Systems Programming Languages

Igor Dimitrov

2024-05-17

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1 Project Setup

Below is a comprehensive summary of the C++ study build setup we designed.

1.1 C++ study build system summary

1.1.1 1. Overall repository structure

Your C++ study code lives inside a larger Quarto book/repository:

```
systems-programming-languages/
  _book/
  .quarto/
  index.qmd
  _quarto.yml
  resources.qmd
  cpp/
    ... notes ...
  code/
    c/
    cpp/
      ...
    rust/
    zig/
    nim/
    D/
```

The important point is:

```
systems-programming-languages/
```

is the repository and Quarto project root, but:

```
systems-programming-languages/code/cpp/
```

is the CMake project root for C++ code.

This means you can open the whole repository in VS Code while still telling CMake Tools to treat `code/cpp` as the actual CMake project.

1.1.2 2. VS Code configuration

Create:

```
systems-programming-languages/.vscode/settings.json
```

with:

```
{
  "cmake.sourceDirectory": "${workspaceFolder}/code/cpp"
}
```

This tells VS Code's CMake Tools extension:

> The CMake project does not start at the repository root. It starts at `code/cpp`.

So you can open:

```
code ~/Documents/notebooks/cs/systems-programming-languages
```

but CMake will configure from:

```
code/cpp/
```

Recommended `.gitignore` behavior:

```
/build/
/.quarto/
/_book/

.vscode/*
!.vscode/settings.json
```

This ignores generated files while keeping the useful project-local VS Code setting.

1.1.3 3. Build directory layout

Generated build files should not live inside the source folders. We chose a central build directory:

```
systems-programming-languages/
  build/
    cpp-debug/
  code/
    cpp/
      CMakeLists.txt
      CMakePresets.json
```

The CMake preset uses:

```
"binaryDir": "${sourceDir}/../../build/cpp-debug"
```

Since `sourceDir` is:

```
systems-programming-languages/code/cpp
```

this resolves to:

```
systems-programming-languages/build/cpp-debug
```

Executables are placed in:

```
systems-programming-languages/build/cpp-debug/bin/
```

because the root `CMakeLists.txt` contains:

```
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
```

So a target like:

```
cpp_learning_stroustrup_ch01_ex_hello
```

will build to:

```
build/cpp-debug/bin/cpp_learning_stroustrup_ch01_ex_hello
```

In normal VS Code usage, you do not need to `cd` into this build folder manually. CMake Tools can select, build, run, and debug targets directly.

1.1.4 4. CMake and Ninja

CMake is the high-level build configuration system. It reads `CMakeLists.txt`.

Ninja is the low-level build tool that actually performs the compilation. CMake generates Ninja build files.

The relationship is:

```
CMakeLists.txt
    ↓ read by cmake
build.ninja
    ↓ executed by ninja
compiled executables/libraries
```

The preset uses Ninja:

```
"generator": "Ninja"
```

You should be able to check your tools with:

```
cmake --version
ninja --version
g++ --version
```

You normally do not write Ninja files yourself. You write CMake files, and CMake generates the Ninja build files.

1.1.5 5. Root CMake project

The root CMake file is here:

```
systems-programming-languages/code/cpp/CMakeLists.txt
```

This is the root of the C++ build system.

It does several things:

1. declares the C++ project;
2. sets the C++ standard;
3. sets the runtime output directory;
4. defines reusable helper functions;
5. adds book directories with `add_subdirectory`.

Example:

```
cmake_minimum_required(VERSION 3.25)

project(cpp_studies LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)

function(add_cpp_study_examples target_prefix)
    file(GLOB example_sources CONFIGURE_DEPENDS
        "${CMAKE_CURRENT_SOURCE_DIR}/ex_*.cpp"
        "${CMAKE_CURRENT_SOURCE_DIR}/demo_*.cpp"
    )

    foreach(source_file IN LISTS example_sources)
        get_filename_component(source_name "${source_file}" NAME_WE)

        set(target_name "${target_prefix}_${source_name}")

        add_executable(${target_name} "${source_file}")

        target_compile_features(${target_name} PRIVATE cxx_std_20)

        if(CMAKE_CXX_COMPILER_ID MATCHES "GNU|Clang")
            target_compile_options(${target_name}
                PRIVATE
                    -Wall
                    -Wextra
                    -Wpedantic
            )
        elseif(MSVC)
            target_compile_options(${target_name}
                PRIVATE
```

```
                    /W4
            )
        endif()
    endforeach()
endfunction()

add_subdirectory(learning-cpp/stroustrup)
```

The helper function is defined at the root level so every book and chapter can use it.

