# Lecture 09

## Introduction to CMake.
## Optimization, debugging, profiling, testing.

**Advanced Programming - SISSA, UniTS, 2023-2024**

**Pasquale Claudio Africa**

**21 Nov 2023**

# Outline

1. Introduction to CMake
2. Optimization
3. Debugging
4. Profiling
5. Testing

# Introduction to CMake

# Build systems

## Purposes

Build systems are a way to deploy software.

They are used to:

1. Provide others a way to configure **your** own project.
2. Configure and install third-party software on your system.

**Configure** means:

- Meet dependencies
- Build
- Test

# Build systems available

- **CMake**
  - **Pros**: Easy to learn, great support for multiple IDEs, cross-platform
  - **Cons**: Does not perform automatic compilation test for met dependencies.
- **GNU Autotools**
  - **Pros**: Excellent support for legacy Unix platforms, large selection of existing modules.
  - **Cons**: Slow, hard to use correctly, painful to debug, poor support for non-Unix platforms.
- **Meson**, **Bazel**, **SCons**, ...

**Package managers:**

- **Conan**, **vcpkg**, ...

# Let's try

Install dependencies, then compile and install.

**Doxygen** (CMake)

```
cd /path/to/doxygen/src/
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=/opt/doxygen ../
make -j<N>
(sudo) make install
```

**GNU Scientific Library** (autotools)

```
cd /path/to/gsl/src/
./configure --prefix=/opt/gsl --enable-shared --disable-static
make -j<N>
(sudo) make install
```

# Why CMake?

- More packages use CMake than any other system
- Almost every IDE supports CMake (or vice-versa)
- Really cross-platform, no better choices for Windows
- Extensible, modular design

**Who else is using CMake?**

- Netflix
- HDF Group, ITK, VTK, Paraview (visualization tools)
- Armadillo, CGAL, LAPACK, Trilinos (linear algebra and algorithms)
- deal.II, Gmsh (FEM analysis)
- KDE, Qt, ReactOS (user interfaces and operating systems)
- ...

# CMake basics

The root of a project using CMake must contain a **CMakeLists.txt** file.

```
cmake_minimum_required(VERSION 3.12)

# This is a comment.
project(MyProject VERSION 1.0
  DESCRIPTION "A very nice project"
  LANGUAGES CXX)
```

Please use a CMake version more recent than your compiler (at least ≥ 3.0).

Command names are **case insensitive**.

# CMake 101

- **Configure**

```
cd /path/to/src/
mkdir build && cd build
cmake .. [options...]
# Or:
# cmake -S /path/to/src/ -B /path/to/build/ [options...]
```

- **Compile**

```
 cd /path/to/build/
 make -j<N>
```

- **List variable values**

```
cd /path/to/build/
cmake /path/to/src/ -L
```

# Targets

CMake is all about targets and properties. An executable is a target, a library is a target. Your application is built as a collection of targets depending on each other.

```
# Header files are optional.
add_executable(my_exec my_main.cpp my_header.hpp)

# Options are STATIC, SHARED (dynamic) or MODULE (plugins).
add_library(my_lib STATIC my_class.cpp my_class.hpp)
```

# Target properties

Targets can be associated with various properties :

```
add_library(my_lib STATIC my_class.cpp my_class.hpp)
target_include_directories(my_lib PUBLIC include_dir)
# "PUBLIC" propagates the property to
# other targets depending on "my_lib".
target_link_libraries(my_lib PUBLIC another_lib)

add_executable(my_exec my_main.cpp my_header.h)
target_link_libraries(my_exec my_lib)
target_compile_features(my_exec cxx_std_20)
# Last command is equivalent to:
# set_target_properties(my_exec PROPERTIES CXX_STANDARD 20)
```

# Interacting with the outside world: local variables

```
set(LIB_NAME "my_lib")

# List items are space- or semicolon-separated.
set(SRCS "my_class.cpp;my_main.cpp")
set(INCLUDE_DIRS "include_one;include_two")

add_library(${LIB_NAME} STATIC ${SRCS} my_class.hpp)
target_include_directories(${LIB_NAME} PUBLIC ${INCLUDE_DIRS})

add_executable(my_exec my_main.cpp my_header.h)
target_link_libraries(my_exec ${LIB_NAME})
```

# Interacting with the outside world: cache variables

Cache variables are used to interact with the command line:

```
# "VALUE" is just the default value.
set(MY_CACHE_VARIABLE "VALUE" CACHE STRING "Description")

# Boolean specialization.
option(MY_OPTION "This is settable from the command line" OFF)
```

Then:

```
cmake /path/to/src/ \
  -DMY_CACHE_VARIABLE="SOME_CUSTOM_VALUE" \
  -DMY_OPTION=OFF
```

# Interacting with the outside world: environment variables

```
# Read.
message("PATH is set to: $ENV{PATH}")

# Write.
set(ENV{variable_name} value)
```

(although it is generally a good idea to avoid them).

