

KIV/CPP - Programming in C++

13. The CMake Tool. C++ Libraries – Boost, Qt, ImGui and Others

Martin Úbl

KIV ZČU

2024/2025

- a tool for dynamically generating build configurations
- generates MSVS solutions, makefiles, Codeblocks projects, etc...
- portable across all dominant platforms
- CLI utility
- also has a GUI frontend
- integration into MSVS

- root file
- usually contains the definition of the minimum version

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.4)
```

- always contains the name of the whole project (workspace, solution)

```
PROJECT(my-fancy-cpp-app)
```

- for clarity, subdirectories can also contain a CMakeLists.txt file that will be loaded separately; we invoke it by adding the subdirectory

```
ADD_SUBDIRECTORY(src)
```

- none of this affects the project structure yet

- adding an executable to build (or an MSVS project within a solution)

```
ADD_EXECUTABLE(<target-name> <file list...>)
```

- e.g.

```
ADD_EXECUTABLE(my-executable main.cpp module.cpp othermodule.cpp)
```

- this generates the file my-executable for building (or my-executable.exe on Windows)

- adding a library to build (or an MSVS project within a solution) works analogously

```
ADD_LIBRARY(<target-name> [library type] <file list...>)
```

- where [library type] can be, for example:
 - SHARED - dynamically linked library (.dll, .so, .dylib)
 - STATIC - statically linked library (.lib, .a)
- e.g.

```
ADD_LIBRARY(my-toolset STATIC libmain.cpp a.cpp b.cpp)
```

- this generates the library my-toolset.a for building (or my-toolset.lib on Windows)

- include directories can be added globally / for a single target

```
INCLUDE_DIRECTORIES(<dirs>)  
TARGET_INCLUDE_DIRECTORIES(<target> <specifier> <dirs>)
```

- the value of specifier can be
 - INTERFACE
 - PUBLIC - this one will be enough for us
 - PRIVATE
- e.g. for adding the include directory

```
INCLUDE_DIRECTORIES(${INCLUDE_DIRECTORIES} include/)  
TARGET_INCLUDE_DIRECTORIES(my-executable PUBLIC  
    ${INCLUDE_DIRECTORIES} include/)
```

- linking libraries also globally / for a single target
- note - globally this must be done before adding targets

```
LINK_LIBRARIES(<libraries>)  
TARGET_LINK_LIBRARIES(<target> <libraries>)
```

- e.g.

```
LINK_LIBRARIES(my-toolset)  
TARGET_LINK_LIBRARIES(my-executable my-toolset)
```

- the library name is either a concrete library from the system/from a path
- or one of the libraries that CMake has defined for building

- a CMake module is needed
- CMake has some bundled with it
- a module can be invoked with

```
FIND_PACKAGE(<packagename> [flag])
```

- only one flag will interest us: `REQUIRED` - it is not possible to continue without this library
- e.g.

```
FIND_PACKAGE(OpenSSL REQUIRED)
```

- the module sets certain variables
- the others are highly specific

- internal CMake modules set standardized variables
- always `<packagename>_FOUND` if the library can be found
- often `<packagename>_INCLUDE_DIR` - where to look for header files
- often `<packagename>_LIBRARIES` - list of libraries for linking
- all modules bundled with CMake are documented:
<https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html>

- it is often necessary to pass some switch to parameterize the build
- again, this can be done globally and for a target

```
ADD_DEFINITIONS(<defines>)  
TARGET_COMPILE_DEFINITIONS(<target> <specifier> <defines>)
```

- note, defines must include the `-D` prefix for preprocessor directives (so they behave like `#define`)
- e.g.

```
ADD_DEFINITIONS(-DDEV_BUILD)  
TARGET_COMPILE_DEFINITIONS(my-executable PUBLIC -DDEV_BUILD)
```

- so that we do not have to tediously list files, we can let them be found
- after being found, they are stored in a variable
- we will use the FILE command, which has broader uses

```
FILE(<operation> <parameters>)
```

- for finding files in a given directory with the .cpp and .h extensions, for example:

```
FILE(GLOB_RECURSE my_app_sources ./ *.cpp *.h)
```

- the GLOB_RECURSE operation also traverses subdirectories
- if we do not want that, only GLOB can be used

- the FetchContent component
- CMake can download dependencies if they can be built automatically

```
INCLUDE(FetchContent)
```

```
FetchContent_Declare(  
    googletest  
    GIT_REPOSITORY https://github.com/google/googletest.git  
)
```