1.1.6 6. Automatic example discovery

We chose the convention:

```
ex_*.cpp
demo_*.cpp
```

are automatically treated as executable programs.

For example:

```
ch01/
  ex_hello.cpp
  ex_variables.cpp
  demo_scope.cpp
```

automatically creates targets like:

```
cpp_learning_stroustrup_ch01_ex_hello
cpp_learning_stroustrup_ch01_ex_variables
cpp_learning_stroustrup_ch01_demo_scope
```

But files like:

```
vector.cpp
list.cpp
utils.cpp
helper.cpp
```

are not automatically made into executables.

This is important because later, in more complex chapters, you may have implementation files that are meant to support examples rather than become runnable programs themselves.

So this convention is future-proof:

```
ex_*.cpp      → standalone exercise/example program
demo_*.cpp    → demonstration program
anything else → support/library/source file, not auto-built as an executable
```

1.1.7 7. Book-level CMake files

A book directory might be:

```
code/cpp/learning-cpp/stroustrup/
  CMakeLists.txt
  ch01/
  ch02/
  ch03/
```

The book-level `CMakeLists.txt` adds chapter directories:

```
add_subdirectory(ch01)

## Later:
# add_subdirectory(ch02)
# add_subdirectory(ch03)
```

This means you only activate chapters when you have created their chapter-level `CMakeLists.txt`.

1.1.8 8. Chapter-level CMake files

A simple chapter looks like:

```
code/cpp/learning-cpp/stroustrup/ch01/
  CMakeLists.txt
  ex_hello.cpp
```

The chapter `CMakeLists.txt` is tiny:

```
add_cpp_study_examples(cpp_learning_stroustrup_ch01)
```

This calls the root-level helper function and creates one executable target for every `ex_*.cpp` and `demo_*.cpp` file in that chapter.

Example source file:

```
##include <iostream>

int main()
{
    std::cout << "Hello from Stroustrup ch01!\n";
    return 0;
}
```

1.1.9 9. CMake presets

The preset file is here:

```
systems-programming-languages/code/cpp/CMakePresets.json
```

Example:

```
{
  "version": 6,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 25,
    "patch": 0
  },
  "configurePresets": [
    {
      "name": "debug",
      "displayName": "Debug",
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/../../build/cpp-debug",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Debug",
        "CMAKE_EXPORT_COMPILE_COMMANDS": "ON"
      }
    }
```

```
  ],
  "buildPresets": [
    {
      "name": "debug",
      "displayName": "Debug",
      "configurePreset": "debug"
    }
  ]
}
```

This gives a reproducible Debug configuration.

From the command line, you can use:

```
cd code/cpp
cmake --preset debug
cmake --build --preset debug
```

But in VS Code, CMake Tools can use the preset directly.

1.1.10 10. Running programs in VS Code

With the CMake Tools extension installed, VS Code can see your CMake targets.

Typical workflow:

1. open the full `systems-programming-languages` folder;
2. CMake Tools uses `code/cpp` as the source directory;
3. choose the `Debug` configure preset;
4. configure;
5. select a target, for example:

```
cpp_learning_stroustrup_ch01_ex_hello
```

6. build/run/debug it from VS Code.

You do not have to manually go into:

```
build/cpp-debug/bin/
```

although you can run the binary manually if desired.

1.1.11 11. Simple chapters versus complex chapters

There are two main modes.

Simple standalone chapter

For beginner-style examples:

```
ch01/
  CMakeLists.txt
  ex_hello.cpp
  ex_variables.cpp
  demo_scope.cpp
```

Use:

```
add_cpp_study_examples(cpp_learning_stroustrup_ch01)
```

Each example is independent.