# Control flow

```
if("${variable}") # Or if("condition").
    #
else()
    # Undefined variables would be treated as empty strings, thus false.
endif()
```

The following operators can be used.

Unary: `NOT`, `TARGET`, `EXISTS` (file), `DEFINED`, etc.
Binary: `STREQUAL`, `AND`, `OR`, `MATCHES` (regular expression), ...

Parentheses can be used to group.

# Branch selection

Useful for switching among different implementations or versions of any third-party library.

```
#ifdef USE_ARRAY
    std::array<double, 100> my_array;
#else
    std::vector<double> my_array(100);
#endif
```

How to select the correct branch?

# Pre-processor flags

```
target_compile_definitions(my_exec PRIVATE USE_ARRAY=1)
```

Or let the user set the desired flag:

```
option(WITH_ARRAY "Use std::array instead of std::vector" ON)

if(WITH_ARRAY)
    target_compile_definitions(my_exec PRIVATE USE_ARRAY=1)
endif()
```

# Modify files depending on variables

**`print_version.hpp.in`:**

```
void print_version() {
  std::cout << "Version number: " << @MY_PROJECT_VERSION@
            << std::endl;
}
```

**`CMakeLists.txt`:**

```
set(MY_PROJECT_VERSION 1.2.0)

configure_file(
  "${CMAKE_CURRENT_SOURCE_DIR}/print_version.hpp.in"
  "${CMAKE_CURRENT_BINARY_DIR}/print_version.hpp")
```

See also: `#cmakedefine`.

# Print messages and debug

Content of variables is printed with

```
message("MY_VAR is: ${MY_VAR}")
```

Error messages can be printed with

```
message(FATAL_ERROR "MY_VAR has the wrong value: ${MY_VAR}")
```

Commands being executed are printed with

```
cmake /path/to/src/ -B build --trace-source=CMakeLists.txt
make VERBOSE=1
```

# Useful variables

- **CMAKE_SOURCE_DIR**: top-level source directory
- **CMAKE_BINARY_DIR**: top-level build directory

If the project is organized in sub-folders:

- **CMAKE_CURRENT_SOURCE_DIR**: current source directory being processed
- **CMAKE_CURRENT_BINARY_DIR**: current build directory

```
# Options are "Release", "Debug",
# "RelWithDebInfo", "MinSizeRel"
set(CMAKE_BUILD_TYPE Release)

set(CMAKE_CXX_COMPILER "/path/to/c++")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall")
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY lib)
```

# Looking for third-party libraries

CMake looks for **module files** `FindPackage.cmake` in the directories specified in `CMAKE_PREFIX_PATH`.

```
set(CMAKE_PREFIX_PATH "${CMAKE_PREFIX_PATH} /path/to/modules/")

# Specify "REQUIRED" if the library is mandatory.
find_package(Boost 1.50 COMPONENTS filesystem graph)
```

If the library is not located in a system folder, often a hint can be provided:

```
cmake /path/to/src/ -DBOOST_ROOT=/path/to/boost/installation/
```

# Using third-party libraries

Once the library is found, proper variables are populated.

```
if(${Boost_FOUND})
    target_include_directories(my_lib PUBLIC
                               ${Boost_INCLUDE_DIRS})

    target_link_directories(my_lib PUBLIC
                            ${Boost_LIBRARY_DIRS})
    # Old CMake versions:
    # link_directories(${Boost_LIBRARY_DIRS})

    target_link_libraries(my_lib ${Boost_LIBRARIES})
endif()
```

# Compilation test

CMake can try to compile a source and save the exit status in a local variable.

```
try_compile(
    HAVE_ZIP
    "${CMAKE_BINARY_DIR}/temp"
    "${CMAKE_SOURCE_DIR}/tests/test_zip.cpp"
    LINK_LIBRARIES ${ZIP_LIBRARY}
    CMAKE_FLAGS
        "-DINCLUDE_DIRECTORIES=${ZIP_INCLUDE_PATH}"
        "-DLINK_DIRECTORIES=${ZIP_LIB_PATH}")
```

See also: `try_run`.

