Workshop Modern C++ build environment

Bart Beumer (bart.beumer@alten.nl)

Alten

2025-07-29

Rationale

Using modern tools in combination with C & C++ we can describe the build environment, 3rd party libraries and executables to build.
We will be able to create a highly reproducable build environment regardless of the OS running on the developer PC (as long as it can run docker).

This workshop aims to demonstrate how to create this build environment, how to use the tools, introduce ourselves to this way of working using included demonstration projects.

Tools

We will be using a combination of tools, each having their own purpose.

- Dev containers [1]
  A development container (or dev container for short) allows you to use a container as a full-featured development environment.
- Conan [2]
  Open source, decentralized and multi-platform package manager.
- CMake [3]
  ... an open source, cross-platform family of tools designed to build, test, and package software.

[1] https://containers.dev/

[2] https://conan.io/faq

[3] https://cmake.org/about/

Dev containers

Solves the problem of having an encapsulated environment containing all tools, files, configuration that makes up the development environment.

Supported by multiple tools like:

- Visual Studio Code [4]
- IntelliJ IDEA [5]
- GitHub codespaces [6]

[4] https://containers.dev/supporting

[5] https://blog.jetbrains.com/idea/2024/07/using-dev-containers-in-jetbrains-ides-part-1/

[6] https://docs.github.com/en/codespaces/about-codespaces/what-are-codespaces

Dev containers

We need the following files:

- Dockerfile
- devcontainer.json

Dev containers

Using docker compose we create the entire container.

```
1  FROM alpine:3.21.3
2
3  # Install tools required for building.
4  RUN apk update && \
5      apk add --no-cache \
6      autoconf bash build-base cmake gdb \
7      git libstdc++ libtool linux-headers \
8      m4 perl python3 py3-pip\
9      && \
10     pip install --break-system-packages conan && \
11     conan profile detect
```

Dev containers

A devcontainer.json contains metadata used by tools.

```
{
	"name": "Workshop Modern C++ Dev",
	"build": {
		"dockerfile": "Dockerfile"
	},
	// Configure tool-specific properties.
	"customizations": {
		"vscode": {
			"settings": {},
			"extensions": [
				"ms-vscode.cpptools",
				"twxs.cmake",
				"ms-vscode.cmake-tools",
				"konicy.conan-extension"
			]
		}
	}
}
```

Dev containers

We will experiment with these files during the workshop.

Conan

Conan is a package manager for C and C++ which aims to solve some very common and difficult challenges.[7]

- Conveniently depend on 3rd party packages and use them.
- Automates the process of downloading, building & deploy for further use.
- Provide a mechanism to use libraries.

[7] https::/conan.io/faq

Conan

A simple text file can be used to configure dependencies on packages.

```
[requires]
boost/1.88.0
gtest/1.14.0
[generators]
CMakeDeps
CMakeToolchain
[layout]
cmake_layout
```

Conan

We will experiment with these files during the workshop.

CMake

CMake is an open source, cross-platform family of tools designed to build, test, and package software. CMake gives you control of the software compilation process using simple independent configuration files. Unlike many cross-platform systems, CMake is designed to be used in conjunction with the native build environment. [8]

- Good documentation and tutorials provided.[9]
- Tools run locally on a system.

[8] https://cmake.org/about/

[9] https://cmake.org/cmake/help/latest/guide/tutorial/index.html

CMake

Using files we can describe libraries, executables, the (external) dependencies they have.

