Organizing Larger Projects

Overview

Up to now a project scaffold has (mostly) been provided to you

- A substantial challenge in larger projects is simply organizing the project itself
- Bad project organization incurs enormous unnecessary overhead, promotes bugs, impedes extensibility and maintainability, ...

This lecture attempts to give some suggestions and an overview of useful tools

- Project layout suggestions (tailored to CMake)
- Integrating third-party tools and libraries with CMake
- Advanced debugging facilities
- We do not claim completeness or bias-free presentation
- Refer to the CMake documentation for much more detail

Project Layout (1)

The general project layout affects several interconnected properties

- Directory and source tree structure
- Namespace structure
- Library and executable structure

Changes to one of these properties likely entail changes to the other properties

- Namespace structure should (roughly) reflect directory structure and vice-versa
- Different libraries and executables ideally reside in separate source trees (i.e. directories)

Project Layout (2)

The project layout will evolve as a project grows

- Different guidelines apply to projects of different size
- Things one might get away with in small projects can become major issues in large projects
- Things that might be necessary in large projects can be overkill in small projects
- If a project is known to grow to a large size it pays off to plan ahead
- Definition of "small" and "large" is subjective

General guidelines

- *Always* clearly organize files, directories and namespaces with modularization in mind
- Start with a monolithic library/executable structure and move to a more independent and modular structure as the project grows

Directory Structure

General directory structure guidelines

- Files belonging to different libraries and executables should reside in different directories
- Files belonging to different components (logically separate parts) within a library or executable should reside in different directories
- Files belonging to different top-level namespaces should reside in different directories
- Tests should reside in a separate directory tree from the actual implementation
- Out-of-source builds should *always* be preferred

Directory Structure: Small Projects (1)

Directory structure guidelines for small projects

- The general directory structure guidelines still apply
- Parts of the `CMakeLists.txt` may be shared by all components within the project
  - Build system setup (e.g. compiler flags)
  - Dependencies (e.g. third-party libraries)
- The test code and executable(s) may be shared by all components within the project

Evolution

- Eventually, some library or executable in a small project will grow large
- Should then be moved into an independent (sub-)project

Directory Structure: Small Projects (2)

Small project example

```
> tree project
project
├── CMakeLists.txt          # Common CMakeLists.txt logic
├── my_executable
│   ├── CMakeLists.txt      # CMakeLists.txt logic for my_executable
│   └── ...                 # Source (& header) files
├── my_library
│   ├── CMakeLists.txt      # CMakeLists.txt logic for my_library
│   └── ...                 # Source (& header) files
└── test
    ├── CMakeLists.txt      # CMakeLists.txt logic for testing
    ├── my_executable
    │   └── ...             # Tests for my_executable
    └── my_library
        └── ...             # Tests for my_library
```

Directory Structure: Large Projects (1)

Directory structure guidelines for large projects

- The general directory structure guidelines still apply
- The components of large projects should be mostly independent subprojects
- Should *not* share most `CMakeLists.txt` logic
- Should *not* share test code and executable(s)

Evolution

- Eventually other projects or people may want to reuse one of the subprojects in a different context
- Should then be moved into an entirely independent project

Directory Structure: Large Projects (2)

Large project example

```
> tree project
project
├── CMakeLists.txt          # Minimal common CMakeLists.txt logic
├── my_executable
│   ├── CMakeLists.txt      # Common my_executable CMakeLists.txt
│   ├── src
│   │   └── ...             # Source (& header) files
│   └── test
│       ├── CMakeLists.txt  # CMakeLists.txt logic for tests
│       └── ...             # Tests for my_executable
└── my_library
    ├── CMakeLists.txt      # Common my_library CMakeLists.txt
    ├── src
    │   └── ...             # Source (& header) files
    └── test
        ├── CMakeLists.txt  # CMakeLists.txt logic for tests
        └── ...             # Tests for my_library
```

Header and Implementation Files

File content

- Generally, there should be one separate pair of header and implementation files for each C++ class
- Very tightly coupled classes (e.g. classes that could also be nested classes) can be placed in the same header and implementation files

