CMake course for Elbit-Elisra

Alex Kushnir

About Claritune

Tailored Technical Courses:

- Modern C++ for Embedded Developers
- Secure Design and Development in C++
- Effective Unit Testing with GoogleTest
- C++ Multithreading and Concurrency
- Design Patterns for C++ Developers
- Linux Fundamentals
- DevOps: Docker
- DevOps: Kubernetes

and more...

CTO: Amir Kirsh

Previously the Chief Programmer at Comverse.

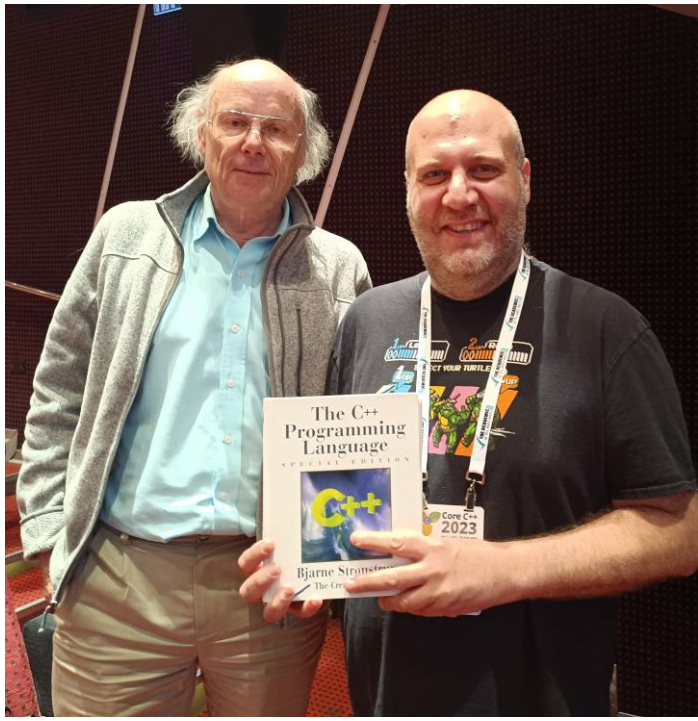
Lecturer at the Academic College of Tel-Aviv-Yaffo and Tel-Aviv University.

A team of top-level experts.

Trusted by a wide range of leading high-tech companies.

Who am I?

- Software engineer
- Mostly in the embedded and low-level domains
- Love C++
- Love CMake
- Delivered 2 talks on Core C++ 2024
  - Optimizing embedded software infrastructure
  - Mastering CTest
- Book reviews
  - Foreword for Modern CMake for C++
  - Editorial review for C++ for embedded systems
  - Post-release review for C++ memory management



Course goals

- Understand and explain the structure of a basic CMake project.
- Build C and C++ projects using CMake, including managing internal dependencies, defining project targets and integrating external dependencies into the project.
- Configure CMake for the selected toolchain, compiler and linker configuration, testing and coverage.
- Create CMake preset files to define a consistent development environment.

Topics - first day

- Introduction
- The CMake language
- Setting up a CMake project
- Hands-on exercise
- CMake targets
- C++ compilation and linkage using CMake
- Hands-on exercise

Topics - second day

- Managing dependencies with CMake
- Hands-on exercise
- Testing with CMake
- Program-analysis tools
- Installation and packaging
- Writing CMake presets

Introduction

What CMake is?

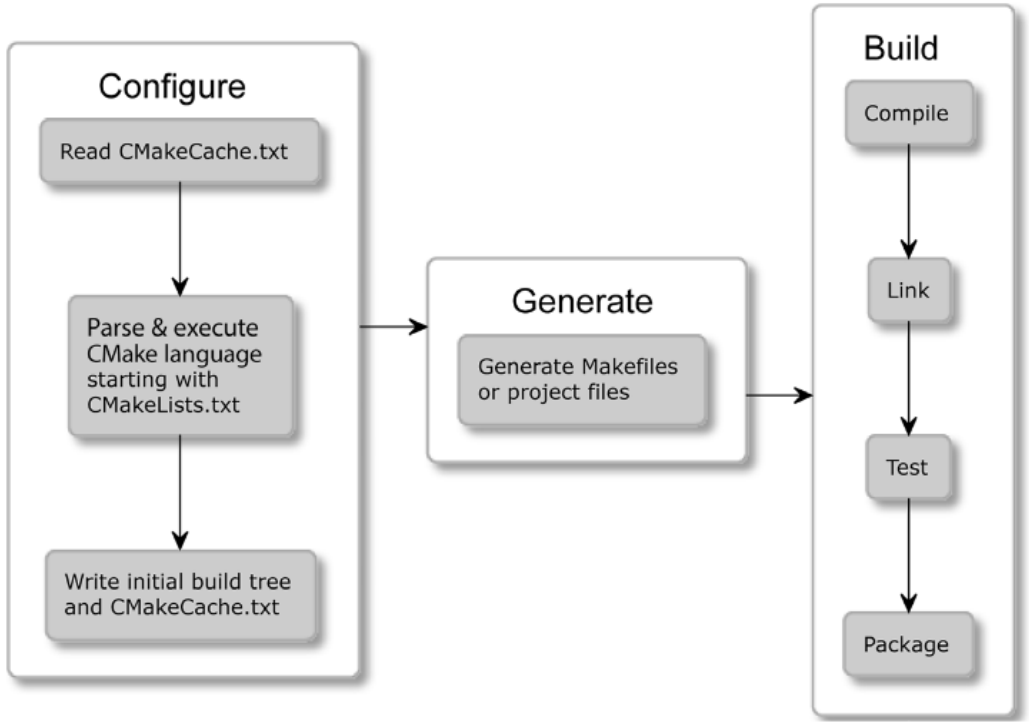
- A family of tools designed to build, test and package software.
- Cross-platform (Windows/Linux/macOS/Cygwin).
- Most modern compilers and toolchains are supported.
- Able to generate project files for all popular IDEs (Visual Studio, Eclipse, CLion, etc.)
- Able to generate Makefiles and ninjafiles
- Rich ecosystem.
- Old features get deprecated to keep CMake lean.
- Developed and maintained by Kitware Inc.

What CMake isn’t?

- CMake is not a build tool by itself
- Relies on other tools in the system

How it works?

- The process of building has 3 stages
  - Configuration
  - Generation
  - Building



Configuration stage

- Reading and parsing **CMakeLists.txt**
- Reading project details in the **source tree**
- Preparing output directory (**build tree**)
- Checking for cached steps (**CMakeCache.txt**)
- Collect the details about the environment – architecture, compilers, etc.
- Store the cached information in **CMakeCache.txt**

Generation stage

- Generate the **buildsystem** for the exact environment.
- These are the build files for other build systems (Make, ninja, etc.)
- The **configuration** and **generation** stages are happening at a single step.
- After the **generation** stage, a **project file** exists
  - Can be opened in IDE (Visual Studio, Eclipse, etc.)
  - Can be built using command line (Make, ninja)

Building stage

- Optional step
- CMake calls the build (test/packing...) tools

```
alex@WLPF46XAH6:~/cmake_course/01_intro$ cmake -B build -S . -G Ninja
-- The C compiler identification is GNU 13.3.0
-- The CXX compiler identification is GNU 13.3.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (1.6s)
-- Generating done (0.0s)
-- Build files have been written to: /home/alex/cmake_course/01_intro/build
alex@WLPF46XAH6:~/cmake_course/01_intro$ cmake --build build
[2/2] Linking CXX executable 01_intro
alex@WLPF46XAH6:~/cmake_course/01_intro$
```

Generators

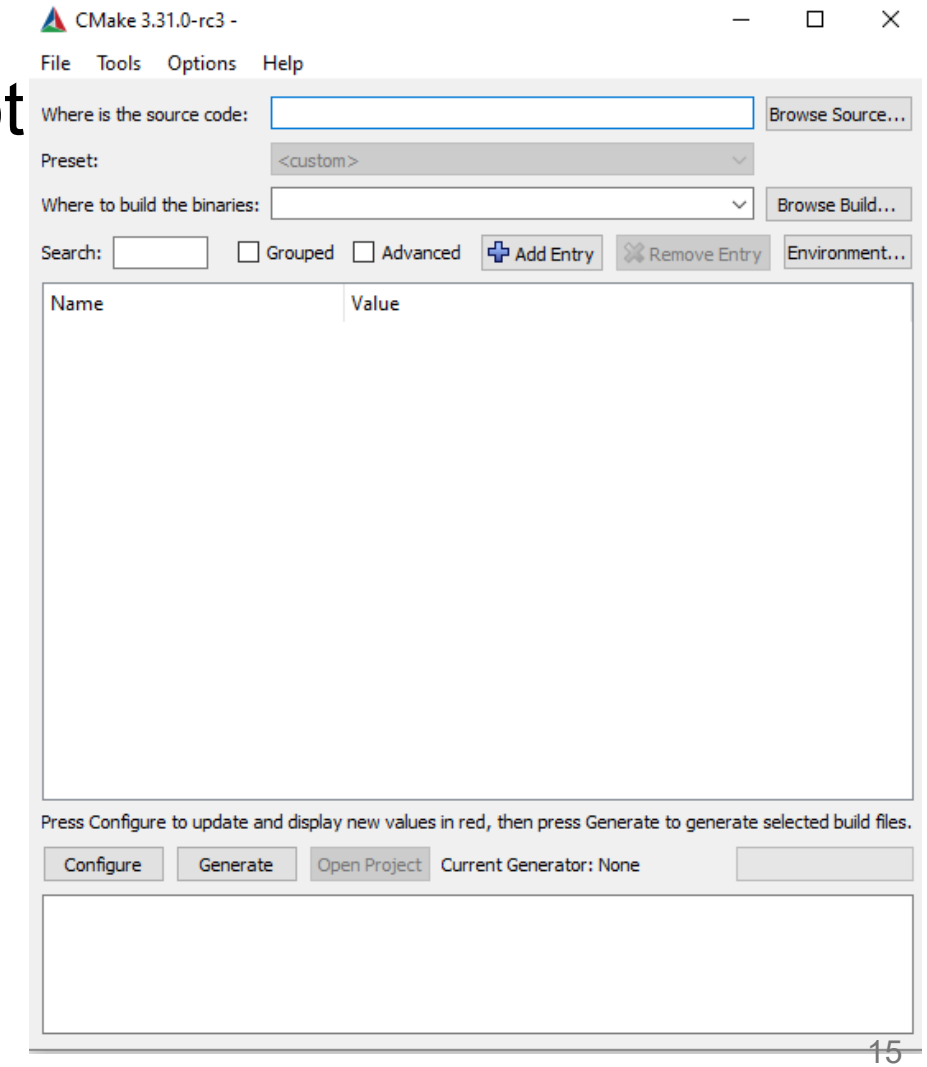
- `cmake --help`

```
Generators

The following generators are available on this platform (* marks default):
  Green Hills MULTI           = Generates Green Hills MULTI files
                                (experimental, work-in-progress).
* Unix Makefiles              = Generates standard UNIX makefiles.
  Ninja                       = Generates build.ninja files.
  Ninja Multi-Config          = Generates build-<Config>.ninja files.
  Watcom WMake                = Generates Watcom WMake makefiles.
  CodeBlocks - Ninja          = Generates CodeBlocks project files
                                (deprecated).
  CodeBlocks - Unix Makefiles = Generates CodeBlocks project files
                                (deprecated).
  CodeLite - Ninja            = Generates CodeLite project files
                                (deprecated).
  CodeLite - Unix Makefiles   = Generates CodeLite project files
                                (deprecated).
  Eclipse CDT4 - Ninja        = Generates Eclipse CDT 4.0 project files
                                (deprecated).
  Eclipse CDT4 - Unix Makefiles= Generates Eclipse CDT 4.0 project files
                                (deprecated).
  Kate - Ninja                = Generates Kate project files (deprecated).
  Kate - Ninja Multi-Config   = Generates Kate project files (deprecated).
  Kate - Unix Makefiles       = Generates Kate project files (deprecated).
  Sublime Text 2 - Ninja      = Generates Sublime Text 2 project files
                                (deprecated).
  Sublime Text 2 - Unix Makefiles
                              = Generates Sublime Text 2 project files
                                (deprecated).
```

```
Generators

The following generators are available on this platform (* marks default):
* Visual Studio 17 2022        = Generates Visual Studio 2022 project files.
                                 Use -A option to specify architecture.
  Visual Studio 16 2019        = Generates Visual Studio 2019 project files.
                                 Use -A option to specify architecture.
  Visual Studio 15 2017 [arch] = Generates Visual Studio 2017 project files.
                                 Optional [arch] can be "Win64" or "ARM".
  Visual Studio 14 2015 [arch] = Generates Visual Studio 2015 project files.
                                 Optional [arch] can be "Win64" or "ARM".
  Borland Makefiles            = Generates Borland makefiles.
  NMake Makefiles              = Generates NMake makefiles.
  NMake Makefiles JOM          = Generates JOM makefiles.
  MSYS Makefiles               = Generates MSYS makefiles.
  MinGW Makefiles              = Generates a make file for use with
                                 mingw32-make.
  Green Hills MULTI            = Generates Green Hills MULTI files
                                 (experimental, work-in-progress).
  Unix Makefiles               = Generates standard UNIX makefiles.
  Ninja                        = Generates build.ninja files.
  Ninja Multi-Config           = Generates build-<Config>.ninja files.
  Watcom WMake                 = Generates Watcom WMake makefiles.
  CodeBlocks - MinGW Makefiles = Generates CodeBlocks project files
                                 (deprecated).
  CodeBlocks - NMake Makefiles = Generates CodeBlocks project files
                                 (deprecated).
  CodeBlocks - NMake Makefiles JOM
                               = Generates CodeBlocks project files
                                 (deprecated).
  CodeBlocks - Ninja           = Generates CodeBlocks project files
                                 (deprecated).
  CodeBlocks - Unix Makefiles  = Generates CodeBlocks project files
                                 (deprecated).
  CodeLite - MinGW Makefiles   = Generates CodeLite project files
                                 (deprecated).
  CodeLite - NMake Makefiles   = Generates CodeLite project files
                                 (deprecated).
  CodeLite - Ninja             = Generates CodeLite project files
                                 (deprecated).
  CodeLite - Unix Makefiles    = Generates CodeLite project files
                                 (deprecated).
  Eclipse CDT4 - NMake Makefiles
                               = Generates Eclipse CDT 4.0 project files
                                 (deprecated).
  Eclipse CDT4 - MinGW Makefiles
                               = Generates Eclipse CDT 4.0 project files
                                 (deprecated).
  Eclipse CDT4 - Ninja         = Generates Eclipse CDT 4.0 project files
                                 (deprecated).
  Eclipse CDT4 - Unix Makefiles= Generates Eclipse CDT 4.0 project files
                                 (deprecated).
  Kate - MinGW Makefiles       = Generates Kate project files (deprecated).
  Kate - NMake Makefiles       = Generates Kate project files (deprecated).
  Kate - Ninja                 = Generates Kate project files (deprecated).
  Kate - Ninja Multi-Config    = Generates Kate project files (deprecated).
  Kate - Unix Makefiles        = Generates Kate project files (deprecated).
  Sublime Text 2 - MinGW Makefiles
                               = Generates Sublime Text 2 project files
                                 (deprecated).
  Sublime Text 2 - NMake Makefiles
                               = Generates Sublime Text 2 project files
                                 (deprecated).
  Sublime Text 2 - Ninja       = Generates Sublime Text 2 project files
                                 (deprecated).
  Sublime Text 2 - Unix Makefiles
                               = Generates Sublime Text 2 project files
                                 (deprecated).
```

CMake GUI

- Convenient tool for those who are not familiar with CMake CLI
- We are going to focus on CLI



Glossary

- **Source tree**
- **Build tree**
- **Project file**
- **Cache file**
- **Package definition file**
- **Generated files**
- **Preset files**
- **Configure log**
- **Scripts and utility modules**

Glossary – trees

- **Source tree**
  - The project root directory. It contains the main CMakeLists.txt
  - Given by the user with `-S` argument
- **Build tree**
  - The root folder where build configuration and build artifacts are created
  - Out-of-source builds are recommended (artifacts do not pollute the source tree)
  - Specified with `-B` argument
  - Don’t source-control this directory

Glossary – listfiles (1)

- **Project file**
  - **`CMakeLists.txt`** – top of the source tree
  - At least 2 commands
    - `cmake_minimum_required(VERSION X.XX)`
    - `project(<name> <OPTIONS>)` – stored in `PROJECT_NAME` variable
    - Can be structured – a top project file and several folders that have their own files
- **Cache file**
  - Named **`CMakeCache.txt`**
  - Resides in the root of the build tree
  - Let’s see an example

CMakeCache.txt example

```
//CXX compiler
CMAKE_CXX_COMPILER:FILEPATH=/usr/bin/c++

//A wrapper around 'ar' adding the appropriate '--plugin' option
// for the GCC compiler
CMAKE_CXX_COMPILER_AR:FILEPATH=/usr/bin/gcc-ar-14

//A wrapper around 'ranlib' adding the appropriate '--plugin' option
// for the GCC compiler
CMAKE_CXX_COMPILER_RANLIB:FILEPATH=/usr/bin/gcc-ranlib-14

//Flags used by the CXX compiler during all build types.
CMAKE_CXX_FLAGS:STRING=

//Flags used by the CXX compiler during DEBUG builds.
CMAKE_CXX_FLAGS_DEBUG:STRING=-g

//Flags used by the CXX compiler during MINSIZEREL builds.
CMAKE_CXX_FLAGS_MINSIZEREL:STRING=-Os -DNDEBUG

//Flags used by the CXX compiler during RELEASE builds.
CMAKE_CXX_FLAGS_RELEASE:STRING=-O3 -DNDEBUG

//Flags used by the CXX compiler during RELWITHDEBINFO builds.
CMAKE_CXX_FLAGS_RELWITHDEBINFO:STRING=-O2 -g -DNDEBUG

//C compiler
CMAKE_C_COMPILER:FILEPATH=/usr/bin/cc

//A wrapper around 'ar' adding the appropriate '--plugin' option
// for the GCC compiler
CMAKE_C_COMPILER_AR:FILEPATH=/usr/bin/gcc-ar-14

//A wrapper around 'ranlib' adding the appropriate '--plugin' option
// for the GCC compiler
CMAKE_C_COMPILER_RANLIB:FILEPATH=/usr/bin/gcc-ranlib-14
```

Glossary – listfiles (2)