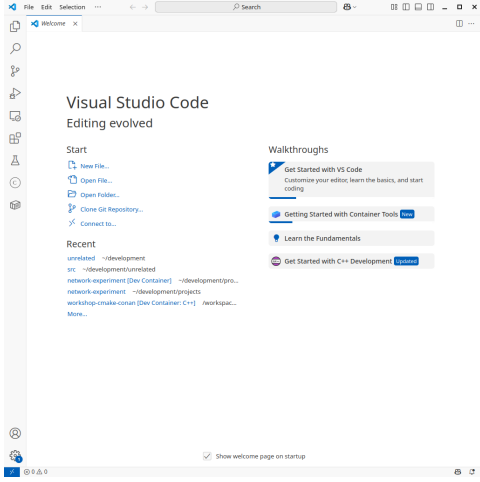
```
1  cmake_minimum_required(VERSION 3.20)
2
3  project(hello_world)
4
5  add_executable(
6      ${PROJECT_NAME}
7      src/main.cpp
8  )
9
10 set_property(
11     TARGET ${PROJECT_NAME}
12     PROPERTY CXX_STANDARD 20
13 )
```

CMake

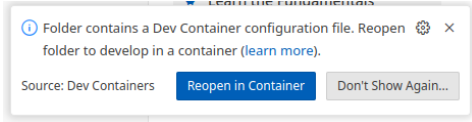
We will experiment with these files during the workshop.

Workshop

Lets start with opening our IDE and our project folder.



Quickly after opening the folder we get the following question:

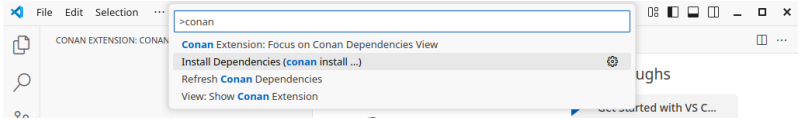


VS Code will build the container if needed and start using it.

Workshop

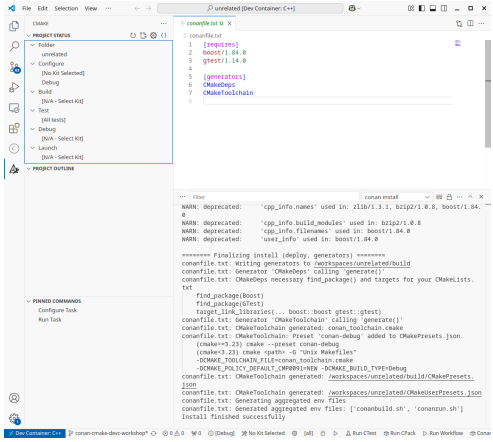
The dev container has been built, now we need to let conan work and fetch and build dependencies.

CTRL + SHIFT + P “conan install”



Workshop

We can now use CMake to build our software.



- configure
- build
- debug/launch

Workshop

Continuing with theory

Conan: important commands

`pip install conan`

We use the python package manager to install conan. (part of Dockerfile)

`conan profile detect`

We let conan detect compilers and tools for which it can use. (part of Dockerfile)

`conan install`

While in the project working directory, download packages, compile them, generate CMake environment. (was run using vs code)
After that we can use CMake as we would normally.

CMake

CMake: The basics

```
1  cmake_minimum_required(VERSION 3.20)
2
3  project(hello_world)
4
5  add_executable(
6      ${PROJECT_NAME}
7      src/main.cpp
8  )
9
10 set_property(
11     TARGET ${PROJECT_NAME}
12     PROPERTY CXX_STANDARD 20
13 )
```

CMake: The basics

```
1 cmake_minimum_required(VERSION 3.20)
```

- Sets the minimum required version of cmake for a project.
- Sets behavior of CMake as it was at a certain version (set policies)[10]

A mechanism to provide backwards compatibility while allowing future versions to change behavior of commands.

[10] https://cmake.org/cmake/help/latest/command/cmake_minimum_required.html

CMake: The basics

```
2
3  project(hello_world)
4
5  add_executable(
6      ${PROJECT_NAME}
7      src/main.cpp
8  )
```

- "project"
  Used to set the PROJECT_NAME variable.
- "add_executable"
  Adding executable by specifying the executable name and the source files.

CMake: The basics

```
9
10 set_property(
11     TARGET ${PROJECT_NAME}
12     PROPERTY CXX_STANDARD 20
13 )
```

- "set_property"
  Enables us to set properties like language standard to use (C++, CUDA, ...)