File location

- Option 1: Place associated implementation and header files in the same directory (preferred by us)
- Option 2: Place associated implementation and header files in separate directory trees (e.g. `src` and `include`)
- Option 1 makes browsing code somewhat easier, option 2 makes system-wide installation easier

Namespaces & Cycles

Namespaces should identify logically coherent components within a library or executable

- Usually, there should be at least a top-level namespace (i.e. don't put stuff in the default namespace)
- Namespaces should group broadly similar or coherent functionality
- Rule of thumb: Think of namespaces as "candidates for moving into a separate library"

Dependencies between namespaces should be cycle-free

- Makes refactoring code *much* easier
- Allows future modularization into separate libraries

Library & Executable Structure

It is usually advisable to separate executables from their core functionality

- Executables often serve as "frontends" to some library functionality
- Library functionality can probably be reused in other programs
- Keeps interaction logic (e.g. I/O) separate from core functionality
- Not necessary in very small projects

There should be a separate `CMakeLists.txt` for each library or executable

- Implies that separate libraries and executables reside in separate directories
- Facilitates future modularization into separate (sub-)projects
- The `add_subdirectory` CMake function can be used to aggregate several such sub-projects

Include Directories

Usually, the include path for a library should contain a prefix

- E.g. includes for a library “foo” could start with `#include "foo/..."`
- Requires a suitable directory structure in the source tree of the library
- Usually requires the use of `target_include_directories` in the `CMakeLists.txt`

```
> tree project/my_library
my_library
├── CMakeLists.txt
├── src
│   └── my_library
│       ├── Bar.hpp
│       ├── Foo.hpp
│       └── Foo.cpp
└── test
    └── ...
```

Libraries & Executables

In most cases, libraries and executables are the main product of a CMake project

- Encoded as *targets* in a CMake project
- Targets can have properties such as dependencies
- CMake projects may contain further targets (e.g. for installing, packaging, linting, etc.)

Libraries

- Collection of compiled code that can be reused in other libraries or executables
- Can either be *static* or *shared* libraries
- Have to conform to the OS application binary interface (ABI)
- Cannot be executed on their own

Executables

- Compiled code that can be executed on a certain operating system
- Have to conform to the OS application binary interface (ABI)
- May contain further metadata such as information about entry points etc.

Executables in CMake (1)

Executables are added with the `add_executable` CMake command

- Syntax: `add_executable(name sources...)`
- Adds a CMake target with the specified name
- Produces an executable with the specified name in the same relative directory as the current `CMakeLists.txt`
- `sources...` can be a whitespace-separated list of source files or a CMake variable that expands to such a list
- Passing source files by variable should be preferred for more than a few files
- Properties such as dependencies can be modified through additional CMake commands

Executables in CMake (2)

Sample `CMakeLists.txt` for the `my_executable` sub-project

```
set(MY_EXECUTABLE_SOURCES
    src/my_executable/Helper.cpp
    ...
    src/my_executable/Main.cpp
)

add_executable(my_executable ${MY_EXECUTABLE_SOURCES})

# further commands required
```

Static Libraries

Static libraries are essentially archives of executable code

- Contain assembly from some number of object files, e.g. for classes, functions, etc.
- Dependencies on static libraries are resolved at *link time*
- Static libraries on Linux typically have the extension `*.a`

The linker is responsible for resolving dependencies on static libraries

- Code from a static library `A` is copied into a library or executable `B` that depends on `A`
- At runtime, no dependency on `A` exists since the relevant code is part of the library or executable `B`

Shared Libraries

Shared libraries are dynamic archives of executable code

- Contain assembly from some number of object files, e.g. for classes, functions, etc.
- Dependencies on shared libraries are resolved at *runtime*
- Shared libraries on Linux typically have the extension `*.so`

The operating system is responsible for resolving dependencies on shared libraries

- Only pointers to the code in a shared library `A` are used in a library or executable `B` that depends on `A`
- At runtime, the operating system loads `A` into memory once
- All programs depending on `A` access this memory to execute code in `A`