- **Package definition file**
  - Config-files contain information regarding how to use the library
  - Sometimes they expose CMake macros or functions to be used
  - Named **`<PackageName>-config.cmake`** or **`PackageNameConfig.cmake`**
  - Use the **`find_package()`** command to include such packages
  - More on that later
- **Generated files**
  - A lot of temporary files are generated during the configuration process
  - Not intended to be edited manually
  - Examples: **`cmake_install.cmake, CTestTestFile.cmake`**

Glossary – JSON and YAML files (1)

- **Preset files**
  - Sometimes command-line configuration becomes too complicated
  - The project maintainer can provide a configuration file that stores all the details
  - Presets can be used to configure **workflows** which tie stages into a single step
  - **`CMakePresets.json`** – Official presets provided by the project maintainers
  - **`CMakeUserPresets.json`** – Extension of the former, can be used locally
  - More on that later

Glossary – JSON and YAML files (2)

- **Configure log**
  - CMake provides a structured log file to allow advanced debugging of the configure stage
  - **`<build-tree>/CMakeFiles/CMakeConfigureLog.yaml`**
  - Let's see an example

CMakeConfigureLog.yaml

```yaml
---
events:
  -
    kind: "message-v1"
    backtrace:
      - "/usr/share/cmake-3.28/Modules/CMakeDetermineSystem.cmake:233 (message)"
      - "CMakeLists.txt:3 (project)"
    message: |
      The system is: Linux - 6.6.87.2-microsoft-standard-WSL2 - x86_64
  -
    kind: "message-v1"
    backtrace:
      - "/usr/share/cmake-3.28/Modules/CMakeDetermineCompilerId.cmake:17 (message)"
      - "/usr/share/cmake-3.28/Modules/CMakeDetermineCompilerId.cmake:64 (__determine_compiler_id_test)"
      - "/usr/share/cmake-3.28/Modules/CMakeDetermineCCompiler.cmake:123 (CMAKE_DETERMINE_COMPILER_ID)"
      - "CMakeLists.txt:3 (project)"
    message: |
      Compiling the C compiler identification source file "CMakeCCompilerId.c" succeeded.
      Compiler: /usr/bin/cc
      Build flags:
      Id flags:

      The output was:
      0


      Compilation of the C compiler identification source "CMakeCCompilerId.c" produced "a.out"

      The C compiler identification is GNU, found in:
        /home/alex/cmake_course/build/CMakeFiles/3.28.3/CompilerIdC/a.out

  -
    kind: "message-v1"
    backtrace:
      - "/usr/share/cmake-3.28/Modules/CMakeDetermineCompilerId.cmake:17 (message)"
      - "/usr/share/cmake-3.28/Modules/CMakeDetermineCompilerId.cmake:64 (__determine_compiler_id_test)"
      - "/usr/share/cmake-3.28/Modules/CMakeDetermineCXXCompiler.cmake:126 (CMAKE_DETERMINE_COMPILER_ID)"
```

Glossary - Scripts and utility modules

- **Scripts**
  - Platform-agnostic programming language
  - Instead of writing utilities in Bash/Powershell, just abstract it away
  - No cache is used

```
# An example of a script
cmake_minimum_required(VERSION 3.26.0)
message("Hello world")
file(WRITE Hello.txt "I am writing to a file")
```

- **Utility modules**
  - CMake comes packed with over 80 utility modules
  - Vary from simple ones like `TestBigEndian` to a complex ones like those provided by `CTest` and `CPack`
  - Excellent repo - https://github.com/onqtam/awesome-cmake

Utility module example (1)

```
cmake_minimum_required(VERSION 3.28)
project(SysInfo)
include(CMakePrintSystemInformation)
```

```
# Distributed under the OSI-approved BSD 3-Clause License.  See accompanying
# file Copyright.txt or https://cmake.org/licensing for details.

#[=======================================================================[.rst:
CMakePrintSystemInformation
---------------------------

Print system information.

This module serves diagnostic purposes. Just include it in a
project to see various internal CMake variables.
#]=======================================================================]

message("CMAKE_SYSTEM is ${CMAKE_SYSTEM} ${CMAKE_SYSTEM_NAME} ${CMAKE_SYSTEM_VERSION} ${CMAKE_SYSTEM_PROCESSOR}")
message("CMAKE_SYSTEM file is ${CMAKE_SYSTEM_INFO_FILE}")
message("CMAKE_C_COMPILER is ${CMAKE_C_COMPILER}")
message("CMAKE_CXX_COMPILER is ${CMAKE_CXX_COMPILER}")


message("CMAKE_SHARED_LIBRARY_CREATE_C_FLAGS is ${CMAKE_SHARED_LIBRARY_CREATE_C_FLAGS}")
message("CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS is ${CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS}")
message("CMAKE_DL_LIBS is ${CMAKE_DL_LIBS}")
message("CMAKE_SHARED_LIBRARY_PREFIX is ${CMAKE_SHARED_LIBRARY_PREFIX}")
message("CMAKE_SHARED_LIBRARY_SUFFIX is ${CMAKE_SHARED_LIBRARY_SUFFIX}")
message("CMAKE_COMPILER_IS_GNUCC = ${CMAKE_COMPILER_IS_GNUCC}")
message("CMAKE_COMPILER_IS_GNUCXX = ${CMAKE_COMPILER_IS_GNUCXX}")

message("CMAKE_CXX_CREATE_SHARED_LIBRARY is ${CMAKE_CXX_CREATE_SHARED_LIBRARY}")
message("CMAKE_CXX_CREATE_SHARED_MODULE is ${CMAKE_CXX_CREATE_SHARED_MODULE}")
message("CMAKE_CXX_CREATE_STATIC_LIBRARY is ${CMAKE_CXX_CREATE_STATIC_LIBRARY}")
message("CMAKE_CXX_COMPILE_OBJECT is ${CMAKE_CXX_COMPILE_OBJECT}")
message("CMAKE_CXX_LINK_EXECUTABLE ${CMAKE_CXX_LINK_EXECUTABLE}")

message("CMAKE_C_CREATE_SHARED_LIBRARY is ${CMAKE_C_CREATE_SHARED_LIBRARY}")
message("CMAKE_C_CREATE_SHARED_MODULE is ${CMAKE_C_CREATE_SHARED_MODULE}")
message("CMAKE_C_CREATE_STATIC_LIBRARY is ${CMAKE_C_CREATE_STATIC_LIBRARY}")
message("CMAKE_C_COMPILE_OBJECT is ${CMAKE_C_COMPILE_OBJECT}")
```

Utility module example (2)

```
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
CMAKE_SYSTEM is Linux-6.6.87.2-microsoft-standard-WSL2 Linux 6.6.87.2-microsoft-standard-WSL2 x86_64
CMAKE_SYSTEM file is Platform/Linux
CMAKE_C_COMPILER is /usr/bin/cc
CMAKE_CXX_COMPILER is /usr/bin/c++
CMAKE_SHARED_LIBRARY_CREATE_C_FLAGS is -shared
CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS is -shared
CMAKE_DL_LIBS is dl
CMAKE_SHARED_LIBRARY_PREFIX is lib
CMAKE_SHARED_LIBRARY_SUFFIX is .so
CMAKE_COMPILER_IS_GNUCC = 1
CMAKE_COMPILER_IS_GNUCXX = 1
CMAKE_CXX_CREATE_SHARED_LIBRARY is <CMAKE_CXX_COMPILER> <CMAKE_SHARED_LIBRARY_CXX_FLAGS> <LANGUAGE_COMPILE_FLAGS> <LINK_FLAGS> <CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS> <SONAME_FLAG><
TARGET_SONAME> -o <TARGET> <OBJECTS> <LINK_LIBRARIES>
CMAKE_CXX_CREATE_SHARED_MODULE is <CMAKE_CXX_COMPILER> <CMAKE_SHARED_LIBRARY_CXX_FLAGS> <LANGUAGE_COMPILE_FLAGS> <LINK_FLAGS> <CMAKE_SHARED_LIBRARY_CREATE_CXX_FLAGS> <SONAME_FLAG><T
ARGET_SONAME> -o <TARGET> <OBJECTS> <LINK_LIBRARIES>
CMAKE_CXX_CREATE_STATIC_LIBRARY is
CMAKE_CXX_COMPILE_OBJECT is <CMAKE_CXX_COMPILER> <DEFINES> <INCLUDES> <FLAGS> -o <OBJECT> -c <SOURCE>
CMAKE_CXX_LINK_EXECUTABLE <CMAKE_CXX_COMPILER> <FLAGS> <CMAKE_CXX_LINK_FLAGS> <LINK_FLAGS> <OBJECTS> -o <TARGET> <LINK_LIBRARIES>
CMAKE_C_CREATE_SHARED_LIBRARY is <CMAKE_C_COMPILER> <CMAKE_SHARED_LIBRARY_C_FLAGS> <LANGUAGE_COMPILE_FLAGS> <LINK_FLAGS> <CMAKE_SHARED_LIBRARY_CREATE_C_FLAGS> <SONAME_FLAG><TARGET_S
ONAME> -o <TARGET> <OBJECTS> <LINK_LIBRARIES>
CMAKE_C_CREATE_SHARED_MODULE is <CMAKE_C_COMPILER> <CMAKE_SHARED_LIBRARY_C_FLAGS> <LANGUAGE_COMPILE_FLAGS> <LINK_FLAGS> <CMAKE_SHARED_LIBRARY_CREATE_C_FLAGS> <SONAME_FLAG><TARGET_SO
NAME> -o <TARGET> <OBJECTS> <LINK_LIBRARIES>
CMAKE_C_CREATE_STATIC_LIBRARY is
CMAKE_C_COMPILE_OBJECT is <CMAKE_C_COMPILER> <DEFINES> <INCLUDES> <FLAGS> -o <OBJECT> -c <SOURCE>
CMAKE_C_LINK_EXECUTABLE <CMAKE_C_COMPILER> <FLAGS> <CMAKE_C_LINK_FLAGS> <LINK_FLAGS> <OBJECTS> -o <TARGET> <LINK_LIBRARIES>
CMAKE_SYSTEM_AND_CXX_COMPILER_INFO_FILE
CMAKE_SYSTEM_AND_C_COMPILER_INFO_FILE
-- Configuring done (1.9s)
-- Generating done (0.0s)
-- Build files have been written to: /home/alex/cmake_course/build
```

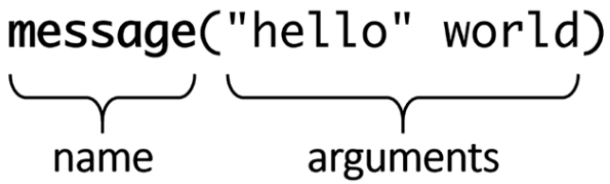
The CMake language

Syntax basics

- Very similar to any other declarative language
- Executed from top to bottom
- Everything in a listfile is a command invocation or a comment
- Comments
  - Single-line (#) or multiline (bracket) (`#[=[ … #]=]`) comments
  - Unlike C++, bracket comments can be nested
  - Use comments in a listfile just as you use them in the code

Syntax basics – invoking commands

- Invoking commands is the basic of the CMake listfiles
- The command syntax consists of the command name followed by parentheses with command arguments



- No case-sensitivity, but `snake_case` convention is preferred
- Command invocations are not expressions – can't pass command as argument to another command
- No semicolons

Toolchain file

- CMake can be invoked with a parameter that indicates the toolchain file
  - `cmake --toolchain /path/to/toolchain_file.cmake`
  - `cmake -DCMAKE_TOOLCHAIN_FILE=/path/to/toolchain_file.cmake`
- The toolchain file is a CMake script file that contains all necessary definitions for the build to succeed, such as compiler, linker, assembler names, path to sysroot, and more
- This can be handy when compiling for multiple platforms
- Tip – provide absolute path to the toolchain file
- Toolchains reference

Exercise - toolchain

- Objective – compile the project for both x86 and arm architectures

CMake commands

Technical prerequisites:

Configure CMake project: `cmake -S <source-tree> -B <build-tree>`

Build CMake project: `cmake -build <build-tree>`

Run CMake script: `cmake -P <script-name.cmake>`

CMake commands

- Always available
- Change the state of the command processor and variables
- Affect other commands and environment

- Project commands – affects only current project (scoped)

Command arguments (1)

- The only data type is string
- CMake evaluates every argument to a static string, then passes it to the command
- This can mean replacing escape sequences, expanding variable references and unpacking lists
- CMake offers 3 types of arguments
  - Bracket arguments
  - Quoted arguments
  - Unquoted arguments

Command arguments (2)

- **Bracket arguments**
  - Aren’t evaluated, used to pass multiline strings as a single arguments
  - Will include whitespace as tabs and newlines
  - Formatted identically to comments
  - This type of arguments is used rarely

```
message([[Hello
    CMake
    Multiline
    Argument
    ]])
```

```
Hello
    CMake
    Multiline
    Argument
```

Command arguments (3)

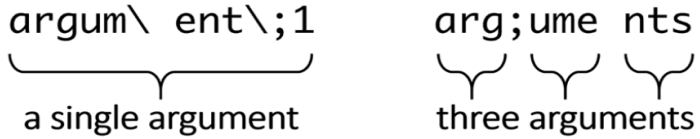
- **Quoted arguments**
  - Resemble a C++ string – group multiple characters
  - Including whitespaces
  - Expand escape sequences
  - Opened and closed using double-quotes
  - That is where the similarities with C++ strings end
  - Can be thinked of as having a built-in std::format function
  - Variables references can be formatted into the string using `${variable}`

```
message("Escape sequence: \t After tab\n New line with backslash\\\nCurrently using CMake version ${CMAKE_VERSION}")
```

```
Escape sequence:        After tab
 New line with backslash\
Currently using CMake version 3.28.3
```

Command arguments (4)

- **Unquoted arguments**
  - Dropping delimiters makes the code easier to read?
  - Evaluate both escape sequences and variable references
  - Be careful with ; - in CMake it means list delimiter
  - Unquoted arguments are most perplexing to work with



  - Cannot contain unescaped *, “, # and \
  - Parentheses () are allowed only if they form correct matching pairs

```
message(Single\ argument)
message(Two arguments)
message(Three;semicolon;arguments)
message(${CMAKE_VERSION})
```

```
Single argument
Twoarguments
Threesemicolonarguments
3.28.3
```

Unquoted arguments – when to use?

- Some CMake commands allow optional arguments
- Preceded by a keyword to signify an argument can be provided
- Example:

```
project(myProject VERSION 1.2.3)
```

- In this case, the **`VERSION`** keyword and `1.2.3` are optional and left unquoted for readability
- Note: keywords are case-sensitive

CMake variables

CMake variables

- Variable names are case sensitive
- Stored under the hood as strings
- But some commands can interpret them as other data types
- Basic variable manipulation commands - `set()` and `unset()`
- Bad naming is harder to read, therefore regular naming convention is encouraged – use only – and _ as special characters
- Avoid `CMAKE_`, `_CMAKE_` or _ prefixes

Variable references

- Use `${VARNAME}` syntax
- Upon evaluation, CMake traverses from innermost to outermost scope, and replaces `${VARNAME}` with value
- If no value is found, CMake will silently replace it with an empty string
- The evaluation is performed in an inside-out manner
- Example: `${OUTER${INNER}}`
  - Try to evaluate `${INNER}`
  - If successfully evaluated, repeat the expansion process until no new expansion is possible

```
set(INNER "INNER")
set(OUTER "OUTER")
message("${OUTER${INNER}}")
```

```
set(INNER "INNER")
set(OUTERINNER "OUTER")
message("${OUTER${INNER}}")
```

Variable references – normal and cache

- The following applies to variable references
  - The `${}` syntax is used to reference normal or cache variables
  - The `$ENV{}` syntax is used to reference environment variables
  - The `$CACHE{}` syntax is used to reference cache variables
- Using the `${}` syntax, if the variable was set in current scope, it will be used
- If not, a cached variable will be used instead
- If the variable is not in cache – evaluated to an empty string
- CMake predefines a lot of built-in normal variables for different purposes
- Example – the command is stored in `CMAKE_ARGV<n>` variables, and `CMAKE_ARGC` contains the count

```
message("Number of arguments: ${CMAKE_ARGC}")
foreach(argno RANGE ${CMAKE_ARGC})
    message("${CMAKE_ARGV${argno}}")
endforeach()
```

```
Number of arguments: 3
cmake
-P
test.cmake
```

Variable references - environment

- The simplest kind of variable
- CMake makes a copy of the environment and makes them available in a global scope
- CMake might change the environment, but the changes are made to the local copy only
- Several environment variables affect CMake behavior
- For example, the CXX variable specifies what compiler will be used to compile the source code
- Complete list of CMake environment variables: https://cmake.org/cmake/help/latest/manual/cmake-env-variables.7.html

```
message("User: $ENV{USER}, Homedir: $ENV{HOME}, Shell: $ENV{SHELL}")
```

```
User: alex, Homedir: /home/alex, Shell: /bin/bash
```

Variable references - cache

- Persistent variables stored in **`CMakeCache.txt`**
- Contain information gathered during configuration stage
- Originate from the system and from the user (CLI/GUI)
- Exist only in projects (scripts do not utilize cache)
- Can be referenced using ${} syntax (if normal variable is not set) or $CACHE{} syntax
- Defined using a special syntax of set() command:

```
set(<variable> <value> CACHE <type> <docstring> [FORCE])
```

- Extra variables are necessary for cache variables (type and docstring)
- The GUI requires this information to display them appropriately

```
set(FOO "BAR" CACHE STRING "interesting value" FORCE)
```

Several useful variables

- **`CMAKE_BUILD_TYPE`** – `Debug`, `Release`, `etc.`
- **`CMAKE_SOURCE_DIR`** – Where the top listfile is
- **`CMAKE_BINARY_DIR`** – Where the build tree is
- **`CMAKE_MAKE_PROGRAM`** – The build tool (make, ninja, etc.)
- **`CMAKE_C_FLAGS`**, **`CMAKE_CXX_FLAGS`** – pass arguments to the compiler if needed (very low-level, modern way exists)
- **`CMAKE_EXE_LINKER_FLAGS`**, **`CMAKE_SHARED_LINKER_FLAGS`** – Same as previous but for linker
- Variables are passed to CMake using –D command line switch

Variable scopes (1)