Complex chapter with implementation files

For data structures or algorithms:

```
ch05/
  CMakeLists.txt
  include/
    ch05/
      vector.hpp
  src/
    vector.cpp
  ex_basic_vector.cpp
  demo_vector_growth.cpp
```

Here you probably want a chapter-level library:

```
add_library(cpp_learning_stroustrup_ch05_lib
    src/vector.cpp
)

target_include_directories(cpp_learning_stroustrup_ch05_lib
    PUBLIC
        ${CMAKE_CURRENT_SOURCE_DIR}/include
)

add_cpp_study_examples(cpp_learning_stroustrup_ch05
    LINK_LIBRARIES cpp_learning_stroustrup_ch05_lib
)
```

For that to work elegantly, the helper function can later be upgraded to accept `LINK_LIBRARIES`.

1.1.12 12. Improved future-proof helper function

The first helper function is enough for simple examples.

A more advanced version allows chapter examples to automatically link against libraries:

```
function(add_cpp_study_examples target_prefix)
    set(options)
    set(one_value_args)
    set(multi_value_args LINK_LIBRARIES INCLUDE_DIRS SOURCES)

    cmake_parse_arguments(ARG
        "${options}"
        "${one_value_args}"
        "${multi_value_args}"
        ${ARGN}
    )

    file(GLOB example_sources CONFIGURE_DEPENDS
        "${CMAKE_CURRENT_SOURCE_DIR}/ex_*.cpp"
        "${CMAKE_CURRENT_SOURCE_DIR}/demo_*.cpp"
    )

    foreach(source_file IN LISTS example_sources)
        get_filename_component(source_name "${source_file}" NAME_WE)
```

```
        set(target_name "${target_prefix}_${source_name}")

        add_executable(${target_name}
            "${source_file}"
            ${ARG_SOURCES}
        )

        target_compile_features(${target_name} PRIVATE cxx_std_20)

        if(ARG_INCLUDE_DIRS)
            target_include_directories(${target_name}
                PRIVATE
                    ${ARG_INCLUDE_DIRS}
            )
        endif()

        if(ARG_LINK_LIBRARIES)
            target_link_libraries(${target_name}
                PRIVATE
                    ${ARG_LINK_LIBRARIES}
            )
        endif()

        if(CMAKE_CXX_COMPILER_ID MATCHES "GNU|Clang")
            target_compile_options(${target_name}
                PRIVATE
                    -Wall
                    -Wextra
                    -Wpedantic
            )
        elseif(MSVC)
            target_compile_options(${target_name}
                PRIVATE
                    /W4
            )
        endif()
    endforeach()
endfunction()
```

Then a complex chapter can simply say:

```
add_library(cpp_learning_stroustrup_ch05_lib
    src/vector.cpp
)

target_include_directories(cpp_learning_stroustrup_ch05_lib
    PUBLIC
        ${CMAKE_CURRENT_SOURCE_DIR}/include
)

add_cpp_study_examples(cpp_learning_stroustrup_ch05
    LINK_LIBRARIES cpp_learning_stroustrup_ch05_lib
)
```

This means all `ex_*.cpp` and `demo_*.cpp` files in that chapter automatically link to the chapter library.

1.1.13 13. Header-only libraries

Some libraries may be header-only, especially template-heavy C++ code.

A header-only chapter library looks like:

```
add_library(cpp_learning_stroustrup_ch08_lib INTERFACE)

target_include_directories(cpp_learning_stroustrup_ch08_lib
    INTERFACE
        ${CMAKE_CURRENT_SOURCE_DIR}/include
)

target_compile_features(cpp_learning_stroustrup_ch08_lib
    INTERFACE
        cxx_std_20
)

add_cpp_study_examples(cpp_learning_stroustrup_ch08
    LINK_LIBRARIES cpp_learning_stroustrup_ch08_lib
)
```

The rule is:

```
Has .cpp implementation files?
  Use normal add_library(... src/file.cpp)

Header-only?
  Use add_library(... INTERFACE)
```

For compiled libraries:

```
target_include_directories(my_lib PUBLIC ...)
target_link_libraries(my_lib PUBLIC ...)
```

For header-only libraries:

```
target_include_directories(my_lib INTERFACE ...)
target_link_libraries(my_lib INTERFACE ...)
```

1.1.14 14. Root-level, book-level, and chapter-level libraries

The build system can support reusable libraries at several levels.