Advantages and Disadvantages of Static Libraries

Advantages

- Can have slightly higher performance since there are no indirections
- Can prevent compatibility issues since there are no external dependencies

Disadvantages

- Much bigger file sizes than shared libraries since code is actually copied
- Programs depending on static libraries have to be recompiled if the static library changes
- Can lead to problems with transitive dependencies even if they are "header only"

Advantages and Disadvantages of Shared Libraries

Advantages

- Much smaller file sizes since the shared library is only loaded into memory at run time
- Much lower memory consumption since only a single copy of a shared library is kept in memory (even for unrelated processes)
- Can be exchanged for other compatible versions without changing programs that depend on a shared library

Disadvantages

- Programs depending on a shared library rely on a compatible version being available
- Can be slightly slower due to additional indirection at runtime

Static Libraries in CMake (1)

Static libraries are added with the `add_library` CMake command

- Syntax: `add_library(name STATIC sources...)`
- Adds a CMake target with the specified name
- Produces a static library with the specified name in the same relative directory as the current `CMakeLists.txt`
- `sources...` can be a whitespace-separated list of source files or a CMake variable that expands to such a list
- Passing source files by variable should be preferred for more than a few files
- Properties such as dependencies can be modified through additional CMake commands

Static Libraries in CMake (2)

Sample `CMakeLists.txt` for the `my_library` sub-project (assuming static library)

```
set(MY_LIBRARY_SOURCES
    src/my_library/ClassA.cpp
    ...
    src/my_library/ClassZ.cpp
)

add_library(my_library STATIC ${MY_LIBRARY_SOURCES})

# further commands required
```

Shared Libraries in CMake (1)

Shared libraries are added with the `add_library` CMake command

- Syntax: `add_library(name SHARED sources...)`
- Adds a CMake target with the specified name
- Produces a shared library with the specified name in the same relative directory as the current `CMakeLists.txt`
- `sources...` can be a whitespace-separated list of source files or a CMake variable that expands to such a list
- Passing source files by variable should be preferred for more than a few files
- Properties such as dependencies can be modified through additional CMake commands

Shared Libraries in CMake (2)

Sample `CMakeLists.txt` for the `my_library` sub-project (assuming shared library)

```
set(MY_LIBRARY_SOURCES
    src/my_library/ClassA.cpp
    ...
    src/my_library/ClassZ.cpp
)

add_library(my_library SHARED ${MY_LIBRARY_SOURCES})

# further commands required
```

Interface Libraries in CMake (1)

Usually *only* the implementation files (`*.cpp`) should be added to a CMake target

- Header files on their own are not compiled
- Only headers that are included by implementation files are relevant for compilation

Exception: Interface libraries

- Syntax: `add_library(name INTERFACE)`
- A library might contain only template definitions
- Cannot be compiled into a static or shared library (unless explicit instantiation is used)
- Can still have properties such as include paths or dependencies

Interface Libraries in CMake (2)

Sample CMakeLists.txt for the my_library sub-project (assuming header-only)

```
add_library(my_library INTERFACE)
target_include_directories(my_library INTERFACE src)
target_link_libraries(my_library INTERFACE some_dependency)
```

Nested Projects in CMake (1)

The `add_subdirectory` CMake command can be used to add a subproject

- Syntax: `add_subdirectory(source_dir)`
- Adds the `CMakeLists.txt` in the specified `source_dir` to the build
- The nested `CMakeLists.txt` will be processed immediately by CMake
- The CMake variable `CMAKE_SOURCE_DIR` refers to the top-level source directory inside nested `CMakeLists.txt`
- The CMake variable `CMAKE_CURRENT_SOURCE_DIR` refers to the source directory in which the nested `CMakeLists.txt` resides

Nested Projects in CMake (2)

Example top-level `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.12)
project(project)