- CMake supports variable **scopes**, implemented in a specific way
- Meant to separate different layers of abstraction
- Outermost scope is called the global scope
- Nested scope can access global variables but not vice versa
- CMake has 2 kinds of variable scopes
  - **File** – used when blocks and functions are executed within a file
  - **Directory** – Used when `add_subdirectory()` command is called to execute a nested `CMakeLists.txt`
- Control blocks, loops and macros don't create a scope

Variable scopes (2)

- When a nested scope is created, CMake copies all the variables from the outer scope
- When the execution of the scope is complete, the copies are deleted, and outer scope is restored
- If a variable created in outer scope is unset(), it will disappear
- If this variable is referenced, CMake ignores the outer scope, but will search the cache

Variable scopes (3)

```
cmake_minimum_required(VERSION 3.26)
set(V 1)
message("> Global: ${V}")

block() # outer block
    message(" > Outer: ${V}")
    set(V 2)

    block() # inner block
        message(" > Inner: ${V}")
        set(V 3)
        message(" < Inner: ${V}")
    endblock()

    message(" < Outer: ${V}")
endblock()

message("< Global: ${V}")
```

```
> Global: 1
 > Outer: 1
 > Inner: 2
 < Inner: 3
 < Outer: 2
< Global: 1
```

Lists

- CMake concatenates all elements into a string using ; as a delimiter
- A list is created by calling the `set()` command
- Example
- CMake provides a `list()` command to read, search and modify lists
- Syntax: `list(COMMAND list_name additional_options)`
- Commands: `LENGTH`, `GET`, `JOIN`, `FIND`, etc.
- Cheatsheet

Creating lists

```
set(list1 A List With Several Elements)
set(list2 A;List;With;Several;Elements)
set(list3 A List "With;Several;Elements")

message("List1 is ${list1}")
message("List2 is ${list2}")
message("List3 is ${list3}")
```

```
List1 is A;List;With;Several;Elements
List2 is A;List;With;Several;Elements
List3 is A;List;With;Several;Elements
```

Manipulating lists

```
set(list1 A List With Several Elements)

list(LENGTH list1 len)
message("List1 length is ${len}")

list(FIND list1 With found)
message("The word With found at index ${found}")

list(REMOVE_ITEM list1 Several)
message("List1 after removal is ${list1}")
```

```
List1 length is 5
The word With found at index 2
List1 after removal is A;List;With;Elements
```

CMake control structures

Conditional blocks

- The only **control block** supported is `if()` command
- Syntax:

```
if(<condition>)
  <commands>
elseif(<condition>) # optional block, can be repeated
  <commands>
else()              # optional block
  <commands>
endif()
```

- No local variable scope is created in such block

Syntax for conditional commands

- Logical operators
  - `NOT <condition>`
  - `<condition1> AND <condition2>`
  - `<condition1> OR <condition2>`
  - Nesting is possible with matching parentheses
- Undefined variable will be evaluated to false
- Checking if a variable is defined:
  - `if (DEFINED VAR1), if (DEFINED CACHE{VAR1}), if (DEFINED ENV{VAR1})`
- Comparing values: `EQUAL`, `LESS`, `LESS_EQUAL`, `GREATER`, `GREATER_EQUAL`
- Software versions can be compared also
- Strings can be compared using STR prefix
- Regex is supported – `if (VAR MATCHES <regex>)`…

Example of cached variable

```
cmake_minimum_required(VERSION 3.28)
project(cached)

if (DEFINED VAR1)
    message("VAR1 is defined")
endif()

if (DEFINED CACHE{VAR1})
    message("VAR1 is defined in cache")
endif()

if (DEFINED ENV{VAR1})
    message("VAR1 is defined in cache")
endif()

set(VAR1 TRUE CACHE BOOL VAR1)
message("VAR1 is ${VAR1}")
```

```
-- The C compiler identification is GNU 14.2.0
-- The CXX compiler identification is GNU 14.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
VAR1 is TRUE
-- Configuring done (2.6s)
-- Generating done (0.0s)
```

```
VAR1 is defined
VAR1 is defined in cache
VAR1 is TRUE
-- Configuring done (0.0s)
-- Generating done (0.0s)
```

Simple checks

- Check if value is in list
- Check if a command is available
- Check if file exists
- Check if a path is directory
- Check if a path is symbolic link
- Check if path is absolute
- Reference

Examples

```
if (VAR)
    message("VAR is defined")
else()
    message("VAR is undefined")
endif()
```

```
VAR is undefined
```

```
set(VAR1 5)
set(VAR2 10)

if (VAR1 LESS VAR2)
    message("VAR1 is less than VAR2")
else()
    message("VAR2 is less than VAR1")
endif()

set(VER1 1.2.3)
set(VER2 1.1.4)

if (VER1 GREATER VER2)
    message("VER1 is greater than VER2")
else()
    message("VER2 is greater than VER1")
endif()

set(STR1 "HELLO1")
set(STR2 "HELLO2")

if (STR1 STRGREATER STR2)
message("STR1 is greater than STR2")
else()
    message("STR2 is greater than STR1")
endif()
```

```
VAR1 is less than VAR2
VER1 is greater than VER2
STR2 is greater than STR1
```

```
cmake_policy(SET CMP0057 NEW)

set(LIST1 A list with a several words)

if ("several" IN_LIST LIST1)
    message("Word several found in list!")
endif()

if (COMMAND set)
    message("Command set exists!")
endif()

if (EXISTS "05_conditions/03_checks.cmake")
    message("03_checks.cmake exists!")
endif()

if (IS_DIRECTORY "05_conditions")
    message("05_conditions is directory!")
endif()

if (IS_SYMLINK "05_conditions/03_checks.cmake")
    message("03_checks.cmake is a symlink!")
endif()

if (IS_ABSOLUTE "05_conditions")
    message("05_conditions is an absolute path!")
endif()
```

```
Word several found in list!
Command set exists!
03_checks.cmake exists!
05_conditions is directory!
```

Loops

Loops

- Two types of loop – while and foreach

```
while(<condition>)
  <commands>
endwhile()
```

```
foreach(<loop_var> RANGE <max>)
  <commands>
endforeach()
```

- Several other forms of foreach loop

```
foreach(<loop_var> RANGE <min> <max> [<step>])
```

```
foreach(<loop_variable> IN [LISTS <lists>] [ITEMS <items>])
```

- Loop controls – break() and continue()
- Zip lists

Loops – example (1)

```
set(COUNT 0)

while (COUNT LESS 10)
    math(EXPR COUNT "${COUNT} + 1")
    message("While loop - count: ${COUNT}")
endwhile()

set(COUNT 0)

foreach (COUNT RANGE 10)
    message("Foreach loop - count: ${COUNT}")
endforeach()
```

Loops – example (2)

```
foreach (COUNT RANGE 10 24 3)
    message("Foreach loop - count: ${COUNT}")
endforeach()

set(LIST_OF_WORDS1 A list of words)
set(LIST_OF_WORDS2 Another list with another words)

foreach (ENTRY IN LISTS LIST_OF_WORDS1 LIST_OF_WORDS2)
    message("Foreach loop - entry ${ENTRY}")
endforeach()

foreach (ENTRY IN LISTS LIST_OF_WORDS1 LIST_OF_WORDS2)
    message("Foreach loop - entry ${ENTRY}")
    if (ENTRY STREQUAL "with")
        break()
    endif()
endforeach()
```

Loops – example (3) - zip

```
set(STRINGS One Two Three Four Five)
set(NUMBERS 1 2 3 4 5)

foreach (ENTRY IN ZIP_LISTS STRINGS NUMBERS)
    message("Literal: ${ENTRY_0}, Number: ${ENTRY_1}")
endforeach()

foreach (WORD NUM IN ZIP_LISTS STRINGS NUMBERS)
    message("Literal: ${WORD}, Number: ${NUM}")
endforeach()
```

Macros and functions

Command definitions

- `macro()` or `function()`
- Like C or C++ - macro is a substitution, doesn't create an entry on a call stack
- Question – what happens if a return is called in a macro?
- `function()` creates a local scope
- Both `macro()` and `function()` offer the following variables
  - `${ARGC}` – Number of arguments
  - `${ARGV}` – All arguments as a list
  - `${ARGV<index>}` – The value of argument at a specific index
  - `${ARGN}` – A list of anonymous arguments that were passed by the caller after last expected argument

Macros (1)

- Macro definition

```
macro(<name> [<argument>…])
  <commands>
endmacro()
```

```
macro(VeryImportantMacro var)
    set(var "new value")
    message("argument: ${var}")
endmacro()

set(var "first value")
message("var is now: ${var}")
VeryImportantMacro("called value")
message("var is now: ${var}")
```

```
var is now: first value
argument: called value
var is now: new value
```

Macros (2)

- What happened in the example?
  - The variable was set but printed something else
  - That's because arguments passed to macros aren't treated as variables, but rather as find/replace instructions
  - However, the global scope variable was changed
  - This is a side effect and considered a bad practice (just like in C++)

```
macro(VeryImportantMacro var)
    set(var "new value")
    message("argument: ${var}")
endmacro()

set(var "first value")
message("var is now: ${var}")
VeryImportantMacro("called value")
message("var is now: ${var}")
```

Functions

- Function declaration

```
function(<name> [<argument>...])
  <commands>
endfunction()
```

- Functions do create their own variable scopes
- Any set() command will be local to the function
- CMake sets the following variables for each function
  - CMAKE_CURRENT_FUNCTION
  - CMAKE_CURRENT_FUNCTION_LIST_DIR
  - CMAKE_CURRENT_FUNCTION_LIST_FILE
  - CMAKE_CURRENT_FUNCTION_LIST_LINE

Functions - example

```
function(VeryImportantFunction arg)
    message("Function: ${CMAKE_CURRENT_FUNCTION}")
    message("File: ${CMAKE_CURRENT_FUNCTION_LIST_FILE}")
    message("arg: ${arg}")

    set(arg "new value")
    message("arg again: ${arg}")
    message("ARGV0: ${ARGV0} ARGV1: ${ARGV1} ARGC: ${ARGC}")
endfunction()


set(arg "first value")
VeryImportantFunction("Value1" "Value2")
message("arg in global scope: ${arg}")
```

```
arg: Value1
arg again: new value
ARGV0: Value1 ARGV1: Value2 ARGC: 2
arg in global scope: first value
```

Frequently used commands (1)

- `message()` – We've already seen it
  - May have `MODE` (like debug level). Default is `STATUS`. Reference
  - Useful debugging tweak – `CMAKE_MESSAGE_CONTEXT` (example)
  - Another one – `CMAKE_MESSAGE_INDENT` (example)
- `include()` – Includes another listfile

```
include(<file|module> [OPTIONAL] [RESULT_VARIABLE <var>])
```

  - If filename (`*.cmake`) is provided, CMake will try to open and execute it
  - No nested scope is created, any changes made to variables will be preserved
  - If file/module is not found, unless `OPTIONAL` was specified
  - If we need to know whether the include succeeded, we will provide `RESULT_VARIABLE` (Will be set to `NOTFOUND` if file/module wasn't found)
  - If no extensions is provided, CMake will try to find a module and include it

Frequently used commands (2)

- `include_guard()`
  - When we include files with side effects, we might want to restrict them to be included only once
  - Put the include_guard(DIRECTORY|GLOBAL) at the top of the include file
- `file()` – various file operations – `READ`, `WRITE`, `DOWNLOAD`
- `execute_process()` – As name implies – runs an external process

```
execute_process(COMMAND <cmd1> [<arguments>]... [OPTIONS])
```

operators won't work)

- Multiple commands can still be chained by providing `COMMAND` more than once

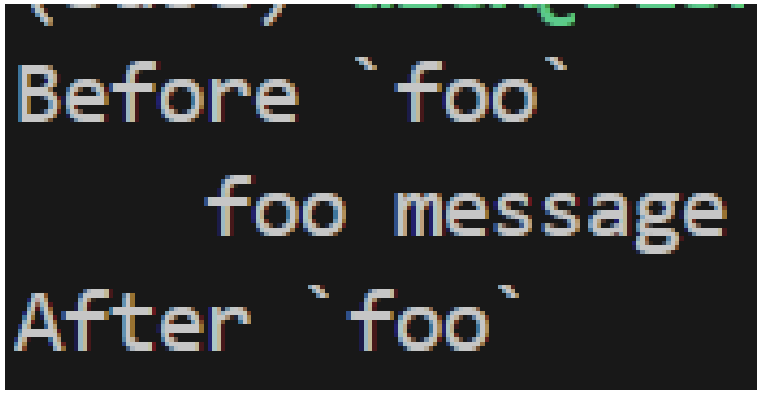
Examples (1)

```cmake
# Run with --log-context command line switch

function(inner_function)
    list(APPEND CMAKE_MESSAGE_CONTEXT ${CMAKE_CURRENT_FUNCTION})
    message("inner_function message")
endfunction()

function(outer_function)
    list(APPEND CMAKE_MESSAGE_CONTEXT ${CMAKE_CURRENT_FUNCTION})
    message("outer_function message")
    inner_function()
endfunction()

list(APPEND CMAKE_MESSAGE_CONTEXT "top")

message("Before `outer_function`")
outer_function()
message("After `outer_function`")
```

```
[top] Before `outer_function`
[top.outer_function] outer_function message
[top.outer_function.inner_function] inner_function message
[top] After `outer_function`
```

Examples (2)

```
function(foo)
    list(APPEND CMAKE_MESSAGE_INDENT "    ")
    message("foo message")
endfunction()

message("Before `foo`")
foo()
message("After `foo`")
```



Intermediate Summary

- We've covered CMake language basics
- Variables and lists
- Control structures (conditions and loops)
- Functions and macros

Setting up a CMake project

The core functionality of CMake

- **Building projects**
- A project contains all required source files and configuration
- **Configurations** includes, but not limited to:
  - Verifying that the target platform is supported
  - Ensuring the presence of all essential dependencies and tools
  - Confirming the compatibility of the provided compiler with the required features
- Generation of the **buildsystem** tailored to the selected build tool
- Execution of the **buildsystem** – compilation and linking

Basic CMake project - explained

```
cmake_minimum_required(VERSION 3.26)
project(Hello)
add_executable(Hello hello.cpp)
```

```
cmake -B <build tree> -S <source tree>
cmake --build <build tree>
```

Basic CMake project - explained

- `cmake_minimum_required()` – Must be at the top of the project files
  - Implicitly triggers cmake_policy(VERSION) which specifies the policies
  - Policies were introduced whenever there was a backward-incompatible change
- `project()` command should be run right after the former

```
project(<PROJECT-NAME> [<language-name>...])
```

```
project(<PROJECT-NAME>
        [VERSION <major>[.<minor>[.<patch>[.<tweak>]]]]
        [DESCRIPTION <project-description-string>]
        [HOMEPAGE_URL <url-string>]
        [LANGUAGES <language-name>...])
```

Partitioning

- Big project separated into "logic" units
- May be compiled into a single executable/library
- We may want to compile each logic unit (e.g. folder) separately, without a specific artifact
- We may want to compile each unit to a separate artifact

Partitioning the project (1) - Example

```
cmake_minimum_required(VERSION 3.28)

project(PaceMaker)

include(Data/data.cmake)
include(Control/control.cmake)

add_executable(PaceMaker Main.cpp
    ${control_sources} ${data_sources})
```

```
set(data_sources
    Data/Data.cpp
    Data/Data.h)
```

```
set(control_sources
    Control/Control.cpp
    Control/Control.h)
```

Partitioning the project (1) - Cons

- It works, but variables from inner scope pollute the top-level scope
- All subdirectories share the same configuration – everything is global, not granular enough
- Shared compilation triggers – any changes to configuration will cause full recompilation
- All the paths are relative to top-level – against DRY principle (rename directory?)

Partitioning the project (2) - Example

```
cmake_minimum_required(VERSION 3.28)

project(PaceMaker)

add_executable(PaceMaker Main.cpp)

add_subdirectory(Control)
add_subdirectory(Data)

target_link_libraries(PaceMaker PRIVATE Control Data)
```

```
add_library(Control OBJECT Control.cpp Control.h)

target_include_directories(Control PUBLIC .)
```

```
add_library(Data OBJECT Data.cpp Data.h)

target_include_directories(Data PUBLIC .)
```

Partitioning the project (2)

- Manage the scope with subdirectories
  - In the main **listfile** we use the `add_subdirectory()` command
  - We created **targets** – Control and Data using `add_library()`
  - The **`OBJECT`** parameter indicates that despite the `add_library()` command, we are only interested in generating object files
  - Grouped all source in a directory under logical **target**
  - We added the Control and Data directories to public include directories using `target_include_directories()` (similar to `-I`)

When to use?

