Concepts of C++ Programming

Lecture 14: Larger Projects

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

Parallelism in C++

- Writing correct parallel code is *hard*
- Writing efficient parallel code is *extremely hard*

- Writing efficient parallel C++ requires understanding of hardware
  - Especially: atomic operations and memory ordering

$\leadsto$ Be especially careful when writing parallel code

Libraries and Executables

Executables

- Compiled code that can be executed on a certain OS
- Can depend on other libraries
- Can be executed directly
- Code cannot be reused elsewhere

Libraries

- Compiled code that can be reused in libraries or executables
- Can depend on other libraries
- Cannot be executed on their own
- Can be static/shared library

Separating Libraries and Executables

- Usually advisable to separate executables from core functionality
- Executables: front-end for library functionality
  - Keeps interaction logic separate (e.g., I/O, parsing) from core functionality
- Library functionality can be reused in other programs
  - E.g., unit tests, other executable, etc.

⇒ Put libraries in separate directories with separate `CMakeLists.txt`

- Use CMake's `add_subdirectory`; also eases future modularization

Libraries: Include Directories

- Usually, library include path should contain prefix
- E.g., for library `foo`: `#include "foo/..."`
- Requires suitable directory layout

```
mylib/
+-- CMakeLists.txt
+-- include/
    +-- mylib/
        +-- Module.hpp
        +-- Printer.hpp
+-- src/
    +-- Module.cpp
    +-- Printer.cpp
```

Static Libraries

- Static library: archive of object files
- Dependencies resolved at link-time
- Typical extensions: `.a` (Windows: `.lib`)

- During linking, static libraries are copied into executable
- At runtime, no dependency on the library exists

+ No indirections, no compatibility issues
– Larger file size due to copying, need recompile if lib changes

Shared Libraries

- Shared library: collection of linked object files
- Dependencies resolved at program startup
- Typical extensions: `.so` (Windows: `.dll`)

- During loading, system needs to search for libraries
- At runtime, library is loaded into memory just once
  - *All* programs that use the library share the same code

+ Smaller size, lower memory consumption, can exchange compatible versions

− Slower due to additional runtime indirection, compatibility is hard

Shared Libraries: ABI Compatibility

- Application Binary Interface: interface between two compiled programs
  - Includes structure layouts, argument/return types, enum values, ...
  - C++: vtable layout, mangled names, ...
  - Also be careful when using the preprocessor
- Unintended ABI breaks can happen easily in C++
- Substitution of shared library requires compatible ABI
  - ABI-incompatible versions often have different so-names
  - Otherwise: might lead to subtle problems

Header-Only Libraries

- Some libraries only consist of header files
  - Example: only templated types
- Some people put everything in header files regardless
  - Primarily to simplify downstream adoption (no build system to integrate)

\+ Possibly easier to integrate

− Like static libraries; and longer compilation times

Libraries in CMake

```
add_library(my_libA STATIC
    src/A.cpp
    src/B.cpp
)
# ---
add_library(my_libB SHARED
    src/C.cpp
    src/D.cpp
)# ---
add_library(my_libC INTERFACE) # no source files
```

Libraries in CMake

- Include directory of libraries/executables needs to be set
  `target_include_directories(target PUBLIC|PRIVATE dirs...)`
  - Public: add to include path for the target and all its dependents
  - Private: add to include path just for the target

- Specify dependencies between target:
  `target_link_libraries(target PUBLIC|PRIVATE libs...)`
  - Adds dependencies: takes care of include paths and linker flags
  - Public: add dependency to the target and all its dependents
  - Private: add dependency just to the target

Third-Party Libraries

- Often, reinventing the wheel is not a good idea
- Reusing existing third-party libraries can save substantial effort
- However: be aware of the general downsides of dependencies

- If possible: *don't bundle* dependencies
  - Many libraries can be installed through a package manager
  - Use CMake's `find_package(<PackageName> [version] [REQUIRED])`
  - If no `Find*.cmake` is provided: `find_library(<VAR> name [path1 path2 ...])`
- Alternatively: submodules with CMake `add_subdirectory`

Interfacing with C

- C headers often surrounded by `extern "C" {...}`
- Changes language linkage to C for external declarations (= no name mangling)

```
//--- my-c-lib.h

#ifdef __cplusplus
extern "C" {
#endif

// Usual C header content

#ifdef __cplusplus
}
#endif
```

- If C header doesn't include wrappers: wrap `#include`

Other Build Systems

- Meson (e.g., GNOME, QEMU)
- Automake/Autoconf (e.g., GCC)
- SCons
- Bazel (e.g., Google)
- GN (e.g., Chromium)

Unified Builds

- Unified build: concatenate multiple source files into one compilation unit

+ Faster build times: less redundant parsing of headers
+ Enables more optimizations between `.cpp` files
– Longer incremental build times
– Possible correctness issues on naming collisions

Other Build Options

- Link-Time Optimization (LTO): Cross-CU Optimizations
  - Object files don't contain machine code, but internal compiler representation
  - Only at link time, everything gets compiled
- Profile-Guided Optimization (PGO):
  - First build with instrumentation to track taken branches etc.
  - Run application on typical load, collect profile
  - Second build that uses the profile for further optimizations
  - Can lead to substantial speedups in practice

Build Tools for Developers

- Pre-Compiled Headers (PCH): precompile headers to improve build times
  - CMake: `target_precompile_headers`
- C++20 Modules[159]
  - Module consists of multiple translation units, can import modules, can export declarations
  - Alternative to header files in certain situations
  - Faster compilation: exported definitions are compiled into binary format
  - Implementation still not ready, thus rarely used up to this point

159 https://en.cppreference.com/w/cpp/language/modules

Where to go from here?

- Advanced Concepts of Programming Languages
  - Covers memory model and C++ class implementation in detail
- Compiler Construction 1
  - Covers implementation of compiler front-ends
- Code Generation
  - Covers implementation of compiler back-ends

Thanks to...

- Michael Freitag, Moritz Sichert, and Maximilian Rieger for the lab course slides "Systems Programming in C++"
- Tobias Lasser for his adaption of the slides
- The various authors of cppreference
- Florian Drescher and Mateusz Gienieczko for the exercises

... and of course to YOU
for participating in this course!