Root-level common library

Useful across all C++ books:

```
code/cpp/common/
  include/
    cppstudy/
      timer.hpp
      print.hpp
  src/
    timer.cpp
```

Root `CMakeLists.txt`:

```
add_library(cppstudy_common
    common/src/timer.cpp
)

target_include_directories(cppstudy_common
```

```
    PUBLIC
        ${CMAKE_CURRENT_SOURCE_DIR}/common/include
)

target_compile_features(cppstudy_common PUBLIC cxx_std_20)
```

You can add more source files:

```
add_library(cppstudy_common
    common/src/timer.cpp
    common/src/print.cpp
    common/src/random.cpp
)
```

Headers may also be listed for IDE visibility, though they are not compiled directly:

```
add_library(cppstudy_common
    common/src/timer.cpp
    common/src/print.cpp
    common/include/cppstudy/timer.hpp
    common/include/cppstudy/print.hpp
)
```

Book-level library

Specific to one book:

```
learning-cpp/stroustrup/
  include/
    stroustrup/
      helpers.hpp
  src/
    helpers.cpp
```

Book `CMakeLists.txt`:

```
add_library(cpp_learning_stroustrup_lib
    src/helpers.cpp
)
```

```
target_include_directories(cpp_learning_stroustrup_lib
    PUBLIC
        ${CMAKE_CURRENT_SOURCE_DIR}/include
)

target_link_libraries(cpp_learning_stroustrup_lib
    PUBLIC
        cppstudy_common
)

target_compile_features(cpp_learning_stroustrup_lib PUBLIC cxx_std_20)

add_subdirectory(ch01)
add_subdirectory(ch02)
```

Chapter-level library

Specific to a chapter:

```
add_library(cpp_learning_stroustrup_ch05_lib
    src/vector.cpp
)

target_include_directories(cpp_learning_stroustrup_ch05_lib
    PUBLIC
        ${CMAKE_CURRENT_SOURCE_DIR}/include
)

target_link_libraries(cpp_learning_stroustrup_ch05_lib
    PUBLIC
        cpp_learning_stroustrup_lib
)

add_cpp_study_examples(cpp_learning_stroustrup_ch05
    LINK_LIBRARIES cpp_learning_stroustrup_ch05_lib
)
```

This gives the dependency chain:

```
cppstudy_common
    ↓
cpp_learning_stroustrup_lib
    ↓
cpp_learning_stroustrup_ch05_lib
    ↓
ex_*.cpp / demo_*.cpp executables
```

1.1.15 15. Meaning of `PUBLIC`, `PRIVATE`, and `INTERFACE`

A rough mental model:

```
PRIVATE
```

means:

> This target needs it internally, but users of this target do not inherit it.

```
PUBLIC
```

means:

> This target needs it, and users of this target also inherit it.

```
INTERFACE
```

means:

> This target does not use it for its own compilation, but anything linking to it inherits it.

For normal compiled libraries, you often use:

```
target_include_directories(my_lib PUBLIC ...)
target_link_libraries(my_lib PUBLIC ...)
```

For executables, you often use:

```
target_link_libraries(my_executable PRIVATE my_lib)
```

For header-only libraries, you use:

```
add_library(my_header_lib INTERFACE)

target_include_directories(my_header_lib INTERFACE ...)
target_link_libraries(my_header_lib INTERFACE ...)
```

1.1.16 16. Multiple libraries at the same level

You can have multiple libraries at the same level.

For example, root-level:

```
add_library(cppstudy_headers INTERFACE)

target_include_directories(cppstudy_headers
    INTERFACE
        ${CMAKE_CURRENT_SOURCE_DIR}/common/include
)

target_compile_features(cppstudy_headers
    INTERFACE cxx_std_20
)


add_library(cppstudy_utils
    common/src/timer.cpp
)

target_include_directories(cppstudy_utils
    PUBLIC
        ${CMAKE_CURRENT_SOURCE_DIR}/common/include
)

target_link_libraries(cppstudy_utils
    PUBLIC
        cppstudy_headers
)

target_compile_features(cppstudy_utils
    PUBLIC cxx_std_20)
```

This is useful if:

- one part is header-only;
- another part has `.cpp` files;
- different parts have different dependencies;
- you want to avoid linking everything everywhere.