- When working with multiple projects built using a single pipeline
- Example – SDK, set of libraries, frameworks
- Porting the buildsystem from a legacy solution – allows flexibility, breaking things down into a smaller pieces
- This is the standard way to manage project structure using CMake

Scoping the environment

- Multiple ways of querying the environment
- `CMAKE_` variables, `ENV` variables, special commands
- Can be used to support cross-platform scripts
- Allows to avoid using platform-specific shell commands
- OS type is stored in `CMAKE_SYSTEM_NAME` variable
  - Set when configuring a new build tree
  - OS names reference
- OS version is stored in `CMAKE_SYSTEM_VERSION` variable
- It is highly recommended to make solutions system-agnostic

Host system information

- `cmake_host_system_information(RESULT <var> QUERY <key>…)`
- Several interesting keys
  - `AVAILABLE_VIRTUAL_MEMORY`
  - `NUMBER_OF_LOGICAL_CORES/NUMBER_OF_PHYSICAL_CORES`
  - `IS_64BIT`
  - `HAS_SSE/HAS_SSE2/…`
- `CMAKE_SIZEOF_VOID_P` – 4 or 8
- `CMAKE_<LANG>_BYTE_ORDER - BIG_ENDIAN/LITTLE_ENDIAN`
  - LANG – C/CXX/OBJC/CUDA
- Full reference
- Example

Host system information - example

```
cmake_minimum_required(VERSION 3.28)

project(OSDetection)

message("Running on ${CMAKE_SYSTEM_NAME} OS, version ${CMAKE_SYSTEM_VERSION}")

add_executable(OSDetection Main.cpp)
```

```
-- Detecting CXX compile features - done
Running on Linux OS, version 6.6.87.2-microsoft-standard-WSL2
-- Configuring done (2.8s)
-- Generating done (0.0s)
-- Build files have been written to: /home/alex/cmake_course/build
```

Configuring the toolchain

Useful commands

- A toolchain consists of all the tools used in building and running the application
- Set the C++ standard globally – `set(CMAKE_CXX_STANDARD 23)`
- Can be overridden for a specific target using
  `set_property(TARGET <target> PROPERTY CXX_STANDARD 23)` or
  `set_target_properties(<target> PROPERTIES CXX_STANDARD 23)`

  Reference of available properties
- Enforce the standard using `set(CMAKE_CXX_STANDARD REQUIRED ON)`
- Set vendor specific extensions to off - `set(CMAKE_CXX_EXTENSIONS OFF)`

Useful commands – cont'd

- Check for supported compiler features

```
list(FIND CMAKE_CXX_COMPILE_FEATURES cxx_variable_templates result)
if(result EQUAL -1)
  message(FATAL_ERROR "Variable templates are required for compilation.")
endif()
```

- Checking for every single feature can be a daunting task
- Checking for certain meta-features is recommended
  - `cxx_std_98`, `cxx_std_11`, `cxx_std_14`, `cxx_std_17`, `cxx_std_20`, `cxx_std_23`, and `cxx_std_26`.
- Full list of features

Compiling and running a test file

- Sometimes we'll need to use a specific C++ feature
- We want to know whether the compiler supports that feature during the configuration stage
- CMake provides a way to test it – `try_compile()` and `try_run()`
- `try_run()` gives more freedom, but can't be used in cross-compilation environment

Example (1)

```cpp
#include <map>
#include <iostream>
#include <format>

int main()
{
    std::map<int, int> m { {1, 2}, {3, 4}, {5, 6} };

    for (const auto& [k, v] : m)
    {
        std::cout
            << std::format("Key {}, Value {}", k, v)
            << "\n";
    }

    return 0;
}
```

```cmake
cmake_minimum_required(VERSION 3.28)
project(try_compile_run)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

try_run(run_result compile_result
    ${CMAKE_BINARY_DIR}/test_output
    ${CMAKE_SOURCE_DIR}/Main.cpp
    RUN_OUTPUT_VARIABLE output)

message("compile_result: ${compile_result}")
message("run_result: ${run_result}")
message("output:\n" ${output})
```

Example (2)

```
cmake_minimum_required(VERSION 3.28)
project(try_compile_run)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

try_run(run_result compile_result
    ${CMAKE_BINARY_DIR}/test_output
    ${CMAKE_SOURCE_DIR}/Main.cpp
    RUN_OUTPUT_VARIABLE output)

message("compile_result: ${compile_result}")
message("run_result: ${run_result}")
message("output:\n" ${output})
```

```
-- The C compiler identification is GNU 14.2.0
-- The CXX compiler identification is GNU 14.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
compile_result: FALSE
run_result:
output:
```

```
compile_result: TRUE
run_result: 0
output:
Key 1 Value 2
Key 3 Value 4
Key 5 Value 6
```

Cross-compilation

- Compiling code on one machine to be executed on a different target platform
- Several variables are available for that purpose
  - `CMAKE_HOST_SYSTEM`
  - `CMAKE_HOST_SYSTEM_NAME`
  - `CMAKE_HOST_SYSTEM_PROCESSOR`
  - `CMAKE_HOST_SYSTEM_VERSION`
- Otherwise, all variables are referencing the target system

Toolchain file

- CMake can be invoked with a parameter that indicates the toolchain file
  - `cmake --toolchain /path/to/toolchain_file.cmake`
  - `cmake -DCMAKE_TOOLCHAIN_FILE=/path/to/toolchain_file.cmake`
- The toolchain file is a CMake script file that contains all necessary definitions for the build to succeed, such as compiler, linker, assembler names, path to sysroot, and more
- This can be handy when compiling for multiple platforms
- Tip – provide absolute path to the toolchain file
- Toolchains reference

Example

```
# arm_toolchain.cmake

set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(CMAKE_C_COMPILER arm-linux-gnueabi-gcc)
set(CMAKE_CXX_COMPILER arm-linux-gnueabi-g++)
set(CMAKE_AR arm-linux-gnueabi-ar)
set(CMAKE_RANLIB arm-linux-gnueabi-ranlib)
```

```
# x86_toolchain.cmake

set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR ${CMAKE_HOST_SYSTEM_PROCESSOR})

set(CMAKE_C_COMPILER gcc)
set(CMAKE_CXX_COMPILER g++)
set(CMAKE_AR ar)
set(CMAKE_RANLIB ranlib)
```

Exercise

Exercise objectives

- Source code is provided
- Complete the code for the top listfile
  - Set the standard to C++23
  - Add all needed subdirectories
- Create executables for x86 and ARM (using any method)
- Create both debug and release builds

CMake targets

What is it all about?

- We can write all our application in a single source code file
- In practice, things are not so simple, can be a lot of files
- We must introduce some sort of partitioning
- CMake **target** represents a logical unit that focuses on a specific objective
- **Targets** can have **dependencies** on other targets
- CMake takes care of proper build order, optimizing for parallel builds, etc.
- When a **target** is built, an **artifact** is generated that can be used by other build targets

The concept of a target

- Very similar to GNU Make
- A recipe to compile a set of files to another file
- CMake works on a higher level of abstraction
- Allows to skip the definitions of the intermediate steps
- No need to write explicit commands as in GNU Make
- Single command
  - `add_executable(target_name list_of_source_files)`
- Several more
  - `add_library()`
  - `add_custom_target()`

Defining executable target

```
add_executable(<name> [WIN32] [MACOSX_BUNDLE]
               [EXCLUDE_FROM_ALL]
               [source1] [source2 ...])
```

- `[WIN32]` and `[MACOS_BUNDLE]` are optional, won't show the default console window, and is expected to generate its own UI
- [EXCLUDE_FROM_ALL] will prevent the executable target from being built in a default build.
- To build the target, it should be built with `cmake --build –t <target>`
- Verifies that the sources are newer than the artifacts

Defining library target

```
add_library(<name> [STATIC | SHARED | MODULE]
            [EXCLUDE_FROM_ALL]
            [<source>...])
```

- `[EXCLUDE_FROM_ALL]` is the same as in executable target
- `[STATIC | SHARED | MODULE]` indicates what type of library is built
  - `STATIC` is for static (.a, .lib)
  - `SHARED` is for dynamic (.so, .dll)
  - `MODULE` is for a “plugin” that can be loaded dynamically using dlopen-like functionality
- Verifies that the sources are newer than the artifacts

Defining custom target

- These targets are a bit different
- Extend the default build functionality
  - Run sanitizers
  - Generate reports
  - Etc.
- Useful mostly in advanced projects

```
add_custom_target(Name [ALL] [COMMAND command2 [args2...] ...])
```

  `command2` is the actual shell command

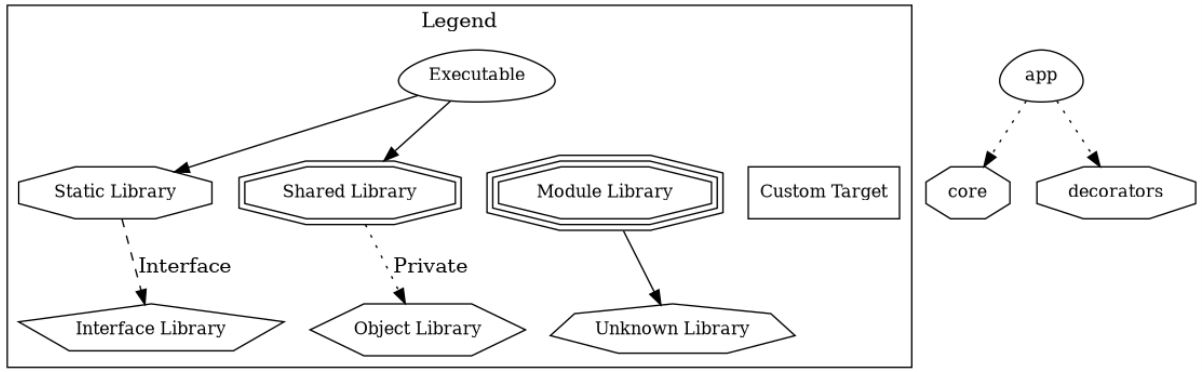
- Unlike executable and library, does not verify that the sources are newer because by default the artifacts are not added to the dependency graph
- Reference

Dependency graph

- Complex applications often built from many components
- Related things are packaged together in a single logical entity
- Can be linked with other targets
- Allows code reuse
- Example
  - Build the libraries
  - Build the app
  - Build the custom target – because of `add_dependencies`
  - The call to `add_dependencies` had to be added explicitly

Visualizing dependencies

- CMake provides a module to generate a dot file
- Run `cmake --graphviz=<filename.dot> <CMAKE_BINARY_DIR>`
- Install graphviz software (https://graphviz.org/download/)
- Alternatively, can be run online (https://dreampuf.github.io/GraphvizOnline)



Tweaking target properties

- Properties are somewhat like private members of C++ objects
- CMake defines a large list of available “known properties”
- New properties can be added

```
get_target_property(<var> <target> <property-name>)
set_target_properties(<target1> <target2> ...
                      PROPERTIES <prop1-name> <value1>
                      <prop2-name> <value2> ...)
```

- It is better to use higher-level commands
  - `add_dependencies()` is a shorthand of `MANUAL_ADDED_DEPENDENCIES` property

Transitive Usage Requirements

- Let's start from the middle – **usage**
  - CMake refers to dependency of one target on another as usage
- There are cases where used target sets properties or dependencies for itself, which, in turn, constitute **requirements** for other targets that use it – link a library, include a directory, etc.
- **Transitive** – some properties can transition (or propagate) across targets implicitly to ease expressing dependencies
- In simpler words – propagated properties between the source target (targets that get used) and destination targets (targets that use other targets)

Compile definitions

```
target_compile_definitions(<source> <INTERFACE|PUBLIC|PRIVATE>
[items1...])
```

- Like gcc –D flag
- The second argument specifies the propagation
  - PRIVATE sets the property of the source target
  - INTERFACE sets the property of the destination targets
  - PUBLIC sets the property of both source and destination targets

- Reference of `set_target_properties()`

Pseudo targets

- The concept of a target is so useful that it would be great if some of its behaviors could be borrowed
- Targets that do not represent output of the buildsystem, but rather inputs (to other projects)
  - Imported targets - CMake can define them as a result of find_package() command
  - Alias targets – Create another reference to a target under a different name

```
add_executable(<name> ALIAS <target>)
add_library(<name> ALIAS <target>)
```

  - Interface libraries – does not compile anything – used mostly for header-only libraries

```
add_library(Eigen INTERFACE
  src/eigen.h src/vector.h src/matrix.h
)
target_include_directories(Eigen INTERFACE
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/src>
  $<INSTALL_INTERFACE:include/Eigen>
)
```

C++ compilation using CMake

Introduction

- Usually, simple compilation scenarios are handled by default toolchain configuration
- In a professional setting, we'll need something more sophisticated
  - Optimizations – speed, size, etc.
  - Debugging capabilities
  - Multi-platform support
- The process of compilation is often not explained very well in C++ books, and the compilers documentation is a mess
- We'll try to fix it here

Compilation basics

- Translating source code into machine code
- C++ uses static compilation – entire program must be translated to a native code prior to execution
- How do we create a C++ program?
  - Design – planning functionality, structure, behavior
  - Compile individual .cpp (translation units) into object files
  - Link object files together into a single executable

Running an executable

- OS uses a tool called loader to map the program machine code into a virtual memory
- Then, the loader reads the program headers to determine where the execution should start and begins running the instructions
- Program start-up code executes - _start() function from libc
- Reads command-line arguments and environment variables
- Initiates threading, static symbols, etc.
- main() function is called

How compilation works?

- Generate a machine code – instructions that a processor can execute
- Object files are a direct translation of individual source files
- The following stages are required to create an object file
  - Preprocessing
  - Linguistic analysis
  - Assembly
  - Optimization
  - Code emission

Preprocessing

- Preparatory step prior to actual compilation
- Executes preprocessor directives (#include, #define, #if, etc.)
- Can be compared to an advanced find-and-replace tool

Linguistic analysis

- Scans the source file (including all the headers)
- Lexical analysis groups characters into meaningful tokens
  - Keywords
  - Variable names
  - Operators
- Syntax analysis - tokens are assembled into chains and examined to verify whether the chains adhere to syntax rules of C++
- Semantic analysis – checking whether the statements are logically correct. For example, assigning non-convertible types one to another

Assembly

- Translation of previously generated tokens into CPU instructions
- Some compilers generate an assembly source file, and then invoke the assembler with the generated file as input
- Other compilers keep the assembly code in memory and optionally dump it to disk
- Just because the code can be read, it doesn’t mean it should be ☺

Optimization

- Not a single step in the compilation process
- Occurs incrementally at each stage
- An important phase is minimizing register usage
- Inline expansion – the criteria for inlining is implementation-dependent
- Can enhance execution, but introduces issues with debugging

Code emission

- Writing the optimized machine code into an object file
- The format of the file aligns with the target platform spec.
- The object file is not ready for execution yet
- Should be passed to the linker
- The linker relocates the sections of object file and resolves references to external symbols
- Each of the described stages is significant and can be configured

Initial configuration

- CMake provides several commands that can affect each stage of the compilation
  - `target_compile_features()` – Setting specific compiler features
  - `target_sources()` – Adding sources to already defined target
  - `target_include_directories()` – Set include paths
  - `target_compile_definitions()` – Set preprocessor definitions
  - `target_compile_options()` – Compiler-specific command-line options
  - `target_precompile_headers()` – Set precompiled headers (pch)
- `Each of these commands accepts similar arguments`
  - `target_XXX(<target name> <INTERFACE|PUBLIC|PRIVATE> <arguments>)`

Compiler features

- Full reference to compiler features
- Unless there is a need for something very specific, it is recommended to pick a high-level meta-feature indicating the C++ standard:
  - `cxx_std_98`
  - `cxx_std_11`
  - `cxx_std_14`
  - `cxx_std_17`
  - `cxx_std_20`
  - `cxx_std_23`
  - `cxx_std_26`

Target source files (1)

- Source file can be added directly to `add_executable()/add_library()` commands
- The list of files can grow, and adding them to `add_library()` can become an exhausting process
- A possible solution:
  - ```
    file(GLOB tgt_SRC “*.h” “*.cpp”)
    add_executable(tgt ${tgt_SRC})
    ```
- Not recommended though
  - CMake generates buildsystems based on the changes in listfiles
  - If no changes are detected, build might fail without warning
  - Need to rerun the configuration stage
    ```

Target source files (2)

- Another question – how can we conditionally add sources?
- For example – We have a platform-specific implementation
- target_sources() command allows to append source files to an existing target

Target source files (3)

```
cmake_minimum_required(VERSION 3.28)
set(CMAKE_CXX_STANDARD 23)
project(target_sources)

add_executable(print_file Main.cpp)

if (CMAKE_SYSTEM_NAME STREQUAL "Windows")
    message("Build system is Windows, adding ReadFile_Windows.cpp")
    target_sources(print_file PRIVATE ReadFile_Windows.cpp)
elseif (CMAKE_SYSTEM_NAME STREQUAL "Linux")
    message("Build system is Linux, adding ReadFile_Linux.cpp")
    target_sources(print_file PRIVATE ReadFile_Linux.cpp)
endif()
```

Target source files (4)

- But how to add multiple files?
- No good answer yet
- Add manually
- Use variables as in previous slide and re-run configuration stage

Preprocessor configuration – cont'd

- Preprocessor definitions - `#define, #if, #elif,` etc.
- We can #define in our code, but what if we want it to be external?
- target_compile_definitions() is our savior
- What if there are too many definitions?
- CMake configure_file command to the resque!
- Generates new files from templates

Full configure_file() reference

Example

```
// Definitions.h.in

#cmakedefine ENABLER_FLAG
#cmakedefine STRING_VARIABLE1 "@STRING_VARIABLE1@"
#cmakedefine STRING_VARIABLE2 "${STRING_VARIABLE2}"
#cmakedefine UNDEFINED_VARIABLE "@UNDEFINED_VARIABLE@"
```

```
cmake_minimum_required(VERSION 3.28)
set(CMAKE_CXX_STANDARD 23)
project(configure)

add_executable(configure Main.cpp)

set(ENABLER_FLAG ON)
set(STRING_VARIABLE1 "STRING_VARIABLE1")
set(STRING_VARIABLE2 "STRING_VARIABLE2")

configure_file(Definitions.h.in Defs/Definitions.h)

target_include_directories(configure PRIVATE ${CMAKE_BINARY_DIR})
```

```
#define ENABLER_FLAG
#define STRING_VARIABLE1 "STRING_VARIABLE1"
#define STRING_VARIABLE2 "STRING_VARIABLE2"
/* #undef UNDEFINED_VARIABLE */
```

Preprocessor configuration

- The preprocessor is still a major player in C++ (unfortunately)
- Paths to included files (-I)

```
target_include_directories(<target> [SYSTEM] [AFTER|BEFORE]
                           <INTERFACE|PUBLIC|PRIVATE> [item1...]
                          [<INTERFACE|PUBLIC|PRIVATE> [item2...]
...])
```

- Modifies the `INCLUDE_DIRECTORIES` property
  - Appending if `AFTER` was specified
  - Prepending if `BEFORE` was specified
  - `SYSTEM` signifies to the compiler that the given directories should be treated like the system ones (<>)

Optimizer configuration

- Optimizer analyzes the output of previous stages and plays its tricks
- The code after optimizations is almost unrecognizable
- Can move code around, compact functions, and even wipe code
- Every compiler has its own unique tricks
- Many compilers won't enable optimizations by default
- We can re-enable it using `target_compile_options()`

