../generated/foo/detection.hpp:192:6: error: #error Unsupported compiler
 #    error Unsupported compiler
      ^
```

Compiler identification relies on `CMAKE_<LANG>_COMPILER_ID` which is not guaranteed to be set by CMake. From documentation:

```
This variable is not guaranteed to be defined for all compilers or languages.
```

CHAPTER 9

Glossary

9.1 -B

Add `-B<path-to-binary-tree>` to set the path to directory where CMake will store generated files. There must be no spaces between `-B` and `<path-to-binary-tree>`. Always must be used with *-H* option.

Path to this directory will be saved in CMAKE_BINARY_DIR variable.

Note: Starting with CMake 3.13, `-B` is an officially supported flag, can handle spaces correctly and can be used independently of the *-S* or *-H* options.

```
cmake -B _builds .
```

See also:

- *Binary tree*
- *-S*
- *-H*

9.2 -H

Note: Has been replaced in 3.13 with the official source directory flag of *-S*.

Add `-H<path-to-source-tree>` to set directory with `CMakeLists.txt`. This internal option is not documented but widely used by community. There must be no spaces between `-H` and `<path-to-source-tree>` (otherwise option will be interpreted as synonym to `--help`). Always must be used with *-B* option. Example:

```
cmake -H. -B_builds
```

Use current directory as a source tree (i.e. start with `./CMakeLists.txt`) and put generated files to the `./_builds` folder.

Path to this directory will be saved in CMAKE_SOURCE_DIR variable.

Warning: PowerShell will modify arguments and put the space between `-H` and `..` You can protect argument by quoting it:

```
cmake '-H.' -B_builds
```

See also:

- *-S*
- *-B*
- *Source tree*

CMake mailing list

- Document -H/-B

9.3 -S

Add `-S <path-to-source-tree>` to set directory with `CMakeLists.txt`. This option was added in CMake 3.13 and replaces the the undocumented and internal variable `-H`. This option can be used independently of `-B`.

```
cmake -S . -B _builds
```

Use current directory as a source tree (i.e. start with `./CMakeLists.txt`) and put generated files to the `./_builds` folder.

Path to this directory will be saved in CMAKE_SOURCE_DIR variable.

See also:

- *-B*
- *Source tree*

9.4 CMake

CMake is a cross-platform build system generator. Well this document entirely about CMake :)

CMake documentation

- CMake

Wikipedia

- CMake

9.5 Git

As a man page states Git is the stupid content tracker originally started by Linus Torvalds. At the time of the writing this, Git used to manage current documents and most of the projects related to CGold. In all cases when *VCS* functionality mentioned to show the practical example Git is used but similar cases can be applied to other *VCS's* as well.

See also:

- Official site

Wikipedia

- Git

9.6 Hunter

Hunter is a cross-platform package manager driven by CMake. Hunter works on Linux, OS X, Windows, iOS, Android, Raspberry Pi platforms in CMake friendly fashion. Actually Hunter is the reason why CGold exists since to be easily integrated into Hunter project should (but not must) be nicely and correctly written. On practice the number of such projects is very low (very-very low).

See also:

- Official documentation
- Simple example with GTest

9.7 Native build tool

Native build tool (also known as `native tool` or `native build system`) is the real tool (collection of tools such as compiler+IDE) used to build your software. *CMake* is not a build tool itself since it can't build your projects or help with development like IDE do. CMake responsibility is to **generate** native build tool files from abstracted configuration code.

Examples of native build tools:

- Xcode
- Visual Studio
- Ninja
- Make

9.7.1 Quotes

Quote from CMAKE_OBJECT_PATH_MAX:

```
Maximum object file full-path length allowed by native build tools
```

Quote from CMake:

```
Users build a project by using CMake to generate a build system for a native
tool on their platform
```

Quote from CMake options:

```
CMake may support multiple native build systems on certain platforms
```

9.8 VCS

Version control system. Quote from wikipedia:

```
A component of software configuration management, version control, also known
as revision control or source control, is the management of changes to
documents, computer programs, large web sites, and other collections of
information
```

Example of such software:

- *Git*
- Subversion (SVN)
- Mercurial
- Bazaar

9.9 Binary tree

This is hierarchy of directories where *CMake* will store generated files and where *native build tool* will store it's temporary files. Directory will contain variables/paths which are specific to your environment so they doesn't mean to be shareable. E.g. you **should never** store files from this directory to *VCS*. Keeping binary tree in a separate directory from *source tree* is a good practice and called *out-of-source build*.

Directory can be specified by *-B* option from command line or by `Browse Build...` in CMake-GUI.

See also:

- *Source tree*
- *-B*
- *GUI + Visual Studio*
- *GUI + Xcode*
- *Files generated by CMake is not designed to be relocatable*

9.10 Cache variables

For optimization purposes there are special type of variables which lifetime is not limited with one CMake run (e.g. like *regular cmake variables*). Variables saved in *CMakeCache.txt* file and persist across multiple runs within a project build tree [1].

[1] Quote from documentation.

9.11 CMake module

Listfiles located in directories specified by CMAKE_MODULE_PATH and having extension `.cmake` called **modules**. They can be loaded by `include` command. Unlike `add_subdirectory` command `include(<modulename>)` doesn't create new node in a source/binary tree hierarchies and doesn't introduce new scope for variables.

Note: In general by `include` you can load file with any name, not only `*.cmake`. For example:

```
include(some/file/abc.tt) # file with extension '.tt'
include(another/file/XYZ) # file without extension
```

Or even `CMakeLists.txt`:

```
include(foo/bar/CMakeLists.txt)
```

Though it is confusing, doesn't make sense and should be avoided.

CMake documentation

- Modules
- include

9.12 CMake variables

Regular CMake variables. Unlike *cache variables* lifetime of regular variables limited with CMake run.

CMake documentation

- Variables

9.13 CMakeCache.txt

File with *CMake cache variables*.

9.14 CMakeLists.txt

CMakeLists.txt is a *listfile* which plays the role of entry point for current source directory. CMake processing will start from top level `CMakeLists.txt` in *source tree* and continue with other dependent `CMakeLists.txt` files added by add_subdirectory directive. Each `add_subdirectory` will create new node in the source/binary tree hierarchy and introduce new scope for variables.

CMake documentation

- Directories

9.15 Developer Command Prompt

Developer Command Prompt is a Command Prompt with Visual Studio development tools available in environment:

```
> where msbuild
C:\Program Files (x86)\MSBuild\14.0\Bin\MSBuild.exe
C:\Windows\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe

> where cl
...\msvc\2015\VC\bin\cl.exe

> where dumpbin
...\msvc\2015\VC\bin\dumpbin.exe
```

Similar test on regular Command Prompt `cmd.exe`:

```
> where msbuild
INFO: Could not find files for the given pattern(s).

> where cl
INFO: Could not find files for the given pattern(s).

> where dumpbin
INFO: Could not find files for the given pattern(s).
```

Note: There is no need to use Developer Command Prompt for running CMake with `Visual Studio` generators, corresponding environment will be loaded automatically by CMake. But for other generators like `NMake` or `Ninja` you should start CMake from Developer Command Prompt.

See also:

- *Visual Studio*
- Developer Command Prompt for Visual Studio

9.16 Listfile

A file with CMake code. Usually (but not always) it's a *CMakeLists.txt* that is loaded by `add_subdirectory` command or module **.cmake* loaded by `include` command.

CMake documentation

- CMAKE_CURRENT_LIST_DIR
- CMAKE_CURRENT_LIST_FILE
- CMAKE_CURRENT_LIST_LINE

9.17 Multi-configuration generator

Generator that allows to use several build types on build step while doing only one configure step. List of available build types can be specified by CMAKE_CONFIGURATION_TYPES. Default value for `CMAKE_CONFIGURATION_TYPES` is a list of:

- `Debug`
- `Release`
- `MinSizeRel`
- `RelWithDebInfo`

Example of configuring `Debug` + `Release` project and building `Debug` variant:

```
> cmake -H. -B_builds -DCMAKE_CONFIGURATION_TYPES=Release;Debug -GXcode
> cmake --build _builds --config Debug
```

It is legal to use same `_builds` directory to build `Release` variant without rerunning configure again:

```
> cmake --build _builds --config Release
```

Multi-configuration generators:

- Xcode
- Visual Studio

CGold

- *Single-configuration generator*

9.18 One Definition Rule (ODR)

ODR is a rule for C++ programs that forbids declarations of the entities with same name but by different C++ code. Better/exact description is out of the scope of this document, please visit the links below for details if needed.

As a brief overview you can't do things like:

```
// Boo.hpp

class Foo {
  int a;
};
```

```
// Bar.hpp

class Foo {
  double a; // ODR violation, defined differently!
};
```

Though this code looks trivial and violation is obvious, there are scenarios when it's no so easy to detect such kind of errors, e.g. see examples from *Library Symbols* section.

See also:

- One Definition Rule

9.19 Single-configuration generator

Generator that allows to have only single build type while configuring project. Build type defined by CMAKE_BUILD_TYPE on configure step and can't be changed on build step.

Example of building `Debug` variant:

```
> cmake -H. -B_builds -DCMAKE_BUILD_TYPE=Debug
> cmake --build _builds
```

To use another build type like `Release` use *out-of-source feature*.

All generators that are not *multi-configuration* are single-configuration. Typical example of such generator is a Unix Makefiles generator.

9.20 Source tree

Hierarchy of directories with source files such as CMake/C++ sources. *CMake* starts with the *CMakeLists.txt* from top of the source tree. This directory can be set by *-H* in command line or by `Browse Source...` in CMake-GUI.

This directory is mean to be shareable. E.g. probably you should not store hard-coded paths specific to your local environment in this code. This is directory that you want to be managed with *VCS*.

See also:

- *-H*
- *Binary tree*
- *GUI + Visual Studio*
- *GUI + Xcode*