# Execution test

CMake can run specific executables and check their exit status to determine (un)successful runs.

```
include(CTest)
enable_testing()
add_test(NAME MyTest COMMAND my_test_executable)
```

# Organize a large project

```
cmake_minimum_required(VERSION 3.12)
project(ExampleProject VERSION 1.0 LANGUAGES CXX)

find_package(...)
find_package(...)

add_subdirectory(src)
add_subdirectory(apps)
add_subdirectory(tests)
```

# Tip: how to organize a large project

```
project/
├── apps/
│   ├── CMakeLists.txt
│   └── my_app.cpp
├── cmake/
│   └── FindSomeLib.cmake
├── doc/
│   └── Doxyfile.in
├── scripts/
│   └── do_something.sh
├── src/
│   ├── CMakeLists.txt
│   └── my_lib.{hpp,cpp}
├── tests/
│   ├── CMakeLists.txt
│   └── my_test.cpp
├── .gitignore
├── CMakeLists.txt
├── LICENSE.md
└── README.md
```

# Further readings

- Official documentation
- Modern CMake
- It's time to do CMake right
- Effective modern CMake
- More modern CMake

# Optimization

# Code optimization

**Code optimization** is the process of enhancing a program's performance, efficiency, and resource utilization without changing its functionality. It involves improving execution speed, reducing memory usage, and enhancing overall system responsiveness.

## Optimization techniques

- **Compiler optimizations:** Utilize compiler features to automatically enhance code during compilation.
- **Algorithmic optimization:** Improve the efficiency of algorithms and data structures.
- **Manual refactoring:** Restructure code for better readability, maintainability, and performance.
- **Profiling and analysis:** Use profiling tools to identify and optimize performance bottlenecks.

# Optimization options

The compiler enhances performance by optimizing CPU register usage, expression refactoring, and pre-computing constants.

- Disable optimization during debugging.
- Pass the `-O{n}` (`n={0,1,2,s,3}`) flag to the compiler to control optimization level, with `-Os` for space optimization and `-O3` for maximum optimization. Here a detailed list of optimizations enabled with each flag.
- Defining the `-DNDEBUG` preprocessor variable, standard assertion are ignored, resulting in faster code.

# Loop unrolling

It is beneficial to unroll small loops. For example, transform:

```
for (int i = 0; i < n; ++i) {
  for(int k = 0; k < 3; ++k) {
    a[k] += b[k] * c[i];
  }
}
```

to:

```
for (int i = 0; i < n; ++i) {
  a[0] += b[0] * c[i];
  a[1] += b[1] * c[i];
  a[2] += b[2] * c[i];
}
```

Compiler may unroll loops with `-funroll-loops`, but better performance isn't guaranteed.

# Prefetching constant values

Prefetch constant values inside the loop for further optimization:

```
for (int i = 0; i < n; ++i) {
  auto x = c[i];
  a[0] += b[0] * x;
  a[1] += b[1] * x;
  a[2] += b[2] * x;
}
```

# Avoid `if` inside nested loops

`if` statements, especially in nested loops, can be costly. Consider these improvements:

```
for(int i = 0; i < 10000; ++i) {
  for (int j = 1; j < 10; ++j) {
    if(c[i] > 0)
      a[i][j] = 0;
    else
      a[i][j] = 1;
  }
}
// Better:
for(int i = 0; i < 10000; ++i)
  if(c[i] > 0)
    for(int j = 0; j < 10; ++j) {
      a[i][j] = 0;
  else
    for(int j = 0; j < 10; ++j) {
      a[i][j] = 1;
}
```

# Sum of a vector: two strategies compared

```
double sum1(double *data, const size_t &size) {
    double sum{0};
    for (size_t j = 0; j < size; ++j)
        sum += data[j];
    return sum;
}

double sum2(double *data, const size_t &size) {
    double sum{0}, sum1{0}, sum2{0}, sum3{0};
    size_t j;
    for (j = 0; j < (size - 3); j += 4) {
        sum += data[j + 0];
        sum1 += data[j + 1];
        sum2 += data[j + 2];
        sum3 += data[j + 3];
    }
    for (; j < size; ++j)
        sum += data[j];
    sum += sum1 + sum2 + sum3;
    return sum;
}
```

# Which one is faster, `sum1` or `sum2`?

The number of floating point operations is the same in both cases!