```
target_compile_options(<target> [BEFORE]
                       <INTERFACE|PUBLIC|PRIVATE> [items1...]
                      [<INTERFACE|PUBLIC|PRIVATE> [items2...]
...])
```

Optimizer configuration - cont'd

- Optimization level - most compilers offer 4 basic levels (0-3)
- Usually a trade-off between optimization level and compilation time
- Also, sometimes there is an "optimize for size" option,
- CMake standardizes the experience for developers
- Default flags for compilers - stored in system-wide variables
  - `CMAKE_CXX_FLAGS_DEBUG` (`-g`)
  - `CMAKE_CXX_FLAGS_RELEASE` (`-O3 -DNDEBUG`)
- Can be easily overridden in several ways
- Each level sets a long list of flags, provided by compiler documentation
- Can be fine-tuned afterwards by adding explicit options such as `-finline-functions/-fno-inline-functions`

Inlining functions

- It has its advantages
  - No creation and destruction of a stack frame
  - No need to jump and return to the next instruction to execute
- Has also drawbacks
  - If used only once - should be copied to all call sites, resulting in a larger executable
  - Impacts debugging - inlined code is no longer at the original line number
- `-finline-functions-called-once` (GCC), `-finline-functions` (GCC/clang), `-finline-hint-functions` (clang)
- `-fno-inline` to disable inlining

Loop unrolling

- Another optimization technique
- Aims to transform loops to series of statements that achieve the same result
- Trade-off between program size and execution speed
- Effective only if the compiler is able to estimate the number of iterations
- Can lead to undesired consequences on modern CPUs as the increased code size can ruin effective caching
- `-floop-unroll` (GCC), `-funroll-loops` (clang)

Loop vectorization

- SIMD
- Effective when each loop iteration is not dependent on previous ones
- `-ftree-vectorize`, `-ftree-slp-vectorize` (GCC), enabled by default in clang
- `-fno-vectorize`, `-fno-slp-vectorize` - Disable vectorization (clang)
- The efficiency stems from the utilization of special CPU instructions rather than substituting the original for with unrolling

Managing the compilation process

Reducing compilation time

- When frequent recompilation is required (imagine altering a header included by all source files), we would like to make compilation time as short as possible
- C++ is already pretty good at managing compilation time (eh)
- Separate translation units
- CMake will recompile only the sources that were impacted by recent changes
- We still can improve things even more

Precompiled headers

- If a header changes, all sources should be recompiled
- Not good, but that's what we have (modules? where are you?)
- CMake offers a command to enable header precompilation
- The headers processed separately from the source

```
target_precompile_headers(<target>
                          <INTERFACE|PUBLIC|PRIVATE> [header1...]
                         [<INTERFACE|PUBLIC|PRIVATE> [header2...]
...])
```

- Stored in `PRECOMPILE_HEADERS` property
- Example
- Can be reused across targets

```
target_precompile_headers(<target> REUSE_FROM <other_target>)
```

Unity builds

- Also known as unified/jumbo builds
- A technique where a source includes other source (implicitly)
- The advantage is clear - less translation units
- Another advantage - the optimizer can act at a greater scale
- Increasing memory needed to process larger files
- Reduce potential parallelization
- Need to consider C++ implications - anonymous namespaces, static variables and functions, etc.
- Usually this technique is optimal when a full recompilation is needed
- Full reference

```
set_target_properties(<target1> <target2> ...
                      PROPERTIES UNITY_BUILD true)
```

Debugging the build

Configuring errors and warnings

- `-Werror` flag
  - Treats warnings as errors
  - Seems like a beneficial approach? Not always
  - Warnings are designed to caution you, and it's up to you to decide
  - Sometimes it can fail just because of the compiler upgrade
  - Usable when a public library is created - they can't fault your code if running in an environment stricter than yours
- `-Wpedantic` flag
  - Enables all warnings demanded by ISO C and C++ standards
  - Does not confirm conformance with the standard
- `-Wall` (`+ -Wextra`)
  - All levels of warnings

Debugging header files inclusion

- -H option can be added to the build
- It will print all included files (recursively)
- Can help when there are header files with similar names
- Example

Saving intermediate outputs to file

- GCC and clang only
- Creates .ii(after preprocessing) and .s (assembly) files
- May be helpful where there are problems with the preprocessor
- Example

Example

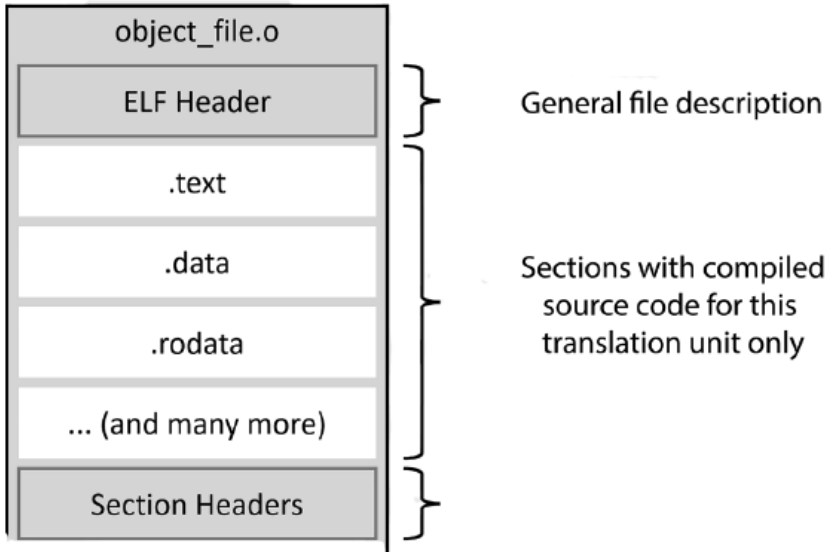
```
cmake_minimum_required(VERSION 3.28)
project(inclusion)
set(CMAKE_CXX_STANDARD 23)

add_executable(inclusion Main.cpp)
target_compile_options(inclusion PRIVATE -H)

add_executable(save_intermediate Main.cpp)
target_compile_options(save_intermediate PRIVATE -save-temps=obj)
```

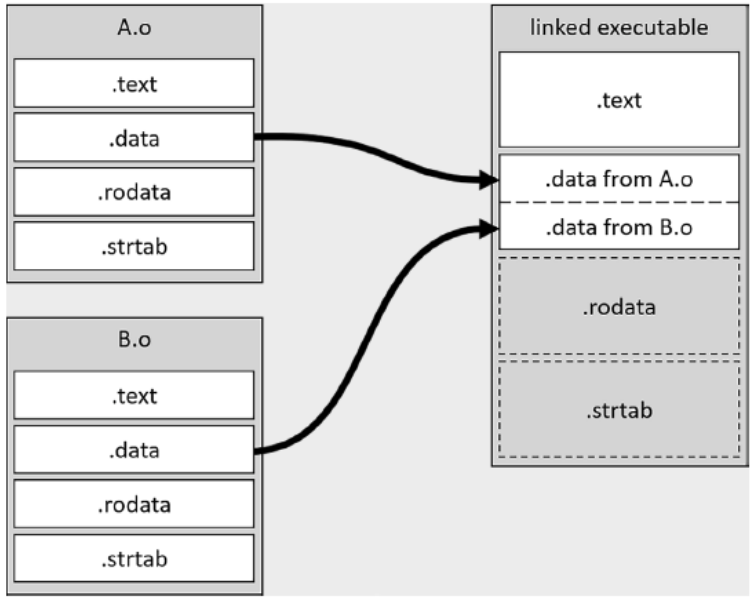
Linking executables and libraries

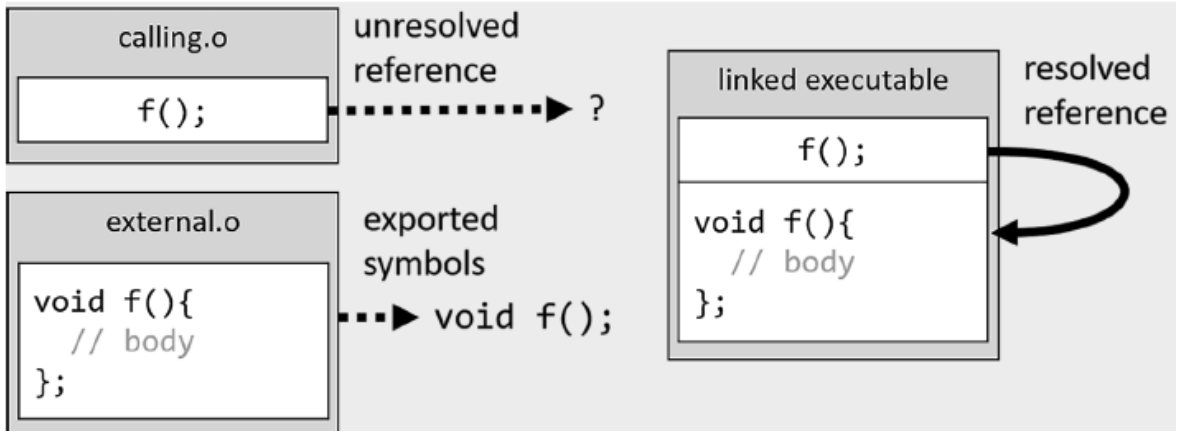
Linking basics



- Such an object is produced for every translation unit
- We can’t simply concatenate object files together and load the file
- Waste of extra memory and time if done without caution
- Hard to transfer data to CPU cache
- Each section is grouped with other sections of the same type
- This is called “relocation”
- The process also includes updating internal references such as addresses of functions, variables, etc.

Linking basics - cont'd

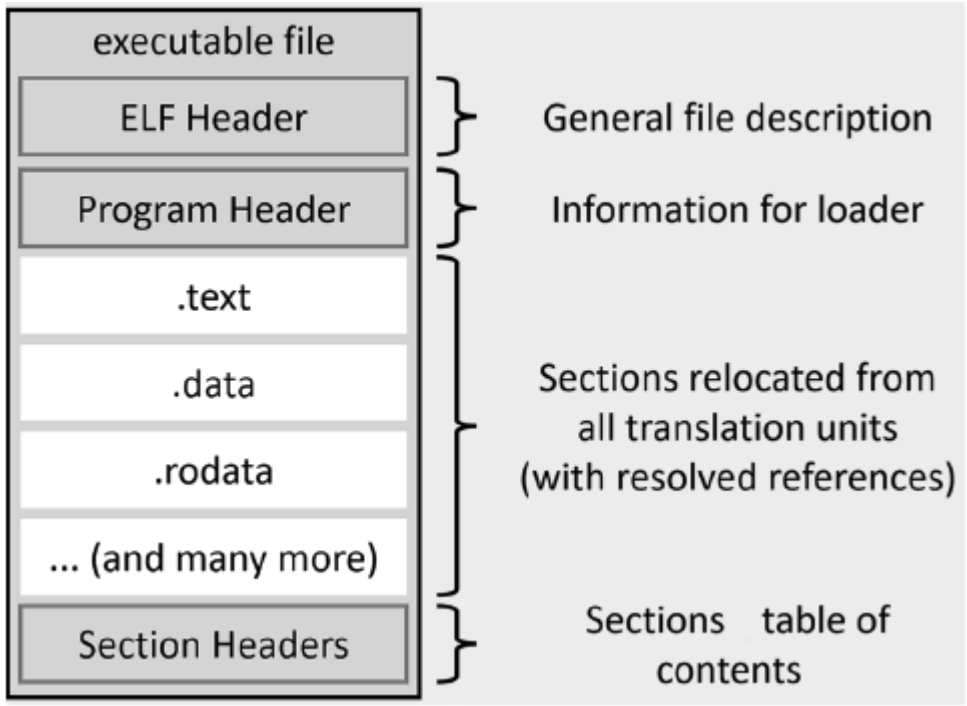




Relocating and resolving references

Linking basics - cont'd

Entries in program header specify which sections will be copied, in what order, and to which addresses in the virtual memory



Library types

- Adding a library target - `add_library()` command
- Linking a library to a target - `target_link_libraries()` command
- By convention, libraries have “lib” prefix
- **Static library** has .a extension (or .lib for Windows)
- **Dynamic library** has .so extension (or .dll for Windows)
- The process of creating libraries is called “**linking**”

Static libraries

- A collection of raw object files stored in an **archive**
- Can be extended with an index to speed up the linking process
- On Unix-like systems, created using the ar tool and indexed with ranlib
- Only necessary symbols are imported into the final executable
- Created using the following syntax
- STATIC can be omitted

```
add_library(<name> [<source>...])
```

```
add_library(<name> STATIC [<source>...])
```

Shared libraries

- Constructed using an actual linker
- Shared libraries can be utilized across multiple distinct apps
- When one program uses a shared library, the OS load one instance of the library to the memory
- Subsequent programs then provided with the same addresses, thanks to the virtual memory mechanisms
- For every process that uses the library, the .data and the .bss are instantiated separately
- The overall memory usage is better
- Build libraries using the following syntax

```
add_library(<name> SHARED [<source>...])
```

Shared libraries loading

- Shared libraries are loaded during program initialization
- There is no association between the program and the library file on the storage device
- The linking is done indirectly
- This is done via a shared object name (`SONAME`)
- Allows flexibility in library versioning
- Ensures that backward-compatible changes to the library don’t break dependent apps

Position-independent code (PIC)

- Compiling a library introduces uncertainty - it's unclear which process would use the library or where it will be located in the memory
- Therefore, we need another level of indirection
- PIC maps symbol files to their runtime addresses
- A new section in the binary file - Global Offset Table(GOT)
- During linking, a relative position of GOT to .text is calculated
- All symbols will be pointed through an offset to a placeholder in the GOT
- Shared libraries must be compiled with PIC flag activated
- CMake automatically enables it, however if the target depends on another target, the property must be applied to the dependent target

```
set_target_properties(dependency
                      PROPERTIES POSITION_INDEPENDENT_CODE ON)
```

Avoiding ODR issues

- In a scope of a single translation unit a symbol can be defined only once
- Even if the same name declared multiple times
- **Declaration** - “introduces” the symbol
- **Definition** - provides all details (function body, variable assignment)
- During linking, the rule is extended to the entire program
- Example
- Types, templates and extern inline functions can have repeated definitions, only if they are identical

ODR with shared libraries

- Example
- The order of linkage matters to the linker
- Can lead to confusion
- Naming collisions are not that uncommon
- Locally visible symbols take precedence over those from DLLs
- Pro tip - use namespaces!

Order of linking and unresolved symbols

- Sometimes the linker can throw weird errors seeming without cause - undefined reference to X
- Happens often when trying to integrate a new library
- Example - trying to reference a variable from libraryB
- Dependency chain: main → libraryA → libraryB
- We have an issue here, but why?
- The linker processes the binaries from left to right
  - Collects all undefined symbols exported and stores them for later
  - Tries to resolve undefined symbols (from all binaries processed)
  - Repeat for the next binary
- Cyclic references will be resolved by linking twice
  - `target_link_libraries(main B A B)`
- Can be solved using generator expressions (advanced topic)

Unreferenced symbols

- Libraries are archives of multiple object files bundled together
- Symbol indexes might be created to speed up the linkage
- Provide a mapping between symbol and the object file in which it is defined
- If no symbols from an object file are referenced, this object might be omitted
- There are scenarios where we might need the unreferenced symbols
  - Static initialization
  - Plugins (code that needs to be loaded at runtime)
  - Unused code in static libraries
  - Template instantiations
- Can be solved by whole-archive linking
  - `--whole-archive` (gcc)
  - `--force-load` (clang)
  - `/WHOLEARCHIVE` (msvc)

```
target_link_libraries(tgt INTERFACE
  "$<LINK_LIBRARY:WHOLE_ARCHIVE,lib1>"
)
```

Exercise

Exercise objectives

- `basicmath` folder should compile as archive (static lib)
- `advancedmath` folder should compile as shared object
- `app` should link only to `advancedmath`
- All listfiles are provided also, but try to write them by yourself and use them as reference only

Warm-up exercise

- git pull (or git clone https://github.com/alexkushnir/cmake_course)
- The exercise is in exercises/day_02_exercise_01 directory
- 2 C++ files (Main.cpp, BasicMath.cpp)
- 1 header file (BasicMath.h)
- 1 executable target (no cross-compilation)
- Empty CMakeLists.txt – complete it
- Relevant slides in the presentation: 76, 101, 122-125
- Optional:
  - Add C++ standard (as a good practice)
- 2 ways of solving the exercise
  - On docker (as on the first day)
  - Directly on the PC (assuming a Windows compiler is installed)

Open-source project CMakeLists.txt

- Zlib OS library
- Simplified version
- Several useful tricks we learned
  - Selecting default build type
  - Usage of CMake built-in modules
  - Read/write files and injecting configuration
  - Multi-platform compilation
  - Add multiple sources to a target
  - Adding subprojects

Managing 3rd party dependencies

Motivation

- CMake excels at handling various approaches to dependency management
- Can use already installed dependencies
- But can also fetch, build and install

Conan vs. CMake FetchContent module

Feature	Conan	FetchContent
Dependency Handling	Manages external libraries and dependencies	Downloads and integrates projects directly
Use Case	Best for public and mature dependencies Able to fetch binary artifacts	Ideal for internal or actively developed projects
Integration	Provides dependency info directly to CMake	Integrates with CMake build system
Cross-Compilation	Strong support for cross-compiling	Limited cross-compilation support

Dependencies installed in the system

Using already installed dependencies

- We need to find out where is the package in the file system
- Can do it manually, but every environment is different
- Need to find these paths automatically during the build process
- Several ways to do that
  - CMake built-in `find_package()` command
  - We can write our own find-module plugin
  - Obsolete – pkg-config (out of scope)
  - 3rd party package managers integration

find_package() (1)

- Let's suppose we have a code that uses a third-party library
- We have to install it prior to using it
- The OS package manager provides us this service (hopefully)
- Now what?
- If a package supports CMake (and a lot of them do), it provides a config file to allow CMake to discover it using find_package()
- If the library does not have a config file, CMake comes with over 180 modules that can find libraries
- Boost, curl, OpenSSL, zlib, X11, Qt, and many more

find_package() (2)

Basic format:

```
find_package(<Name> [version] [EXACT] [QUIET] [REQUIRED])
```

- Version can be a number (minimal), or a range
- EXACT (if a non-range was provided) - use exact version
- QUIET - Suppresses all messages whether the package was found
- REQUIRED - Will stop the build if a package is not found
- There are more options, reference

find_package() (3)

- CMake starts by checking its built-in find modules
- Then it moves on to checking the config files provided by different packages
- It scans paths where packages are usually installed
- Looks for patterns:
  - `<CamelCasePackageName>Config.cmake`
  - `<kebab-case-package-name>-config.cmake`
- Set `CMAKE_MODULE_PATH` variable to add external find modules location

find_package() (4)

- After find_package() finishes, it sets several variables, usually
  - `<PKG_NAME>_FOUND` - indicates whether the package was found
  - `<PKG_NAME>_INCLUDE_DIRS` or `<PKG_NAME>_INCLUDES`
  - `<PKG_NAME>_LIBRARIES` or `<PKG_NAME>_LIBS`
  - `<PKG_NAME>_DEFINITIONS`
- Can be found in the find_package() configuration file
- If the package is not found, but you are absolutely sure that it is present:
  - `cmake -S <source_dir> -B <build_dir> --debug-find-pkg=<pkg>`

find_package() - example (1)

Utilize Boost.Timer to measure loop execution time