But the recommendation is not to over-split too early.

Start with:

```
cppstudy_common
cpp_learning_stroustrup_lib
chapter_lib when needed
```

Only introduce multiple libraries at one level when there is a concrete reason.

1.1.17 17. Practical naming convention

Use globally unique target names.

For example:

```
cpp_learning_stroustrup_ch01_ex_hello
cpp_learning_stroustrup_ch05_lib
cpp_ads_goodrich_ch03_demo_stack
cpp_ads_goodrich_ch03_lib
cppstudy_common
```

This avoids CMake target name collisions.

CMake target names are global inside one CMake project, so two targets cannot have the same name.

1.1.18 18. Recommended development workflow

For a simple new chapter:

1. create the chapter folder:

```
code/cpp/learning-cpp/stroustrup/ch02/
```

2. create:

```
ch02/CMakeLists.txt
```

with:

```
add_cpp_study_examples(cpp_learning_stroustrup_ch02)
```

3. add the chapter in the book-level `CMakeLists.txt`:

```
add_subdirectory(ch02)
```

4. create examples:

```
ex_variables.cpp
demo_input.cpp
```

5. configure/build in VS Code.

For a complex chapter:

1. create:

```
include/
src/
ex_*.cpp
demo_*.cpp
```

2. define a chapter library;
3. link all examples to it using the improved helper function.

1.1.19 19. What this setup gives you

This setup gives you:

- one unified repository for notes and code;
- Quarto project remains at the top level;
- C++ build system is cleanly rooted at `code/cpp`;
- build files go into `/build/cpp-debug`;
- every `ex_*.cpp` and `demo_*.cpp` becomes a runnable target automatically;
- support `.cpp` files are not accidentally turned into executables;
- simple chapters remain simple;
- complex chapters can grow into proper library-plus-demo structures;
- root-level, book-level, and chapter-level libraries are all supported;

- header-only libraries are supported;
- VS Code can run/debug individual examples without manual terminal navigation.

The main design principle is:

```
Keep the default case extremely simple,
but make the structure capable of scaling when a book becomes more serious.
```

That is why the setup starts with tiny chapter CMake files but still supports common libraries, book libraries, chapter libraries, and automatic target generation.

1.2 Extending with Tests

Yes — the common way is not merely "compile files named `test_*.cpp`". The common CMake way is:

```
1. build each test file as an executable
2. register that executable as a CTest test using add_test(...)
3. run tests through CTest, VS Code Test Explorer, or CMake Tools test commands
```

CMake's testing support is based on `enable_testing()` / `include(CTest)` plus `add_test(...)`. CTest then discovers and runs the registered tests from the build tree. (cmake.org)

1.2.1 Examples versus tests

In our setup so far:

```
ex_*.cpp
demo_*.cpp
```

means:

```
build runnable programs
but do not treat them as formal tests
```

For tests, we could use:

```
test_*.cpp
```

to mean:

```
build runnable test programs
and register them with CTest
```

So a chapter might later look like:

```
ch05/
  CMakeLists.txt

  include/
    ch05/
      vector.hpp

  src/
    vector.cpp

  ex_basic_vector.cpp
  demo_vector_growth.cpp

  test_vector_push_back.cpp
  test_vector_size.cpp
```

The examples are run manually when you want to explore. The tests are run as part of a test suite.

1.2.2 Root-level setup

At the top of your C++ CMake project:

```
code/cpp/CMakeLists.txt
```

I would add:

```
include(CTest)
```

near the top, after `project(...)`.

For example:

```
cmake_minimum_required(VERSION 3.25)

project(cpp_studies LANGUAGES CXX)

include(CTest)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
```

`include(CTest)` automatically enables testing unless `BUILD_TESTING` is turned off. CMake's own documentation says `include(CTest)` is the usual way to add testing support, and the module creates the `BUILD_TESTING` option. (cmake.org)

So you can later configure with tests on or off.

1.2.3 Add a test helper function

Keep your existing `add_cpp_study_examples(...)` for examples/demos.