```
FetchContent_MakeAvailable(googletest)
```

- defines the variables `<package>_SOURCE_DIR` and `<package>_BINARY_DIR`, which can be used
- in the example above: `googletest_SOURCE_DIR`, `googletest_BINARY_DIR`

- and more...
- control structures - IF, FOR, ...
- console messages - MESSAGE
- package generation, e.g. dpkg - CPack
- ...
- <https://cmake.org/cmake/help/latest/>

- C++ has a fairly large ecosystem of libraries
- also thanks to the existence of libraries for C and backward compatibility
- but there are also many purely C++ libraries

- Boost - a general library extending the std library and functionality
- Qt - a GUI library that has expanded into supporting functionality as well
- Wt - web framework, DBO
- Eigen - a library of mathematical structures and operations
- ACE - a high-level abstraction over networking
- SDL - a library for 2D graphics
- OpenCV - computer vision and image manipulation
- Ogre3D, Irrlicht - libraries for 3D graphics
- Box2D - physics simulation
- Intel TBB - parallelization (KIV/PPR)
- Google Test, cppunit - testing frameworks
- Log4cpp - logging library
- and thousands more...

- Boost
- a general library, extending the standard library and functionality
- “runs ahead of” the C++ standard
- C++ standard developers participate in Boost library development
- many concepts and principles were adopted directly from Boost
 - threads
 - any, variant, optional, tuple
 - filesystem
 - smart pointers
 - generation of (pseudo-)random numbers
 - `enable_if`
 - chrono
 - some standard algorithms
 - and many others

- Boost
- the boost namespace
- a large part is part of the standard and all modern standard libraries
- this does not mean it is not worth using for some things
- e.g. backward compatibility with older compilers
 - compilation on an obsolete compiler with support only for C++03
 - for some things it is enough to replace `std::` with `boost::`
 - the interface is adapted to the standard

- Boost
- download
 - <https://www.boost.org/users/download/>
 - <https://github.com/boostorg/boost>
 - package system (apt, ...)
- complete documentation
 - <https://www.boost.org/doc/libs/>
- build
 - in case we downloaded the source files

- build - Windows

```
bootstrap.bat
b2.exe --toolset=msvc-14.1
      --address-model=64
      --architecture=x86
      --runtime-link=static,shared
      --link=static
      --build-dir=build\x64
      --prefix="C:\boost"
      threading=multi
      install
```

- the result will be copied to the specified directory (prefix), paths must be configured in MSVS

- GNU/Linux (Debian)
- complete development package

```
apt install libboost-all-dev
```

- parts

```
apt install libboost-tools-dev  
            libboost-thread-dev ...
```

- the required libraries are stored in standard paths, as are the header files

- some Boost library modules
 - Algorithm - a collection of algorithms, some are in the STL
 - Asio - network communication
 - Beast - HTTP and WebSocket client/server
 - Bimap - “double map” - either value can be the key
 - Compute - a high-level wrapper over OpenCL
 - CRC - calculation of CRC codes
 - DLL - a high-level abstraction over working with libraries
 - Fiber - a library for user-level threads
 - Geometry - a library for geometric calculations
 - Graph - graph creation and graph algorithms
 - Log - logging library
 - Math - additional mathematical algorithms and functions
 - Meta State Machine - a library for finite-state machines

- some Boost library modules
 - Multiprecision - a library for higher-precision calculations
 - Polygon - algorithms for working with polygons
 - Process - cross-platform work with processes
 - Python - creating Python bindings for C++
 - Stacktrace - runtime stacktrace output
 - Test - testing library
 - Timer - timer library
 - and more...
 - <https://www.boost.org/doc/libs/>

- Qt
- <https://www.qt.io/>
- a cross-platform library for GUI creation
- expansion - extension with additional functionality
 - XML parsing
 - network communication
 - database communication
 - communication over Bluetooth
 - mobile applications - sensors, GPS, in-app purchases, NFC, ...
 - and more...
 - <http://doc.qt.io/qt-6/qtmodules.html>
- GUI remains central

- Qt
- <https://www.qt.io/>
- two distributions
 - commercial
 - open-source (LGPLv3) - free for non-commercial use
- the installer downloads the selected components
- optionally integrates with MSVS (Windows)
- can be used with C++, but also with Python, Rust, Go, C#, Haskell, ...

- Qt
- the problem with GUI across platforms
 - Windows uses messages and WinAPI
 - Unix systems use signals/X11
- Qt unifies the API using signals and slots (observer pattern)
- signal
 - an event that an object emits when some change occurs
 - the body is not implemented
 - e.g. button click, value change, ...
- slot
 - signal receiver
 - the body is implemented
 - e.g. a handler for what happens after pressing a button, ...

- Qt
- GUI definition
 - programmatically by adding elements
 - QML
 - Qt Creator (uses the above)
- we will deal with the definition in C++ code

- Qt
- various GUI elements
 - QWidget - a general (empty) widget
 - QPushButton - button
 - QLabel - text element
 - QLineEdit - text input field
 - QGroupBox - grouping elements into a visual area
 - QDateTimeEdit, QDateTimeEdit - fields for entering date and time
 - QCheckBox - checkbox
 - QRadioButton - radio button
 - layouts

- Qt
- layouts
 - QVBoxLayout - vertically arranged elements
 - QHBoxLayout - horizontally arranged elements
 - QGridLayout - grid arrangement
 - QFormLayout - two-column arrangement
 - and more...

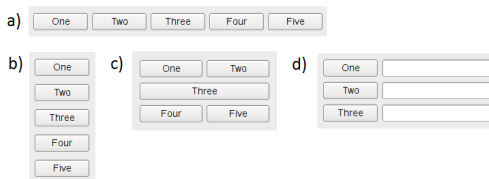


Figure: a) QHBoxLayout, b) QVBoxLayout, c) QGridLayout, d) QFormLayout

- every descendant of `QObject` can have signals/slots
 - it is possible even without inheriting from `QObject`
- every class using signals and slots must contain the `Q_OBJECT` macro
- a “new section” signals for signals and slots for slots
 - in reality, just a macro that Qt treats as a marker
 - header files then have to be processed by the Qt moc tool