```cpp
#include <boost/timer/timer.hpp>
#include <print>

int main()
{
    boost::timer::auto_cpu_timer t1;

    for (auto i = 0; i < 100000; ++i)
    {
        std::println("Index is {}", i);
    }

    return 0;
}
```

```
# Pre-requisite step: sudo apt-get install -y libboost-all-dev
cmake_minimum_required(VERSION 3.28)
project(find_package_example)
set(CMAKE_CXX_STANDARD 23)
set(Boost_VERBOSE)

find_package(Boost REQUIRED ALL)
if (Boost_FOUND)
    message("Boost_FOUND: ${Boost_FOUND}")
    message("Boost_INCLUDE_DIRS: ${Boost_INCLUDE_DIRS}")
    message("Boost_LIBRARIES: ${Boost_LIBRARIES}")
    add_executable(app Main.cpp)
    target_include_directories(app PRIVATE ${Boost_INCLUDE_DIRS})
    target_link_libraries(app Boost::timer)
endif()
```

find_package() - example (2)

```
(base) alex@DESKTOP-K5UVK7M:~/cmake_course$ cmake -S 22_find_package/ -B build
-- The C compiler identification is GNU 14.2.0
-- The CXX compiler identification is GNU 14.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found Boost: /usr/lib/x86_64-linux-gnu/cmake/Boost-1.83.0/BoostConfig.cmake (found version "1.83.0")
Boost_FOUND: TRUE
Boost_INCLUDE_DIRS: /usr/include
Boost_LIBRARIES: Boost::atomic;Boost::chrono;Boost::container;Boost::context;Boost::coroutine;Boost::date_time;Boost::exception;Boost::fiber;Boost::filesystem;Boost::graph;
Boost::graph_parallel;Boost::iostreams;Boost::json;Boost::locale;Boost::log;Boost::log_setup;Boost::math_c99;Boost::math_c99f;Boost::math_c99l;Boost::math_tr1;Boost::math_t
r1f;Boost::math_tr1l;Boost::mpi;Boost::mpi_python;Boost::nowide;Boost::numpy;Boost::prg_exec_monitor;Boost::program_options;Boost::python;Boost::random;Boost::regex;Boost::
serialization;Boost::stacktrace_addr2line;Boost::stacktrace_backtrace;Boost::stacktrace_basic;Boost::stacktrace_noop;Boost::system;Boost::test_exec_monitor;Boost::thread;Bo
ost::timer;Boost::type_erasure;Boost::unit_test_framework;Boost::url;Boost::wave;Boost::wserialization
-- Configuring done (5.9s)
-- Generating done (0.0s)
-- Build files have been written to: /home/alex/cmake_course/build
(base) alex@DESKTOP-K5UVK7M:~/cmake_course$
```

Writing a find module (1)

- Sometimes the package you want to use does not provide a config file
- We can write a custom find module and store it in our project
- Not ideal, but to avoid issues from your users side it is acceptable
- Example - a PostgreSQL client - libpqxx

Writing a find module (2)

```cpp
#include <pqxx/pqxx>
#include <iostream>

int main()
{
    std::cout << "libpqxx version: " <<
        PQXX_VERSION_MAJOR << "." <<
        PQXX_VERSION_MINOR << std::endl
    pqxx::connection c;

    return 0;
}
```

```cmake
# Pre-requisite step: sudo apt-get install -y libpqxx-dev
cmake_minimum_required(VERSION 3.28)
set(CMAKE_CXX_STANDARD 17)
# Because the libpqxx from apt was compiled with C++17 :(

project(find_module)

list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake/module/")
find_package(PQXX REQUIRED)

add_executable(app Main.cpp)
target_link_libraries(app PRIVATE PQXX::PQXX)
```

Writing a find module (3)

```
(base) alex@DESKTOP-K5UVK7M:~/cmake_course$ cmake -S 23_writing_find_module/ -B build
-- The C compiler identification is GNU 14.2.0
-- The CXX compiler identification is GNU 14.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found PostgreSQL: /usr/lib/x86_64-linux-gnu/libpq.so (found version "16.9")
-- Found PQXX: /lib/x86_64-linux-gnu/libpqxx.so
-- Configuring done (16.9s)
-- Generating done (0.0s)
-- Build files have been written to: /home/alex/cmake_course/build
```

Writing a find module (4)

- Guidelines to writing a find module
  - If the path to headers and libs are known (supplied by user or cached), use them to create `IMPORTED` target. Stop here
  - Find the library and headers for the underlying dependency (PostgreSQL)
  - Search the well-known paths to locate the binary client library
  - Similarly, scan the well-known paths to locate the headers
  - Confirm whether both the library and the headers are located. If found - create an `IMPORTED` target

Defining IMPORTED target

- We will need to define a library defined with the IMPORTED
- Will enable the caller to call target_link_libraries using target defined name (`PQXX::PQXX` in our case)
- The `UNKNOWN` is the library type, as we don't care what type of library was installed on the system
- Set `IMPORTED_LOCATION` and `INTERFACE_INCLUDE_DIRECTORIES` properties
- Store the found paths in cache variables (save the search for future runs)
- Mark them as advanced, to hide them in the CMake GUI

User-provided paths and reuse cached values

- Assume that the user installed pqxx in a non-standard location
- He can provide the path via CLI using -D flag
- This is the “easy case” - just define `IMPORTED` target
- We already know all the rest

Search for dependencies and library files

- PostgreSQL should be installed to utilize PQXX
- Luckily, it provides us with its' own find module, so we can use it
- Then, we'll use the `find_library()` built-in command
  - The command receives the filenames to look for and paths to look in
- The same goes for headers using `find_path()` built-in command
  - The command receives the filenames to look for and paths to look in

Returning the final results

- We will use find_package_handle_standard_args() function from the `FindPackageHandleStandardArgs` module
- It will set the `PQXX_FOUND` to 1 if the paths were correctly filled
- Can provide diagnostic messages (respects the `QUIET` keyword)
- Will halt execution if `REQUIRED` was specified
- Then, if `PQXX_FOUND` is set, we will call the `define_imported_target()` function
- PROFIT!

Dependencies not installed on the system

FetchContent (1)

- A user-friendly wrapper around `ExternalProject` module
- `FetchContent` brings dependencies in during configuration stage
- `ExternalProject` does it during build stage
- The targets defined by `FetchContent` will be in the same namespace, and can be linked to other targets
- There are some cases when this is not desirable, and then `ExternalProject` comes in hand

FetchContent (2)

- Management of directory structure for an external project
- Downloading the source code from URL (and extracting)
- Support for various sources (git, ftp, etc.)
- Fetching updates (if needed)
- Configuring and building projects with CMake, Make, or user-specified tool
- Providing nested dependencies on other targets

FetchContent (3)

- Practical recipe
  - Add the FetchContent module using include(FetchContent)
  - Configure the dependencies using FetchContent_Declare(). This will instruct FetchContent where the dependencies are and what version should be used
  - Complete the setup using FetchContent_MakeAvailable(). This will download, build, install and add the listfiles to the main project

FetchContent – example (1)

Example - Catch2 utilization

Note: There is no built-in way to show the targets the library exposes, only through reading the code/documentation

FetchContent – example (2)

```cpp
#include "catch2/catch_test_macros.hpp"

#include <print>

TEST_CASE("Dummy", "[Dummy]")
{
    REQUIRE(1 == 1);
}
```

```cmake
cmake_minimum_required(VERSION 3.28)
set(CMAKE_CXX_STANDARD 23)

project(catch2_example)
add_executable(catch2_example Main.cpp)

include(FetchContent)

FetchContent_Declare(catch2
    GIT_REPOSITORY https://github.com/catchorg/Catch2.git
    GIT_TAG v3.8.1
)

FetchContent_MakeAvailable(catch2)
target_link_libraries(catch2_example PRIVATE Catch2::Catch2WithMain)

include(CMakePrintHelpers)
cmake_print_properties(TARGETS Catch2::Catch2 PROPERTIES TYPE SOURCE_DIR)
```

Using already installed dependency

- CMake provides a feature that allows `FetchContent` to skip downloading if the dependencies are available locally

```
FetchContent_Declare(dependency-id
                     GIT_REPOSITORY <url>
                     GIT_TAG <tag>
                     FIND_PACKAGE_ARGS <args>
)
```

- Added `FIND_PACKAGE_ARGS` with `NAMES` argument to specify what we are looking for. Without names, CMake will default to dependency-id (user-defined name)

Example

```
cmake_minimum_required(VERSION 3.28)
set(CMAKE_CXX_STANDARD 23)

project(catch2_example)
add_executable(catch2_example Main.cpp)

include(FetchContent)

FetchContent_Declare(catch2
    GIT_REPOSITORY https://github.com/catchorg/Catch2.git
    GIT_TAG v3.8.1
    FIND_PACKAGE_ARGS NAMES Catch2
)

FetchContent_MakeAvailable(catch2)
target_link_libraries(catch2_example PRIVATE Catch2::Catch2WithMain)

include(CMakePrintHelpers)
cmake_print_properties(TARGETS Catch2::Catch2
    PROPERTIES TYPE SOURCE_DIR INTERFACE_INCLUDE_DIRECTORIES)
```

Example (2)

```
-- The C compiler identification is GNU 14.2.0
-- The CXX compiler identification is GNU 14.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
--
 Properties for TARGET Catch2::Catch2:
   Catch2::Catch2.TYPE = "STATIC_LIBRARY"
   Catch2::Catch2.SOURCE_DIR = "/home/alex/cmake_course/24_fetch_content/02_fetch_if_not_present"
   Catch2::Catch2.INTERFACE_INCLUDE_DIRECTORIES = "/usr/include"

-- Configuring done (2.5s)
-- Generating done (0.0s)
-- Build files have been written to: /home/alex/cmake_course/build
(base) alex@DESKTOP-K5UVK7M:~/cmake_course$
```

Exercise

Exercise objectives

- Utilize the FetchContent module
- Fetch and setup the fmt library
- Link the fmt library to the executable
- Fmtlib exported target is called fmt::fmt

ExternalProject

- Populates build dependencies during the build stage
- Cannot import targets into the project
- Can install dependencies directly into the system, and more
- More verbose method, FetchContent is more elegant
- FetchContent is suitable for small and quick to compile deps
- Sometimes (think Boost or Qt), the dependencies building is time-consuming
- If the package is already installed, this stage will not consume time

ExternalProject - example

```
cmake_minimum_required(VERSION 3.28)
set(CMAKE_CXX_STANDARD 23)

project(catch2_example)
add_executable(catch2_example Main.cpp)

configure_file(ExternalProjectCMakeLists.txt.in
    catch2-download/CMakeLists.txt)

execute_process(COMMAND "${CMAKE_COMMAND}" -G "${CMAKE_GENERATOR}" .
    WORKING_DIRECTORY "${CMAKE_BINARY_DIR}/catch2-download"
)
execute_process(COMMAND "${CMAKE_COMMAND}" --build .
    WORKING_DIRECTORY "${CMAKE_BINARY_DIR}/catch2-download"
)
add_subdirectory("${CMAKE_BINARY_DIR}/catch2-src"
                 "${CMAKE_BINARY_DIR}/catch2-build"
)

target_link_libraries(catch2_example
    PRIVATE Catch2::Catch2WithMain)

include(CMakePrintHelpers)
cmake_print_properties(TARGETS Catch2::Catch2
    PROPERTIES TYPE
    SOURCE_DIR INTERFACE_INCLUDE_DIRECTORIES)
```

```
cmake_minimum_required(VERSION 3.28)
project(Catch2Fetch)

include(ExternalProject)
ExternalProject_Add(Catch2
    GIT_REPOSITORY https://github.com/catchorg/Catch2.git
    GIT_TAG v3.8.1
    SOURCE_DIR        "${CMAKE_BINARY_DIR}/catch2-src"
    BINARY_DIR        "${CMAKE_BINARY_DIR}/catch2-build"
    INSTALL_COMMAND ""
    TEST_COMMAND ""
)
```

CMake and Conan package manager

- conan is popular open-source package manager for C++ libraries
- Helps manage dependencies, build from source or use precompiled binaries
- After running the conan client, it adds a preset file to the CMake project
- Then, the flow is as usual, a call to find_package() as if the package was already installed

Conan – example (1)

```
[requires]
nlohmann_json/3.12.0

[generators]
CMakeDeps
CMakeToolchain
```

```
======== Computing dependency graph ========
Graph root
    conanfile.txt: /home/alex/cmake_course/25_cmake_and_conan/01_simple_project/conanfile.txt
Requirements
    nlohmann_json/3.12.0#2d634ab0ec8d9f56353e5ccef6d6612c - Cache

======== Computing necessary packages ========
Requirements
    nlohmann_json/3.12.0#2d634ab0ec8d9f56353e5ccef6d6612c:da39a3ee5e6b4b0d3255bfef95601890afd80709#2ea4b814fc9da53f7cdf5b7d68052688 - Cache

======== Installing packages ========
nlohmann_json/3.12.0: Already installed! (1 of 1)

======== Finalizing install (deploy, generators) ========
conanfile.txt: Writing generators to /home/alex/cmake_course/25_cmake_and_conan/build
conanfile.txt: Generator 'CMakeDeps' calling 'generate()'
conanfile.txt: CMakeDeps necessary find_package() and targets for your CMakeLists.txt
    find_package(nlohmann_json)
    target_link_libraries(... nlohmann_json::nlohmann_json)
conanfile.txt: Generator 'CMakeToolchain' calling 'generate()'
conanfile.txt: CMakeToolchain generated: conan_toolchain.cmake
conanfile.txt: CMakeToolchain: Preset 'conan-release' added to CMakePresets.json.
    (cmake>=3.23) cmake --preset conan-release
    (cmake<3.23) cmake <path> -G "Unix Makefiles" -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake  -DCMAKE_POLICY_DEFAULT_CMP0091=NEW -DCMAKE_BUILD_TYPE=Release
conanfile.txt: CMakeToolchain generated: /home/alex/cmake_course/25_cmake_and_conan/build/CMakePresets.json
conanfile.txt: CMakeToolchain generated: /home/alex/cmake_course/25_cmake_and_conan/01_simple_project/CMakeUserPresets.json
conanfile.txt: Generating aggregated env files
conanfile.txt: Generated aggregated env files: ['conanbuild.sh', 'conanrun.sh']
Install finished successfully
```

Using conan – example (2)

```cpp
#include "nlohmann/json.hpp"
#include <fstream>
#include <iostream>

int main(int argc, char** argv)
{
    if (argc < 2)
    {
        std::cout << "Usage: json_parser <path_to_json>\n";
        return 1;
    }

    auto json_path = argv[1];

    nlohmann::json data = nlohmann::json::parse(
        std::ifstream{json_path});

    for (const auto& field : data.items())
    {
        std::cout << field.key() << " : " <<  field.value() << "\n";
    }

    return 0;
}
```

```
cmake_minimum_required(VERSION 3.15)

if (NOT CMAKE_TOOLCHAIN_FILE)
    message("CMAKE_TOOLCHAIN_FILE is not set!")
    set(CMAKE_TOOLCHAIN_FILE conan_toolchain.cmake)
endif()

if (NOT CMAKE_BUILD_TYPE)
    set(CMAKE_BUILD_TYPE Release)
endif()

project(json_parser)

find_package(nlohmann_json REQUIRED)

add_executable(json_parser Main.cpp)
target_link_libraries(json_parser
    PRIVATE nlohmann_json::nlohmann_json)

configure_file(${CMAKE_CURRENT_SOURCE_DIR}/example.json
    ${CMAKE_CURRENT_BINARY_DIR} COPYONLY)
```

Using conan – example (3)

```
(base) alex@DESKTOP-K5UVK7M:~/cmake_course/25_cmake_and_conan$ cmake -S 01_simple_project/ \
> -B build -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Release
-- Using Conan toolchain: /home/alex/cmake_course/25_cmake_and_conan/build/conan_toolchain.cmake
-- Conan toolchain: Defining architecture flag: -m64
-- Conan toolchain: C++ Standard 23 with extensions ON
-- The C compiler identification is GNU 14.2.0
-- The CXX compiler identification is GNU 14.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Conan: Target declared 'nlohmann_json::nlohmann_json'
-- Configuring done (2.7s)
-- Generating done (0.0s)
-- Build files have been written to: /home/alex/cmake_course/25_cmake_and_conan/build
```

Simplify the flow

- We are lazy, we want a single command! Not installing and the n building
- conan provides a CMake script that allows it in this repo
- Refer to develop2 branch (develop is legacy)
- One cmake file should be included and referred in the project
- CMake variable CMAKE_PROJECT_TOP_LEVEL_INCLUDES should be set to this script
- As the name implies, this are the first files that should be included in the listfile
- Intended for specifying one-time setup for the build (injection point)

Example (1)

- Let's check the build output