# more general setup code ...

add_subdirectory(my_executable)
add_subdirectory(my_library)
```

Important Project Properties (1)

Usually, the include directory of libraries and executables needs to be set

- `target_include_directories(target PUBLIC|PRIVATE dirs...)`
- Should be set to the `src` or `include` directory of a subproject in our suggested layout
- `PUBLIC` include directories are passed on to targets that depend on the current target

Important Project Properties (2)

Dependencies between targets can be set with `target_link_libraries`

- `target_link_libraries(target PUBLIC|PRIVATE libs...)`
- `libs...` can refer to libraries defined by the current project or imported third-party library targets
- `PUBLIC` dependencies are passed on to targets that depend on the current target

Important Project Properties (3)

Sample `CMakeLists.txt` for the `my_executable` sub-project

```
set(MY_EXECUTABLE_SOURCES
    src/my_executable/Helper.cpp
    ...
    src/my_executable/Main.cpp
)

add_executable(my_executable ${MY_EXECUTABLE_SOURCES})
# allows includes to be '#include "my_executable/..."
# instead of '#include "my_executable/src/my_executable/..."
target_include_directories(my_executable PRIVATE src/)
# dependency on the my_libary target defined in other subproject
target_link_libraries(my_executable PRIVATE my_library)
```

Paths in CMake

CMake defines several variables for often-used paths

`CMAKE_SOURCE_DIR`
: Contains the full path to the top level of the source tree, i.e. the location of the top-level `CMakeLists.txt`

`CMAKE_CURRENT_SOURCE_DIR`
: Contains the full path the the source directory that is currently being processed by CMake. Differs from `CMAKE_SOURCE_DIR` in directories added through `add_subdirectory`.

`CMAKE_BINARY_DIR`
: Contains the full path to the top level of the build tree, i.e. the build directory in which `cmake` is invoked.

`CMAKE_CURRENT_BINARY_DIR`
: Contains the full path the binary directory that is currently being processed. Each directory added through `add_subdirectory` will create a corresponding binary directory in the build tree.

Relative paths are usually relative to the current source directory

Third-Party Libraries

Usually we do not want to reinvent the wheel

- There is a vast ecosystem of (open-source) third-party libraries
- If there exists a well-maintained third-party library that matches your requirements you should use it

If possible and feasible, your project should *not* bundle third-party dependencies

- Many libraries can easily be installed through a package manager
- Reduces complexity of project configuration and maintenance
- CMake provides facilities for locating third-party dependencies in a platform-independent way

find_package (1)

Preferred CMake function for locating third-party dependencies

- `find_package(<PackageName> [version] [REQUIRED])`
- Finds and loads settings from an external project
- Sets the `<PackageName>_FOUND` CMake variable if the package was found
- May provide additional variables and imported CMake targets depending on the package

`find_package` relies on CMake scripts

- Attempts to find a `Find<PackageName>.cmake` file in the path specified by the `CMAKE_MODULE_PATH` variable and in the CMake installation
- Many `Find*.cmake` scripts are provided by CMake itself
- CMake documentation can be consulted for details about provided `Find*.cmake` scripts
- Own `Find*.cmake` scripts can be written if necessary

find_package (2)

Example

```
...

# Attempt to locate system-wide installation of libgtest
# Invokes the FindGTest.cmake script provided by CMake
# Configuration will fail if libgtest cannot be found
find_package(GTest REQUIRED)

add_executable(tester ...)
target_link_libraries(tester PRIVATE
    ...
    GTest::GTest  # Imported target for the gtest library
                  # as specified by the documentation of
                  # FindGTest
)
```

find_library (1)

If no Find*.cmake script is available, find_library can be used

- find_library(<VAR> name [path1 path2 ...])
- Creates a cache entry named <VAR> to store the result of the command
- If nothing is found, the result will be <VAR>-NOTFOUND
- name specifies the name of the library (e.g. gtest for libgtest)
- Additional paths beside the default search paths can be specified

find_library simply searches directories for a library

- A wide range of (highly configurable) paths is searched for the library
- Does not automatically configure non-standard include paths like find_package
- Should only be used as a fallback or within Find*.cmake scripts

find_library (2)

Example (assuming there is no `FindGTest.cmake` script)

```
...