Then add a separate helper for tests:

```
function(add_cpp_study_tests target_prefix)
    set(options)
    set(one_value_args)
    set(multi_value_args LINK_LIBRARIES INCLUDE_DIRS SOURCES)

    cmake_parse_arguments(ARG
        "${options}"
        "${one_value_args}"
        "${multi_value_args}"
        ${ARGN}
    )

    file(GLOB test_sources CONFIGURE_DEPENDS
        "${CMAKE_CURRENT_SOURCE_DIR}/test_*.cpp"
    )

    foreach(source_file IN LISTS test_sources)
        get_filename_component(source_name "${source_file}" NAME_WE)
```

```
        set(target_name "${target_prefix}_${source_name}")

        add_executable(${target_name}
            "${source_file}"
            ${ARG_SOURCES}
        )

        target_compile_features(${target_name} PRIVATE cxx_std_20)

        if(ARG_INCLUDE_DIRS)
            target_include_directories(${target_name}
                PRIVATE
                    ${ARG_INCLUDE_DIRS}
            )
        endif()

        if(ARG_LINK_LIBRARIES)
            target_link_libraries(${target_name}
                PRIVATE
                    ${ARG_LINK_LIBRARIES}
            )
        endif()

        if(CMAKE_CXX_COMPILER_ID MATCHES "GNU|Clang")
            target_compile_options(${target_name}
                PRIVATE
                    -Wall
                    -Wextra
                    -Wpedantic
            )
        elseif(MSVC)
            target_compile_options(${target_name}
                PRIVATE
                    /W4
            )
        endif()

        add_test(
            NAME ${target_name}
            COMMAND ${target_name}
        )
    endforeach()
```

```
endfunction()
```

The key new line is:

```
add_test(
    NAME ${target_name}
    COMMAND ${target_name}
)
```

This registers the executable as a CTest test. CMake's `add_test` command is the standard way to define tests, and it only works as test infrastructure after testing has been enabled. (cmake.org)

1.2.4 Simple chapter with tests

Suppose:

```
ch01/
  CMakeLists.txt
  ex_hello.cpp
  test_arithmetic.cpp
```

Then:

```
add_cpp_study_examples(cpp_learning_stroustrup_ch01)
add_cpp_study_tests(cpp_learning_stroustrup_ch01)
```

`ex_hello.cpp` becomes a normal runnable example.

`test_arithmetic.cpp` becomes:

```
cpp_learning_stroustrup_ch01_test_arithmetic
```

and is also registered with CTest.

1.2.5 Complex chapter with a chapter library

Suppose:

```
ch05/
  CMakeLists.txt

  include/
    ch05/
      vector.hpp

  src/
    vector.cpp

  ex_basic_vector.cpp
  demo_vector_growth.cpp
  test_vector.cpp
```

Then:

```
add_library(cpp_learning_stroustrup_ch05_lib
    src/vector.cpp
)

target_include_directories(cpp_learning_stroustrup_ch05_lib
    PUBLIC
        ${CMAKE_CURRENT_SOURCE_DIR}/include
)

target_link_libraries(cpp_learning_stroustrup_ch05_lib
    PUBLIC
        cpp_learning_stroustrup_lib
)

target_compile_features(cpp_learning_stroustrup_ch05_lib
    PUBLIC
        cxx_std_20
)

add_cpp_study_examples(cpp_learning_stroustrup_ch05
    LINK_LIBRARIES cpp_learning_stroustrup_ch05_lib
)

add_cpp_study_tests(cpp_learning_stroustrup_ch05
    LINK_LIBRARIES cpp_learning_stroustrup_ch05_lib
)
```

Now:

```
ex_basic_vector.cpp       → runnable example
demo_vector_growth.cpp    → runnable demo
test_vector.cpp           → runnable CTest test
src/vector.cpp            → library implementation file, not auto-executable
```

1.2.6 What does a minimal test file look like?

Without any external testing framework, a simple test can just return `0` for success and nonzero for failure.

```
#include <cassert>

int main()
{
    int x = 2 + 2;
    assert(x == 4);
}
```

CTest treats exit code `0` as pass and nonzero as fail. The CMake tutorial describes CTest at this basic level as a task launcher that runs commands and reports success/failure from return codes. (cmake.org)

For your own vector:

```
#include <cassert>
#include <ch05/vector.hpp>

int main()
{
    Vector v;

    assert(v.size() == 0);

    v.push_back(42);

    assert(v.size() == 1);
    assert(v[0] == 42);
}
```

This is enough for early study code.