```
cmake_minimum_required(VERSION 3.28)
set(CMAKE_PROJECT_TOP_LEVEL_INCLUDES
    ${CMAKE_SOURCE_DIR}/conan_provider.cmake)
project(json_parser)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(nlohmann_json REQUIRED)
add_executable(json_parser Main.cpp)
target_link_libraries(json_parser
    PRIVATE nlohmann_json::nlohmann_json)
configure_file(
    ${CMAKE_CURRENT_SOURCE_DIR}/example.json
    ${CMAKE_CURRENT_BINARY_DIR} COPYONLY)
```

Exercise

Exercise objectives

- Source code is provided - basic file creation/modification monitor
- The code depends on a 3rd party library
  - `efsw` - cross-platform filesystem watcher (`efsw::efsw`)
- Install the dependencies using conan
- Run the code
- From another terminal, create, modify, delete file in /tmp
- Hit Ctrl + C to finish

Testing with CMake

Why testing at all?

- A belief - writing tests is a waste of time and effort
- “I compiled and run it; it has been tested”
- Time to move on to next task?
- Wrong
- Testing ensures that new changes don’t break old functionality
- Automated testing allows to cover more cases
- Allows regression testing on a large codebase in a short time
- CTest helps us to coordinate test execution (test manager)

Why automated tests? (1)

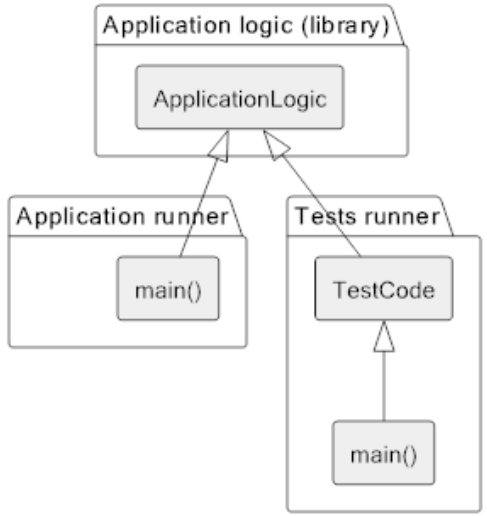
- It’s hard to predict which code will change frequently
- Software functionality constantly expands
- Hence, we must ensure we don’t break things
- Everyone makes mistakes, even the best ones
- Moreover, most of a programmer career he will work on code someone else wrote
- A rule of thumb - the programmer won’t understand the assumptions made on the code he works on
- Similar for the implications of the changes he makes to the code

Why automated tests? (2)

- Automated tests can prevent most of these issues
- Code snippets that verify whether another piece of code behaves correctly
- Run automatically, usually when someone makes a change
- Typically - as part of the build process, but can also run manually
- A common scenario is committing the change upon test suite success
- Gives the developer a level of confidence to change the code he is unfamiliar with

Structuring the project for testing

- How can we avoid multiple code compilations?
  - First time for the "real" code (code called from the main function)
  - Second time for the test code
- Tip – "divide and conquer"
  - Compile your code as a library
  - Link it to the unit tests
  - Create a bootstrap executable that links to the library and executes the code



CTest as test manager (2)

- For a new developer who is not familiar with the testing framework used in the project, it might be overwhelming
- Not always the testing conventions are documented (תושב"ע)
- What executable to run? What framework is in use? What arguments? How to collect results?
- CTest makes all these questions not relevant

CTest as test manager (3)

- CTest is a command line tool
- Configured by listfiles
- Offers a standardized way to run tests
- Applies to every project built with CMake
- Integrating with CI/CD becomes easier
- Robust test manager with minimal effort

Registering tests with ctest

- `ctest` needs to know how to run a specific test
- The test must be registered in CMakeLists.txt
- The `CTest` module must be included
- `add_test()` command from `CTest` module does the job
- The syntax is `add_test(NAME <test-name> COMMAND <test-command>)`
- If supported framework is used, tests can be discovered automatically (GoogleTest, Catch2)

The most basic unit test

```
project(calctester)

include(CTest)

add_executable(calctester Tester.cpp)
target_link_libraries(calctester calculator)
target_include_directories(calctester
    PRIVATE ${CMAKE_SOURCE_DIR}/src)

add_test(NAME "AddTest" COMMAND calctester a)
```

```
#include "Calculator.h"
#include <utility>
#include <cstdlib>
#include <cstring>

int main(int argc, char** argv)
{
    if (argc > 1 && std::strcmp(argv[1], "a") == 0)
    {
        float a = 6.3;
        float b = 2.4;
        auto expectedSum = a + b;
        auto sum = calculator::Add(a, b);
        if (expectedSum != sum)
        {
            std::exit(1);
        }
    }
}
```

Use CTest in a configured project

- 3 available modes
  - Dashboard
  - Test
  - Build-and-test
- The dashboard mode allows to send test results to a tool called CDash which collects and presents test results
- In test mode, the syntax is `ctest [<options>]` - **must** run in the build tree **after** the project was built with CMake.
- To make things easier, build-and-test mode allows to run the tests right after the build in a single command

Test mode

- Running ctest without arguments is sufficient in most cases
- If all test pass, ctest will return 0, which can be verified in the CI/CD pipeline
- Several issues might arise – timing, concurrency, deadlocks, etc.
- CTest offers various options to mitigate these issues
- Many options can be controlled – which tests to run, output they generate, etc (reference).

Handling failures

- Failing forward meaning learning from our mistakes
- This is exactly what we want when running tests
- CTest keeps things brief and only lists tests that failed
- Hard to see when and which test fails
- Can run with `--output-on-failure` for verbose output upon failure
- Can run with `--stop-on-failure` to stop execution upon failure
- Can provide `--rerun-failed` to skip the passed tests

Handling failures – example (1)

- Let’s run the tests together
- ctest --output-on-failure
- ctest --stop-on-failure
- ctest; ctest --rerun-failed

```cpp
#include "catch2/catch_test_macros.hpp"

#include <print>

TEST_CASE("Passed1", "[TestCase1]")
{
    REQUIRE(2 + 2 == 4);
}

TEST_CASE("Passed2", "[TestCase1]")
{
    REQUIRE(2 + 2 == 4);
}

TEST_CASE("Passed3", "[TestCase1]")
{
    REQUIRE(2 + 2 == 4);
}

TEST_CASE("Failed1", "[TestCase1]")
{
    REQUIRE(2 + 2 == 5);
}

TEST_CASE("Failed2", "[TestCase1]")
{
    REQUIRE(2 + 2 == 5);
}
```

Unit-testing frameworks

- Using test frameworks is easier than reinventing the wheel
  - Results reporting
  - Rich set of assertions
  - Less boilerplate code
- Zero cost of integrating the test code with CTest – just write the code according to framework rules
- Several frameworks like Catch2 and googletest even have built-in helper modules to write less CMake code

Unit test example using GoogleTest

- GoogleTest has a built-in CMake module
- Allows to discover tests automatically
- Include the `GoogleTest` module
- Add `gtest_discover_tests(<target>)` command

Example

```
set(PROJ_NAME "calctester")
project(${PROJ_NAME})

include(CTest)
include(FetchContent)

# Fetch googletest
FetchContent_Declare(googletest
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG main)

set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
set(BUILD_GMOCK OFF CACHE BOOL "" FORCE)
set(BUILD_GTEST ON CACHE BOOL "" FORCE)

FetchContent_MakeAvailable(googletest)

# Tests with googletest
add_executable(${PROJ_NAME} Tester.cpp)
    "${CMAKE_HOME_DIRECTORY}/src")
target_include_directories(${PROJ_NAME}
    PRIVATE "${googletest_BINARY_DIR}/include")
target_link_directories(${PROJ_NAME}
PRIVATE ${CMAKE_BINARY_DIR}/lib)
target_link_libraries(${PROJ_NAME} gtest gtest_main calculator)

include(GoogleTest)
gtest_discover_tests(${PROJ_NAME})
```

```
#include "Calculator.h"
#include "gtest/gtest.h"

TEST(AdditionTest, HappyPath)
{
    int a = 6;
    int b = 2;
    ASSERT_EQ(a + b, calculator::Add(a, b));
}

TEST(AdditionTest, FloatNumbers)
{
    float a = 6.3;
    float b = 2.4;
    ASSERT_EQ(a + b, calculator::Add(a, b));
}
```

Build-and-test mode

- `ctest --build-and-test <source-tree> <build tree> --build-generator <generator> [<options>] [--build-options <options>…] [--test-command <command>[<args>…]`
- Essentially, this is a wrapper around Test mode
- No test will be run unless the ctest keyword after --test-command is included
- Example: `ctest --build-and-test project/src project/build --build-generator "Ninja" --test-command ctest`
- Full build-options reference

Build-and-test example

- Demonstrates framework agnosticity
- Let’s go over the listfile in the IDE

Measuring coverage

Test coverage

- Challenge
  - Hard to track what parts of the code were covered by the tests
  - Tech debt evaluation (some people think so)
  - Requirements for code coverage in a regulated environment
- Solution (Linux)
  - LCOV – a graphical frontend for gcov
  - During tests run, coverage data is created
  - Metrics are collected into a dedicated file
  - A HTML report is generated
  - Note – the project should be compiled in debug

Setting up the report

- Add instrumentation to the binary (--coverage)
- Add a function that generates the report
- Add a custom target that calls the above function
- Build the target cmake --build <build-target> -t coverage
- Open the generated HTML report

CMake presets


They don't know
I wrote cmake preset

What are presets?

- A way to manage project settings
- Before presets, user had to memorize command line switches, or put configuration directly in listfiles
- Users can set up presets once and use whenever needed
- Help standardize settings across different users and computers
- Presets are compatible with the 4 modes of CMake – configure, build, test and package
- Workflows are supported also – combining all stages together

Using presets defined in a project

- We can create a preset file to store the required configuration
- CMakePresets.json - Official presets by the project authors
- CMakeUserPresets.json - Users can add custom presets. This file can be added to .gitignore or similar
- Must be placed in the project top directory
- Multiple presets can be defined for each stage
- We can show the available presets -
  cmake --<stage> --list-presets (--build/ --workflow)
- Similarly, for ctest and cpack

Writing a preset file

- The format for both files is similar
- The keys are:
  - version – Version of the JSON schema (required) Version guide
  - cmakeMinimumRequired – Object that specifies the CMake version
  - include – Array of strings that includes external presets
  - configurePresets – Array of objects that defines config stage presets
  - buildPresets – Array of objects that defines build stage presets
  - testPresets – Array of objects that are specific to test stage presets
  - packagePresets – Array of objects that are specific to pack stage presets
  - workflowPresets - Array of objects that are specific to workflow presets
  - vendor – Object for custom settings defined by the IDE or other vendor

Preset file example

- Only the version is mandatory
- No requirement to add empty arrays, can just omit it
- Run the preset:
  cmake --preset <name>

```
{
  "version": 6,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 26,
    "patch": 0
  },
  "include": [],
  "configurePresets": [],
  "buildPresets": [],
  "testPresets": [],
  "packagePresets": [],
  "workflowPresets": [],
  "vendor": {
    "data": "IDE-specific information"
  }
}
```

Common features across presets

- Unique name fields
  - Every preset must have a unique name within its stage
  - Beware, as the CMakeUserPresets implicitly includes the CMakePresets file
- Optional fields
  - displayName – a string representing a user-friendly name for the preset
  - description – a string explaining what the preset does
  - inherits – a string/array of strings that copies the config of presets as a base
  - hidden – a boolean that hides the preset from the listings. Can only be used through inheritance
  - environment – an object that overrides env vars for this stage
  - condition – an object that enables or disables the preset
  - vendor – custom object with vendor-specific values (same as root-level)
- All stage-specific presets must be linked with a configuration preset to know the location of the build tree
- CMake won't automatically execute the associated config stage preset

Defining config stage presets

- Suppose we have a preset named configPreset
- To configure the project using this preset, type cmake --preset configPreset
- The preset has some general fields but also has a unique set of optional fields
  - generator – a string that specifies a generator to use for the preset
  - architecture and toolset – a string that configures generators supporting these options
  - binaryDir – a string that provides a relative or absolute path to the build tree (supports macros)
  - installDir - a string that provides a relative or absolute path to the installation tree (supports macros)
  - cacheVariables – a map that defines cache variables (supports macros)

Configure stage preset example

```
{
  "version": 6,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 28,
    "patch": 0
  },
  "include": [],
  "configurePresets": [
    {
      "name": "configPreset",
      "displayName": "Configure Preset",
      "description": "Ninja generator",
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/../build",
      "installDir": "${sourceDir}/../build/install"
    }
  ]
}
```

```
cmake_minimum_required(VERSION 3.28)

project(configure_preset)

add_executable(configure_preset Main.cpp)
```

```
(base) alex@DESKTOP-K5UVK7M:~/cmake_course/37_presets/01_config_preset$ cmake --preset configPreset
Preset CMake variables:

  CMAKE_INSTALL_PREFIX:PATH="/home/alex/cmake_course/37_presets/build/install"

-- The C compiler identification is GNU 14.2.0
-- The CXX compiler identification is GNU 14.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (2.5s)
-- Generating done (0.0s)
-- Build files have been written to: /home/alex/cmake_course/37_presets/build
```

Defining build stage presets

- Suppose we have a preset named buildPreset
- To configure the project using this preset, type
  cmake --build --preset buildPreset
- The preset has some general fields but also has a unique set of optional fields
  - jobs – sets the number of parallel jobs to build the project
  - targets – string/string array that sets the targets to build (supports macros)
  - configuration – Debug/Release/etc.
  - cleanFirst – a boolean that indicates whether to perform a full build
- Example

Build stage preset example

```
{
  "version": 6,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 28,
    "patch": 0
  },
  "include": [],
  "configurePresets": [
    {
      "name": "configPreset",
      "displayName": "Configure Preset",
      "description": "Ninja generator",
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/../build",
      "installDir": "${sourceDir}/../build/install"
    }
  ],
  "buildPresets": [
    {
      "name": "buildPreset",
      "displayName": "Build Preset",
      "description": "Four jobs",
      "configurePreset": "configPreset",
      "jobs": 4
    }
  ]
}
```

A word about install preset

- There is no such thing...
- A neat workaround that allows to use the build preset to achieve the goal
- Add the "target" field with a value of install

Install preset - example

```json
{
  "version": 6,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 28,
    "patch": 0
  },
  "include": [],
  "configurePresets": [
    {
      "name": "configPreset",
      "displayName": "Configure Preset",
      "description": "Ninja generator",
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/../build",
      "installDir": "${sourceDir}/../build/install"
    }
  ],
  "buildPresets": [
    {
      "name": "buildPreset",
      "displayName": "Build Preset",
      "description": "Four jobs",
      "configurePreset": "configPreset",
      "jobs": 4
    },
    {
      "name": "installPreset",
      "displayName": "Installation",
      "targets" : "install",
      "configurePreset": "configPreset"
    }
  ]
}
```

```
(base) alex@DESKTOP-K5UVK7M:~/cmake_course/37_presets/03_install$ cmake --build --preset installPreset
[0/1] Install the project...
-- Install configuration: ""
-- Installing: /home/alex/cmake_course/37_presets/build/install/bin/configure_preset
```

Workflow presets

- The ultimate automation solution for any project
- Allow to execute multiple stage-specific presets in predetermined order
- End-to-end config-build-test-packaging in a single step
- Example
- Note – with `--fresh` flag, the entire build tree is regenerated

Workflow preset - example

```json
{
  "version": 6,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 28,
    "patch": 0
  },
  "include": [],
  "configurePresets": [
    {
      "name": "configPreset",
      "displayName": "Configure Preset",
      "description": "Ninja generator",
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/../build",
      "installDir": "${sourceDir}/../build/install"
    }
  ],
  "buildPresets": [
    {
      "name": "buildPreset",
      "displayName": "Build Preset",
      "description": "Four jobs",
      "configurePreset": "configPreset",
      "jobs": 4
    },
    {
      "name": "installPreset",
      "displayName": "Installation",
      "targets" : "install",
      "configurePreset": "configPreset"
    }
  ],
  "workflowPresets": [
    {
      "name": "workflowPreset",
      "steps": [
        {
          "type": "configure",
          "name": "configPreset"
        },
        {
          "type": "build",
          "name": "buildPreset"
        },
        {
          "type": "build",
          "name": "installPreset"
        }
      ]
    }
  ]
}
```

```
(base) alex@DESKTOP-K5UVK7M:~/cmake_course/37_presets/04_workflow$ cmake \
> --workflow --preset workflowPreset
Executing workflow step 1 of 3: configure preset "configPreset"

Preset CMake variables:

  CMAKE_INSTALL_PREFIX:PATH="/home/alex/cmake_course/37_presets/build/install"

-- The C compiler identification is GNU 14.2.0
-- The CXX compiler identification is GNU 14.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (2.1s)
-- Generating done (0.0s)
-- Build files have been written to: /home/alex/cmake_course/37_presets/build

Executing workflow step 2 of 3: build preset "buildPreset"

[2/2] Linking CXX executable configure_preset

Executing workflow step 3 of 3: build preset "installPreset"

[0/1] Install the project...
-- Install configuration: ""
-- Installing: /home/alex/cmake_course/37_presets/build/install/bin/configure_preset
```

Program analysis tools

What is it all about?

- We talked a lot about compiling and testing
- But what about readability?
- Coding convention?
- Modernization?
- Managing all this manually can be exhausting
- Instead of catching real issues in code review, the remarks are about style or readability
- Luckily, there are tools that can help
- Enforce formatting, static and dynamic analysis

Formatting enforcement

- A lot of options
  - Tabs vs. Spaces
  - Opening bracket location?
  - Snake case or hungarian notation?
- Doesn't change program behavior
- But surely can cause long argument based on personal opinions
- To ensure consistency, a tool like `clang-format` is recommended

Integrating with CMake

- Create a function that will run `clang-format`
  - Find the installed `clang-format` binary
  - Create a list of file extensions to format
  - Prepend each expression with a path to directory
  - Search recursively for sources and headers
  - Put found file paths into the `SOURCE_FILES` variable
  - Attach the formatting command to the `PRE_BUILD` step of target.
- Call the function from the main listfile
- For large projects we might need to convert absolute file paths to relative, due to character limitation in shell commands
- Optional – a .clang-format file can be added to dictate the exact formatting (reference)

Formatting example (1)

```
cmake_minimum_required(VERSION 3.28)
set(CMAKE_CXX_STANDARD 23)
project(Formatter)

list(APPEND CMAKE_MODULE_PATH
    ${CMAKE_SOURCE_DIR}/cmake)

add_executable(main Main.cpp)

include(Format)

Format(main .)
```

```
function(Format target directory)
    find_program(CLANG-FORMAT_PATH
        clang-format REQUIRED)
    set(EXPRESSION h hpp hh c cc cxx cpp)
    list(TRANSFORM EXPRESSION
        PREPEND "${directory}/*.")
    file(GLOB_RECURSE SOURCE_FILES
        FOLLOW_SYMLINKS LIST_DIRECTORIES
        false ${EXPRESSION})
    add_custom_command(TARGET ${target}
        PRE_BUILD COMMAND ${CLANG-FORMAT_PATH}
        -i --style=file ${SOURCE_FILES})
endfunction()
```

Formatting example (2)

Before

```
#include              <print>

using                                  namespace
std;

int main()
{println("Messy code!");return 0;}
```

After

```
#include <print>

using namespace std;

int main() {
    println("Messy code!");
    return 0;
}
```

Static analysis (1)

- Usually involves examining the source code without running it
- A lot of static checkers are available (OSS and proprietary)
- Many of the tools recognize CMake as industry standard and offer ready-to-use support or integration tutorials
- Some will prefer not to write CMake code, and use external modules - https://github.com/bilke/cmake-modules (cppcheck)
- A common belief is that setting up the checker is difficult
- Because checkers emulate the behavior of a real compiler
- It doesn't have to be so!

Static analysis (2)

- CMake allows to enable checkers for the following-tools
  - include-what-you-use (https://include-what-you-use.org/) - clang only
  - clang-tidy (https://clang.llvm.org/extra/clang-tidy)
  - Link What You Use (built-in CMake checker)
  - Cppcheck (https://cppcheck.sourceforge.io)
- To enable these, set a target property to a list containing the path to the checkers executables and their options
  - `<LANG>_CLANG_TIDY`
  - `<LANG>_CPPCHECK`
  - `<LANG>_CPPLINT`
  - `<LANG>_INCLUDE_WHAT_YOU_USE`
  - `LINK_WHAT_YOU_USE`

Static analysis – example (1)