```
class MyQObject : public QObject {  
    Q_OBJECT  
  
    private slots:  
        // ...  
    signals:  
        // ...  
};
```

- connecting a slot to a signal
- the `connect()` method
- signals and slots of different objects can be connected

```
QObject::connect(&a, &MyObj::valueChanged,  
                &b, &Handler::onValueChanged);
```

- - when object a emits the `valueChanged` signal, the `onValueChanged` method of object b is called

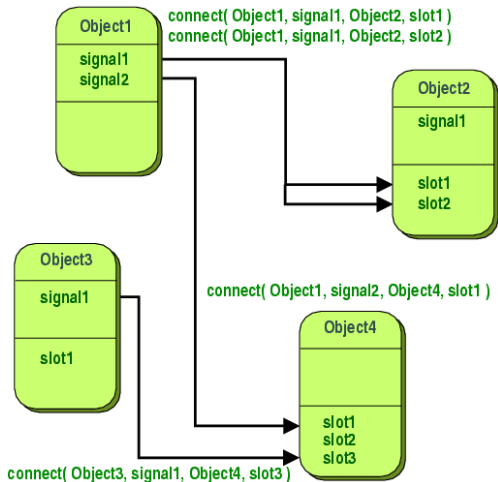


Figure: <http://doc.qt.io/archives/qt-4.8/signalsandslots.html>

- connecting a slot to a signal
- e.g. pressing a button

```
QPushButton* btn = new QPushButton("Push_it");  
QObject::connect(btn, &QPushButton::clicked,  
                 this, &MyWindow::onBtnClicked);
```

```
// in the class declaration  
private slots:  
    void onBtnClicked();
```

```
// in the implementation  
void MyWindow::onBtnClicked() {  
    // ...  
}
```

- Qt
- the library is fairly extensive
- many widgets, behaviors
- many supporting libraries
- <http://doc.qt.io/qt-6/index.html>
- defining a GUI programmatically is not very convenient
- higher comfort - Qt Creator GUI, QML
- or another language (C# + XAML, Java + JavaFX, Python, ...) and use C++ for the backend in the form of a library

- ImGui
- an *immediate mode* GUI library
- a library for creating user interfaces that has a different philosophy compared to Qt
- <https://www.dearimgui.com/>

- *immediate mode*
- in Qt, we define elements, pass them to Qt, and Qt renders them
- immediate mode relies on requesting the drawn elements ourselves in every rendering cycle

```
int clicks = 0;

while (true) {
    ImGui::NewFrame();
    ImGui::Begin("Hello, \uworld!");

    ImGui::Text("Clicked: \u%d", clicks);

    if (ImGui::Button("Button")) {
        clicks++;
    }

    ImGui::End();
    ImGui::Render();
}
```

- ImGui needs a so-called backend to function
- available, for example:
 - OpenGL + GLFW
 - SDL2, SDL3
 - glut
 - DirectX
- it is usually downloaded separately
- we choose it once for the whole project
- the selected service code depends on it
 - e.g. initialization, opening a window

ImGui + GLFW + OpenGL3

```
GLFWwindow* window = glfwCreateWindow(1280, 720, "Hello", nullptr);

ImGui_ImplGlfw_InitForOpenGL(window, true);
ImGui_ImplOpenGL3_Init();

ImVec4 clear_color = ImVec4(0.00f, 0.00f, 0.05f, 1.00f);

while (!glfwWindowShouldClose(window)) {
    glfwPollEvents();

    ImGui_ImplOpenGL3_NewFrame();
    ImGui_ImplGlfw_NewFrame();

    // ImGui drawing, ...

    int display_w, display_h;
    glfwGetFramebufferSize(window, &display_w, &display_h);
    glViewport(0, 0, display_w, display_h);
    glClearColor(clear_color.x * clear_color.w, clear_color.y * clear_c
    glClear(GL_COLOR_BUFFER_BIT);
    ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());

    glfwSwapBuffers(window);
}
```

- examples

```
if (ImGui::Button("Button")) {  
    messageIdx = (messageIdx + 1) % messages.size();  
}  
  
ImGui::Text("%s", messages[messageIdx]);
```