CMake: The basics, library

```
3  project(awesome_log)
4
5  add_library(
6      ${PROJECT_NAME}
7      ./src/console_writer.cpp
8      ./src/log.cpp
9      ./src/null_writer.cpp
10     ./src/scoped_log.cpp
11 )
12
13 set_property(
14     TARGET ${PROJECT_NAME}
15     PROPERTY CXX_STANDARD 20
16 )
17
18 target_include_directories(
19     ${PROJECT_NAME}
20     PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include
21 )
```

CMake: The basics, library

```
5  add_library(
6      ${PROJECT_NAME}
7      ./src/console_writer.cpp
8      ./src/log.cpp
9      ./src/null_writer.cpp
10     ./src/scoped_log.cpp
11 )
```

Similar to "add_executable" we can create a library, specifying the name of the library and the sources that form it.

CMake: The basics, library

```
18 target_include_directories(
19     ${PROJECT_NAME}
20     PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include
21 )
```

Add include directories, for both creating the target itself AND targets depending on this target.

No need for consumers of the library to set include paths, they inherit from this library target.

Using 3rd-party packages

Conan: 3rd-party packages

```
1 [requires]
2 boost/1.84.0
3 gtest/1.14.0
4
5 [generators]
6 CMakeDeps
7 CMakeToolchain
```

- "[requires]"
  Declare the libraries that we want to use in our project.
- "[generators]"
  Tell Conan to generate files for given build system

Conan: 3rd-party packages the python way

```
1  import os
2  from conan import ConanFile
3  from conan.tools.files import copy
4  from conan.tools.cmake import CMake, CMakeToolchain, CMakeDeps
5
6  class HelloConan(ConanFile):
7      settings = "os", "compiler", "build_type", "arch"
8      requires = "boost/1.84.0", "gtest/1.14.0", "libmagic/5.45"
9      generators = "CMakeDeps"
10     build_policy = "*"
11
12     def generate(self):
13         # We need to find the folder of libmagic and supply it to cmake so that
14         # we can deploy the magic file.
15         libmagic = self.dependencies["libmagic"]
16
17         tc = CMakeToolchain(self)
18         tc.variables["CONAN_LIBMAGIC_PACKAGE_FOLDER"] = libmagic.package_folder
19         tc.generate()
```

CMake: Using 3rd-party packages

```
1  cmake_minimum_required(VERSION 3.20)
2
3  find_package(Boost 1.88.0 REQUIRED COMPONENTS serialization)
4
5  project(example_boost)
6
7  add_executable(
8      ${PROJECT_NAME}
9      ./src/main.cpp
10 )
11 set_property(
12     TARGET ${PROJECT_NAME}
13     PROPERTY CXX_STANDARD 20
14 )
15 target_link_libraries(
16     ${PROJECT_NAME}
17     Boost::serialization
18 )
```

CMake, 3rd-party packages

```
2
3  find_package(Boost 1.88.0 REQUIRED COMPONENTS serialization)
```

Find a package (usually provided by something external to the project), and load its package-specific details.[11]

```
15 target_link_libraries(
16     ${PROJECT_NAME}
17     Boost::serialization
18 )
```

Using exposed target names we can use those packages.

- Affects libraries linked to target.
- Affects include paths to look for headers.

[11] https://cmake.org/cmake/help/latest/command/find_package.html

CMake: Testing

`enable_testing()`

Use enable_testing() at the top level CMakeLists.txt to enable cmake to generate things needed to support testing[12].

`add_test()`

Use add_test() inside your CMakeLists.txt to register the test[13].

`ctest`

Commandline application to run all registered tests[14].

[12] https://cmake.org/cmake/help/latest/command/enable_testing.html

[13] https://cmake.org/cmake/help/latest/command/add_test.html

[14] https://cmake.org/cmake/help/latest/manual/ctest.1.html

Worshop time! Suggestions:

- Play around in the environment, have a look at some of the projects.
- Try and add a 3rd party package and use it (fmt, ...)
- Use a different compiler.
- Add a tool to your image (valgrind, ...).