# Attempt to locate libgtest library
# Searches for the library file in a range of paths
find_library(GTest gtest)

if (${GTest} STREQUAL "GTest-NOTFOUND")
    message(FATAL_ERROR "libgtest not found")
endif()

add_executable(tester ...)
target_link_libraries(tester PRIVATE
    ...
    GTest  # Only adds the libgtest library
           # Does not set include paths
)
```

Further Reading

We only scratched the surface of CMake in this lecture

- CMake provides much more highly useful functionality
- E.g. checks for compiler flags
- E.g. checks for compiler features
- E.g. checks for host system features
- E.g. defining custom Makefile targets
- ...

The CMake documentation provides a good overview

Testing

Tests should be an integral part of every larger project

- Unit tests
- Integration tests
- ...

Good test coverage greatly facilitates implementing a large project

- Tests can ensure (to some extent) that modifications do not break existing functionality
- Can easily refactor code
- Can easily change the internals of a component
- ...

Googletest (1)

We use Googletest in the programming assignments and final project

- Works on a large variety of platforms
- Contains a large set of useful functions
- Can usually be installed through a package manager
- Can be added to a CMake project through the `FindGTest.cmake` module
- Alternative test frameworks are of course available

Functionality overview

- Test cases
- Predefined and user-defined assertions
- Death tests
- ...

Googletest (2)

Simple tests

```
#include <gtest/gtest.h>
//-----------------------------------------------------------
TEST(TestSuiteName, TestName) {
    ...
}
```

- Defines and names a test function that belongs to a test suite
- Test suites can for example map to one class or function
- Googletest assertions can be used to control the outcome of the test function
- If any assertion fails or the test function crashes, the entire test case fails

Googletest (3)

Fatal assertions

- Fatal assertions are prefixed with `ASSERT_`
- When a fatal assertion fails the test function is immediately terminated

Non-fatal assertions

- Non-fatal assertions are prefixed with `EXPECT_`
- When a non-fatal assertion fails the test function is allowed to continue
- Nevertheless the test case will fail
- All assertions exist in fatal and non-fatal versions

Assertion examples

- `ASSERT_TRUE(condition);` or `ASSERT_FALSE(condition);`
- `ASSERT_EQ(val1, val2);` or `ASSERT_NE(val1, val2);`
- ...

Googletest (4)

A custom `main` function needs to be provided for Googletest

```
#include <gtest/gtest.h>
//---------------------------------------------------------
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

- Should usually be placed in a separate `Tester.cpp` or `main.cpp`

Coverage (1)

Code coverage can help ensure proper testing of a project

- Simple metrics like line coverage have to be interpreted carefully
- Can indicate that a certain part of a project has *not* been tested properly
- Can usually *not* indicate that a certain part of a project has been tested exhaustively

Line coverage information can automatically be collected during test execution

- Possible with a variety of tools
- GCC contains the build-in coverage tool `gcov`
- Clang can produce `gcov`-like output
- `lcov` together with `genhtml` can be used to generate HTML line coverage reports from information collected during test execution

Coverage (2)

Brief example

```
# build executable with gcov enabled
> g++ -fprofile-arcs -ftest-coverage -o main main.cpp

# run executable and generate coverage data
> ./main

# generate lcov report
> lcov --coverage --directory . --output-file coverage.info

# generate html report
> genhtml coverage.info --output-directory coverage
```

- Produces HTML coverage report in `coverage/index.html`
- Configuration for coverage reports should be part of CMake configuration

Continuous Integration

Platforms like GitLab provide *continuous integration* (CI) functionality

- Can automatically run tests or other checks each time some commits are pushed to GitLab
- Highly useful in larger projects with multiple contributors
- Can be used to enforce certain standards in a project (e.g. minimum line coverage, no failing tests etc.)
- Has to be taken seriously to be effective (e.g. refuse merge requests with failing CI tests etc.)