```
struct S
{
    int i;
};

int main()
{
    char a[10];
    a[10] = 0;
    S* s = nullptr;

    s->i = 5;
    return 0;
}
```

```
cmake_minimum_required(VERSION 3.10)

project(CppCheckTest)

set(CMAKE_CXX_CPPCHECK cppcheck)
add_executable(CppCheckTest Main.cpp)

unset(CMAKE_CXX_CPPCHECK)
add_executable(main Main.cpp)
```

Static analysis – example (2)

```
Checking /home/alex/cmake_course/31_static_analysis/Main.cpp ...
/home/alex/cmake_course/31_static_analysis/Main.cpp:9:6: error: Array 'a[10]' accessed at index 10, which is out of bounds. [arrayIndexOutOfBounds]
    a[10] = 0;
     ^
/home/alex/cmake_course/31_static_analysis/Main.cpp:12:5: error: Null pointer dereference: s [nullPointer]
    s->i = 5;
    ^
/home/alex/cmake_course/31_static_analysis/Main.cpp:10:12: note: Assignment 's=nullptr', assigned value is 0
    S* s = nullptr;
           ^
/home/alex/cmake_course/31_static_analysis/Main.cpp:12:5: note: Null pointer dereference
    s->i = 5;
    ^
```

Installation and packaging

Are we there yet? PM asks

- We have built, tested and documented our project
- Time to release it to our users
- The 2 steps to take – installation and packaging
- Built on top of what we've learned so far
  - Managing targets
  - Managing targets dependencies
  - Transient usage requirements
  - Generator expressions
  - Etc...

Exporting without installation

- How can we make the targets of our project to be available to another project?
- `find_package()`! Oh, sh*t...we forgot to write it. No time to write it now
- Include the main listfile of our project in the consuming project?
  - Can work, but our project can include also some global configurations that we don't want to expose out of the project
- We can provide a target export file for the consuming project
- The consuming project can include it

Example

```cmake
cmake_minimum_required(VERSION 3.28)
set(CMAKE_CXX_STANDARD 23)
project(export_library)

set(EXPORT_DIR "${CMAKE_BINARY_DIR}/cmake")

add_library(static STATIC static_library.cpp)
target_sources(static
    PUBLIC FILE_SET HEADERS BASE_DIRS .
    FILES "static_library.h"
)

add_library(shared SHARED shared_library.cpp)
target_sources(shared
    PUBLIC FILE_SET HEADERS BASE_DIRS .
    FILES "shared_library.h"
)

export(TARGETS static shared
       FILE "${CMAKE_BINARY_DIR}/cmake/LibraryTargets.cmake"
       NAMESPACE library::
)
```

```cmake
cmake_minimum_required(VERSION 3.28)
set(CMAKE_CXX_STANDARD 23)
project(library_user)

include(${CMAKE_BINARY_DIR}/../library_build/cmake/LibraryTargets.cmake)

include(CMakePrintHelpers)
cmake_print_properties(TARGETS "library::static" PROPERTIES
  IMPORTED_CONFIGURATIONS
  INTERFACE_INCLUDE_DIRECTORIES
)

cmake_print_properties(TARGETS "library::shared" PROPERTIES
  IMPORTED_CONFIGURATIONS
  INTERFACE_INCLUDE_DIRECTORIES
)

add_executable(main_static MainStatic.cpp)
target_link_libraries(main_static PRIVATE library::static)

add_executable(main_shared MainShared.cpp)
target_link_libraries(main_shared PRIVATE library::shared)
```

A bit awkward, isn't it?

Installing projects in the system

- Luckily, CMake offers us a command-line mode for installation
  - `cmake --install <dir> [<options>]`
- Various set of options
- Installation typically involves copying generated artifacts and dependencies to a system directory
- Using CMake for installation offers the following benefits:
  - Provides platform-specific installation paths depending on type
  - Enhances the installation process by generating target export files
  - Creates discoverable packages through config files, wrapping the target export files and package-specific macros and functions

The install() command

- The first step of the installation process is copying artifacts to destination directory
- The install command can be run in several modes
  - `TARGETS` – installs output artifacts (executables/libraries)
  - `FILES | PROGRAMS` – installs other files and controls permissions
  - `DIRECTORY` – installs entire directory
  - `SCRIPT | CODE` – runs a CMake script during installation
  - `EXPORT` – generates and installs a target export file
  - `RUNTIME_DEPENDENCY_SET` – installs runtime dependencies
  - `IMPORTED_RUNTIME_ARTIFACTS` – queries imported targets for runtime artifacts and installs them
- Adding these commands generates a `cmake_install.cmake` file
- Not recommended to run with `cmake -P`, better run `cmake --install`
- Example

Installation example

- We have a library that we want to install and an executable that we want to link to the library

```
cmake_minimum_required(VERSION 3.28)
set(CMAKE_CXX_STANDARD 23)
project(install_library)

add_library(static STATIC static_library.cpp)
target_sources(static
    PUBLIC FILE_SET HEADERS BASE_DIRS include
    FILES "include/staticlib/static_library.h"
)

include(GNUInstallDirs)
install(TARGETS static ARCHIVE FILE_SET HEADERS)
```

```
(base) alex@DESKTOP-K5UVK7M:~/cmake_course$ sudo cmake --install build
[sudo] password for alex:
-- Install configuration: ""
-- Installing: /usr/local/lib/libstatic.a
-- Installing: /usr/local/include/staticlib/static_library.h
(base) alex@DESKTOP-K5UVK7M:~/cmake_course$
```

A word about public headers

- Best practice is to store in a directory that indicates their origin (`/usr/local/your-library/your-header.h`)
- Enables to write `#include <your-library/your-header.h>`
- We can use the `GNUInstallDirs` module to populate the `DESTINATION` part of the installation path
- We can add headers to be exposed to the appropriate target using the `target_sources` command

```
target_sources(<target>
  [<PUBLIC|PRIVATE|INTERFACE>
   [FILE_SET <name> TYPE <type> [BASE_DIR <dir>] FILES]
   <files>...
  ]...
)
```

Running scripts during installation

- Example - instructing the dynamic linker to rebuild its cache (ldconfig)

```
# ldconfig -p | grep libshared
cmake_minimum_required(VERSION 3.28)
set(CMAKE_CXX_STANDARD 23)
project(export_library)

add_library(shared SHARED shared_library.cpp)
target_sources(shared
    PUBLIC FILE_SET HEADERS BASE_DIRS .
    FILES "shared_library.h"
)

include(GNUInstallDirs)
install(TARGETS shared
        FILE_SET LIBRARIES DESTINATION "lib"
        FILE_SET HEADERS DESTINATION "include"
)

if (UNIX)
    install(CODE
    "execute_process(COMMAND ldconfig)")
endif()
```

```
(base) alex@DESKTOP-K5UVK7M:~/cmake_course$ sudo cmake --install build
-- Install configuration: ""
-- Installing: /usr/local/lib/libshared.so
-- Installing: /usr/local/include/shared_library.h
```

```
(base) alex@DESKTOP-K5UVK7M:~/cmake_course$ ldconfig -p | grep libshared
        libshared.so (libc6,x86-64) => /usr/local/lib/libshared.so
```

Creating reusable packages

- We have used `find_package()` before
- A convenient way to consume 3rd party libraries
- How can we publish our package?
  - Make targets relocatable
  - Install the target export file to a standard location
  - Create a config file for the package
  - Generate a version file for the package

Relocatable targets

- Since the path in `target_include_directories()` is relative, the property is implicitly prepended with `CMAKE_CURRENT_SOURCE_DIR` variable
- After the installation, the project must not rely on files from source or build trees
- Hence, the target is not suitable for reuse
- Its include directory still points to its source tree

Solving the issue

- Generator expressions (advanced topic)
- `$<BUILD_INTERFACE:…>` - evaluates … for regular builds, but excludes it for installation
- `$<INSTALL_INTERFACE:…>` - evaluates … for installation, but excludes for regular build
- `$<BUILD_LOCAL_INTERFACE:…>` - evaluates … when used by another target in the same buildsystem

- The generators are mutually exclusive, meaning only one will be used in the final step

Relocatable targets - example

```
cmake_minimum_required(VERSION 3.28)
set(CMAKE_CXX_STANDARD 23)
project(relocatable_target)

add_library(static STATIC static_library.cpp)
target_include_directories(static INTERFACE
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
  $<INSTALL_INTERFACE:${CMAKE_INSTALL_INCLUDEDIR}>
)

set_target_properties(static PROPERTIES
  PUBLIC_HEADER include/staticlib/static_library.h)
```

Installing target export files

- Creates and installs a target export file for an export that must be defined with install(`TARGETS`) command
- Will install a `.cmake` file which can be later included by the consuming target
- This is not a config file, and the consumer can't consume the package using `find_package()` yet
- However it is possible to include these files directly if necessary

Install target export files – example (1)

```
cmake_minimum_required(VERSION 3.28)
set(CMAKE_CXX_STANDARD 23)
project(relocatable_target)

add_library(static STATIC static_library.cpp)
target_sources(static
  PUBLIC FILE_SET HEADERS BASE_DIRS include
  FILES include/staticlib/static_library.h
)

set_target_properties(static PROPERTIES P
UBLIC_HEADER include/staticlib/static_library.h)

include(GNUInstallDirs)
install(TARGETS static EXPORT
  StaticLibTarget ARCHIVE FILE_SET HEADERS)

install(EXPORT StaticLibTarget
  DESTINATION ${CMAKE_INSTALL_LIBDIR}/static/cmake
  NAMESPACE Library::
)

install(FILES StaticLibConfig.cmake
  DESTINATION ${CMAKE_INSTALL_LIBDIR}/static/cmake
)
```

```
-- Install configuration: ""
-- Installing: /usr/local/lib/libstatic.a
-- Installing: /usr/local/include/static_library.h
-- Up-to-date: /usr/local/include/staticlib/static_library.h
-- Installing: /usr/local/lib/static/cmake/StaticLibTarget.cmake
-- Installing: /usr/local/lib/static/cmake/StaticLibTarget-noconfig.cmake
-- Installing: /usr/local/lib/static/cmake/StaticLibConfig.cmake
```

Install target export files – example (2)

```
# cmake -S <source-dir> -B <build-dir> -DCMAKE_PREFIX_PATH=/usr/local/lib/static
cmake_minimum_required(VERSION 3.28)
project(FindStaticLibPackage)

find_package(StaticLib REQUIRED)

include(CMakePrintHelpers)
message("CMAKE_PREFIX_PATH: ${CMAKE_PREFIX_PATH}")
message("StaticLib_FOUND: ${StaticLib_FOUND}")

cmake_print_properties(TARGETS "Library::static" PROPERTIES
    IMPORTED_CONFIGURATIONS
    INTERFACE_INCLUDE_DIRECTORIES
)
```

```
-- Detecting CXX compile features - done
CMAKE_PREFIX_PATH: /usr/local/lib/static
StaticLib_FOUND: 1
--
 Properties for TARGET Library::static:
   Library::static.IMPORTED_CONFIGURATIONS = "NOCONFIG"
   Library::static.INTERFACE_INCLUDE_DIRECTORIES = "$<BUILD_INTERFACE:/usr/local/include>"

-- Configuring done (2.5s)
-- Generating done (0.0s)
-- Build files have been written to: /home/alex/cmake_course/exe_build
```

Writing a basic config file

- Technically, all `find_package()` needs is a config file
- Optionally, also version and target exports files
- Acts as a package definition, providing package functions, etc.
- CMake searches for config files in default and in user-provided locations
- Searches for `<PackageName>-config.cmake` or `<PackageName>Config.cmake`
- The simplest config file only includes the target export file
- For more advanced options – reference

Generating package version files

- CMake searches for `<config-file>-version.cmake` or `<config-file>Version.cmake`
- This file specifies compatibility with other versions
- A specific version or a range can be requested by find_package:
  - `find_package(static 1.2.3 REQUIRED)`
  - `find_package(static 1.2.3...4.5.6 REQUIRED)`
- CMake provides a built-in module `CMakePackageConfigHelpers` to ease the task of writing such files

Generating package version files

```
write_basic_package_version_file(
  <filename> [VERSION <ver>]
  COMPATIBILITY <AnyNewerVersion | SameMajorVersion |
                 SameMinorVersion | ExactVersion>
  [ARCH_INDEPENDENT]
)
```

- Filename should follow the rules defined on previous slide
- Version is optional, if not specified, the version specified in project() will be used. If not specified, error will occur
- COMPATIBILITY defines the behavior of matching versions
  - ExactVersion does not support ranges
  - AnyNewerVersion matches any **older** version (surprise!)
- If the package is arch-agnostic, ARCH_INDEPENDENT can be specified

Managing links for versioned shared libs

- Done by install(TARGET <target> LIBRARY)
- First need to set the target property SOVERSION
- Example

Packaging with CPack

- Now, after configuring automatic installation, how to distribute?
- CMake provides a tool called CPack that allows to package the artifacts into a well-defined package format
- Several package generators are available
  - Archives (7z, tgz, zip, etc.)
  - Bundle (macOS)
  - DEB (Debian packages)
  - External (json files for 3rd party packagers)
  - FreeBSD (pkg)
  - Nuget
  - Wix
  - And more
- Each one of the generators has extensive configuration (out of scope)

Creating a package with CPack

- After running and building the project, just navigate to the build tree and run cpack [options]
- Example

CPack example

```
cmake_minimum_required(VERSION 3.28)
project(cpack_packaging VERSION 7.8.9)
set(CMAKE_CXX_STANDARD 23)

add_library(static STATIC static_library.cpp)
target_sources(static PUBLIC FILE_SET HEADERS
                             BASE_DIRS include
                             FILES include/staticlib/static_library.
)

include(GNUInstallDirs)
install(TARGETS static EXPORT StaticLibTarget ARCHIVE FILE_SET HEADER
install(EXPORT StaticLibTarget
  DESTINATION ${CMAKE_INSTALL_LIBDIR}/static/cmake
  NAMESPACE Library::
)
install(FILES "StaticLibConfig.cmake"
  "${CMAKE_CURRENT_BINARY_DIR}/StaticLibConfigVersion.cmake
  DESTINATION ${CMAKE_INSTALL_LIBDIR}/static/cmake
)
set(CPACK_PACKAGE_VENDOR "Alex Kushnir")
set(CPACK_PACKAGE_CONTACT "kushnir.alexander@gmail.com"
set(CPACK_PACKAGE_DESCRIPTION "Dummy static library")
include(CPack)
```

```
set(CPACK_BINARY_DEB "ON")
set(CPACK_BINARY_FREEBSD "OFF")
set(CPACK_BINARY_IFW "OFF")
set(CPACK_BINARY_NSIS "OFF")
set(CPACK_BINARY_RPM "OFF")
set(CPACK_BINARY_STGZ "ON")
set(CPACK_BINARY_TBZ2 "OFF")
set(CPACK_BINARY_TGZ "ON")
set(CPACK_BINARY_TXZ "OFF")
set(CPACK_BINARY_TZ "ON")
set(CPACK_BUILD_SOURCE_DIRS "/home/alex/cmake_course/36_cpack;/home/alex/cmake_course/build")
set(CPACK_CMAKE_GENERATOR "Unix Makefiles")
set(CPACK_COMPONENT_UNSPECIFIED_HIDDEN "TRUE")
set(CPACK_COMPONENT_UNSPECIFIED_REQUIRED "TRUE")
set(CPACK_DEFAULT_PACKAGE_DESCRIPTION_FILE "/usr/share/cmake-3.28/Templates/CPack.GenericDescription.txt")
set(CPACK_DEFAULT_PACKAGE_DESCRIPTION_SUMMARY "cpack_packaging built using CMake")
set(CPACK_GENERATOR "STGZ;TGZ;TZ")
set(CPACK_INNOSETUP_ARCHITECTURE "x64")
set(CPACK_INSTALL_CMAKE_PROJECTS "/home/alex/cmake_course/build;cpack_packaging;ALL;/")
set(CPACK_INSTALL_PREFIX "/usr/local")
set(CPACK_MODULE_PATH "")
set(CPACK_NSIS_DISPLAY_NAME "cpack_packaging 7.8.9")
set(CPACK_NSIS_INSTALLER_ICON_CODE "")
set(CPACK_NSIS_INSTALLER_MUI_ICON_CODE "")
set(CPACK_NSIS_INSTALL_ROOT "$PROGRAMFILES")
set(CPACK_NSIS_PACKAGE_NAME "cpack_packaging 7.8.9")
set(CPACK_NSIS_UNINSTALL_NAME "Uninstall")
set(CPACK_OBJCOPY_EXECUTABLE "/usr/bin/objcopy")
set(CPACK_OBJDUMP_EXECUTABLE "/usr/bin/objdump")
set(CPACK_OUTPUT_CONFIG_FILE "/home/alex/cmake_course/build/CPackConfig.cmake")
set(CPACK_PACKAGE_CONTACT "kushnir.alexander@gmail.com")
set(CPACK_PACKAGE_DEFAULT_LOCATION "/")
set(CPACK_PACKAGE_DESCRIPTION "Dummy static library")
set(CPACK_PACKAGE_DESCRIPTION_FILE "/usr/share/cmake-3.28/Templates/CPack.GenericDescription.txt")
```

Documentation

What we will do?

- Doxygen is de-facto standard for documenting C and C++ code
- Can provide plenty of formats – HTML, PDF, man pages and more
- Can also produce charts and diagrams
- The code needs to be annotated in a very specific way
- Steps to generate the documentation
  - Add Doxygen to the project
  - Generate documentation
  - Enhance the output by decorating the HTML

Make the output look nicer

- Based on open-source CSS-based theme
- Look-and-feel is customized by CSS variables (reference)
- Several other documentation generation utilities
  - Adobe Hyde - clang based, produces markdown, mostly for tools like Jekyll and Zola
  - Standardese – uses libclang to compile. Aims to be the next Doxygen

Example

- Let's go to the IDE

Any questions before we conclude?