1.2.7 Running tests manually

From `code/cpp`:

```
cmake --preset debug
cmake --build --preset debug
ctest --test-dir ../../build/cpp-debug --output-on-failure
```

Or from the build directory:

```
cd ../../build/cpp-debug
ctest --output-on-failure
```

`ctest` is the CMake test driver program for build trees that use `enable_testing()` and `add_test()`. (cmake.org)

1.2.8 Test presets

Yes, `CMakePresets.json` can contain `testPresets`. CMake's preset system supports configure, build, and test presets, and CMake Tools also exposes commands like "Select Test Preset" and "Run Tests." (cmake.org)

You can extend your `CMakePresets.json` like this:

```
{
  "version": 6,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 25,
    "patch": 0
  },
  "configurePresets": [
    {
      "name": "debug",
      "displayName": "Debug",
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/../../build/cpp-debug",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Debug",
        "CMAKE_EXPORT_COMPILE_COMMANDS": "ON",
        "BUILD_TESTING": "ON"
```

```
      }
    }
  ],
  "buildPresets": [
    {
      "name": "debug",
      "displayName": "Debug",
      "configurePreset": "debug"
    }
  ],
  "testPresets": [
    {
      "name": "debug",
      "displayName": "Debug tests",
      "configurePreset": "debug",
      "output": {
        "outputOnFailure": true
      }
    }
  ]
}
```

Then from `code/cpp`:

```
cmake --preset debug
cmake --build --preset debug
ctest --preset debug
```

1.2.9 VS Code integration

In VS Code, once CMake has configured the project, tests registered through CTest can be run through CMake Tools. The CMake Tools documentation includes commands such as:

```
CMake: Select Test Preset
CMake: Run Tests
```

and supports CMake presets. (GitHub)

So the workflow becomes:

```
write test_*.cpp
→ configure/build
→ CMake/CTest discovers tests
→ run tests from VS Code or with ctest
```

1.2.10 Should `test_*.cpp` be auto-globbed?

For your study project, yes, I think this convention is reasonable:

```
ex_*.cpp      manually runnable examples
demo_*.cpp    manually runnable demos
test_*.cpp    CTest-registered tests
```

This is consistent with the style we already designed.

The important distinction is that `test_*.cpp` should not merely be built. It should also be registered with:

```
add_test(...)
```

That is what makes it a real CMake/CTest test.

1.2.11 Recommended setup

I would keep two separate helper functions:

```
add_cpp_study_examples(...)
add_cpp_study_tests(...)
```

rather than one giant function.

That keeps the meaning clean:

```
examples/demos → exploratory runnable programs
tests          → correctness checks registered with CTest
```

And in a chapter:

```
add_cpp_study_examples(cpp_learning_stroustrup_ch05
    LINK_LIBRARIES cpp_learning_stroustrup_ch05_lib
)

add_cpp_study_tests(cpp_learning_stroustrup_ch05
    LINK_LIBRARIES cpp_learning_stroustrup_ch05_lib
)
```

That is the natural extension of our current setup.

Part I

C++

2 Resources & Reading List

2.1 C++

2.1.1 Programming Intro with C++

- Programming Principles and Practice Using C++. Stroustrup
- Programming Abstractions in C++. Eric S. Roberts
- Data Structures and Problem Solving Using C++. Mark Allen Weiss
- Data Structures and Other Objects Using C++. Savitch

2.1.2 Learning C++ as a Language

- C++ Crash Course. Josh Lospinoso
- Tour of C++. Stroustrup
- C++ Primer. Lippman

2.1.3 Further C++

- C++ STL. Josuttis
- Discovering Modern C++. Gottschling
- Move Semantics. Josuttis
- C++ Templates. Josuttis
- Effective Modern C++. Scott Meyers
- Objekt-orientertes Programmieren in C++. Josuttis
- Functional Programming with C++. Cukic
- Modern C++ Programming with Test-Driven Development. Jeff Langr
- The C++ Programming Language. Stroustrup

2.1.4 Data Structure & Aglorithms with C++

- Data Structures and Algorithm Analysis with C++. Mark Allen Weiss
- Data Structures and Algorithms C++. Goodrich