Configured through `.gitlab-ci.yml` file in the repository

- Rather complex initial server-side setup
- Already provided by our GitLab server
- `.gitlab-ci.yml` configures the CI for a certain GitLab repository
- Refer to the GitLab documentation for details

Linting

A *linter* performs static source code analysis

- Can detect some types of "bad" code
- Some forms of bugs
- Stylistic errors that may lead to bugs
- Suspicious constructs that may lead to bugs

`clang-tidy` is a `clang`-based C++ linter

- Widely available through package manager
- Highly configurable set of checks (e.g. through `.clang-tidy` file)
- Integrated in CLion
- Can be integrated in CMake configuration of a project

`perf` (1)

`perf` is a highly useful performance analysis tool for Linux

- Can profile any program using the standalone executable `perf`
- Can be integrated in a program by using the `perf` API
- Can interface with hardware and software performance counters

Standalone `perf` examples

- `perf stat [OPTIONS] command`
  - Run command and display information about event counts such as cache misses, branch misses etc.
- `perf record [OPTIONS] command`
  - Run command and sample a certain event on the instruction level
  - If possible, command should be built with debug symbols
- `perf report`
  - Analyze a file generated by `perf record`
  - Generates an interactive report that shows sampled event counts for each instruction.

perf (2)

perf stat example

```
> perf stat --detailed ./my_executable
...
Performance counter stats for './my_executable':

      56.505,78 msec task-clock                #    2,573 CPUs utilized
        854.187      context-switches          #    0,015 M/sec
          7.827      cpu-migrations            #    0,139 K/sec
        309.550      page-faults               #    0,005 M/sec
177.728.516.281      cycles                    #    3,145 GHz
 60.347.961.620      instructions              #    0,34  insn per cycle
 12.694.777.815      branches                  #  224,663 M/sec
     89.725.841      branch-misses             #    0,71% of all branches
 16.672.843.754      L1-dcache-loads           #  295,064 M/sec
  1.267.581.260      L1-dcache-load-misses     #    7,60% of all L1-dcache hits
    471.681.999      LLC-loads                 #    8,347 M/sec
    258.238.607      LLC-load-misses           #   54,75% of all LL-cache hits

21,964215591 seconds time elapsed

44,360970000 seconds user
16,626546000 seconds sys
```

Valgrind

Valgrind is a general-purpose dynamic analysis tool

- Mainly used for memory debugging, memory leak detection and profiling
- Essentially runs programs on a virtual machine, allowing tools to do arbitrary transformations on the program before execution
- Extremely high overhead compared to other tools like ASAN

Use cases

- Complex memory bugs that are not detected by simpler tools like the address sanitizer
- Complex profiling tasks

Reverse Debugging (1)

Regular debuggers like GDB can only step forward in the program

- Does not necessarily fit debugging requirements
- E.g. when a crash occurs, we would like to step *backwards* until we have found the source of the crash

Reverse debuggers provide such functionality

- Usually, a program run is recorded first
- Subsequently, the program run can be replayed reproducing the exact same behavior
- During debugging, execution can step forward and backward in time
- Example: `rr` by Mozilla

Reverse Debugging (2)

Buggy class

main.cpp

```cpp
#include <cassert>
//------------------------------------------------------------
struct Foo {
    static constexpr int max = 15;
    int a = 0;

    void bar() {
        assert((a % 2) == 0);
        a = (a + 2) % max;
    }
};
//------------------------------------------------------------
int main() {
    Foo foo;
    for (unsigned i = 0; i < 16; ++i)
        foo.bar();
}
```

Reverse Debugging (3)

rr example

```
> g++ -g -o main main.cpp
> rr record main        # record execution of main, including crash
> rr replay             # start rr GDB session, will break at _start
...
(rr) continue           # continue program until crash
(rr) up 4               # go to Foo::bar stack frame
(rr) watch -l a         # hardware watchpoint for Foo::a
(rr) reverse-continue   # continue backwards, will break at SIGABRT
(rr) reverse-continue   # continue backwards, will break at watchpoint
Continuing.

Hardware watchpoint 1: -location a

Old value = 1
New value = 14
0x00005568dba67208 in Foo::bar (this=0x7fff75f38980) at main.cpp:9
9                         a = (a + 2) % max;
```