The answer is not straightforward: it depends on the computer's architecture.

On my laptop (8th Gen Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz), **`sum2` is approximately 10 times faster than `sum1`** with `size = 1e9`! (see `examples/unrolling/unrolling.cpp`).

Why? The Streaming SIMD Extensions (SSE2) instruction set of the CPU allows for parallelization at the microcode level. It's a super-scalar architecture with multiple instruction pipelines to execute several instructions concurrently during a clock cycle. The code of `test2` better exploits this capability.

**Take-home message:** Counting operations doesn't necessarily reflect performance. Compiler optimizers can transform `sum1` into `sum2` automatically. Sometimes, giving it a hand is beneficial.

# Cache friendliness

Efficiency often depends on how variables are accessed in memory. Access variables contiguously for cache pre-fetching effectiveness. For example, if `mat` is a dynamic matrix organized **row-wise**:

```
// Not cache-friendly, inefficient.
for (j = 0; j < n_cols; ++j) {
  for (i = 0; i < n_rows; ++i) {
    a += mat(i, j);
  }
}

// Cache-friendly, thus more efficient.
for (i = 0; i < n_rows; ++i) {
  for (j = 0; j < n_cols; ++j) {
    a += mat(i, j);
  }
}
```

# Debugging

# Static analysis vs. debugging (1/2)

## Static analysis

- **Nature:** Examines code without executing it.
- **Purpose:** Identifies potential issues and coding standards violations.
- **Tools:** Code linters, security scanners, and complexity analyzers.
- **Integration:** Often part of development workflows or continuous integration.

## Debugging

- **Nature:** Inspects and troubleshoots code during runtime.
- **Purpose:** Locates and resolves bugs, runtime errors, and unexpected behavior.
- **Tools:** Debuggers with features like breakpoints and variable inspection.
- **Integration:** Interactive process during development or post-runtime.

# Static analysis vs. debugging (2/2)

## Key differences

- **Timing:** Static analysis is pre-runtime; debugging is during or post-runtime.
- **Focus:** Static analysis emphasizes code quality; debugging resolves runtime issues.
- **Use cases:** Static analysis is proactive; debugging is reactive.
- **Automation:** Static analysis tools can be automated; debugging is more interactive.
- **Complementarity:** Both are complementary, with static analysis preventing issues and debugging addressing runtime problems.

# Static analysis

Static analysis tools analyze source code by inspecting it for potential issues, vulnerabilities, or adherence to coding standards. Common ones include:

- cppcheck
- cpplint
- clang-tidy

Some of the checks they perform:

- Automatic variable checking.
- Bounds checking for array overruns.
- Unused functions, variable initialization and memory duplication.
- Invalid usage of Standard Template Library functions and idioms.
- Missing `#include`s.
- Memory or resource leaks, performance errors.

# Other useful tools

- Cling is an interactive C++ interpreter, built on LLVM and Clang. It's part of the ROOT project at CERN and can be integrated into a Jupyter workspace ( see here ). While experimental, an interpreter aids in code prototyping.
- Compiler Explorer to check how code translates into assembly language.
- C++ Insights allows viewing source code through a compiler's eyes.

# Debuggers

**Debuggers** are software tools that enable developers to inspect, analyze, and troubleshoot code during the development process. They provide a set of features for identifying and fixing errors in programs.

## Key features

- **Breakpoints:** Pauses program execution at specified points to inspect variables and code.
- **Variable inspection:** Allows developers to examine the values of variables during runtime.
- **Step-through execution:** Enables line-by-line execution for precise debugging.
- **Call stack analysis:** Displays the sequence of function calls leading to the current point in code.

# Debugging

During code development, debugging allows step-by-step execution. To use a debugger, compile with `-g` (which implies no optimization). `-g` adds information for locating source lines in machine code.

Two debugging types:

- **Static debugging:** Analyze core dump if code aborts.
- **Dynamic debugging:** Execute through a debugger, breaking at points to examine variables.

Two common debuggers are `gdb` and `lldb`. See, e.g.:

- Debugging with GDB
- GDB cheat sheet
- GDB tutorial

# Debugging levels

Debugging can be at different levels, and using `-g -O` together is allowed. `-g` tells the compiler to provide extra information for the debugger. However, line-by-line debugging reliability decreases with optimization. `-g` implies `-O0` by default.

Debugging levels and special optimization options linked to debugging:

- `-g0`: No debugging information.
- `-g1`: Minimal information for backtraces.
- **The default debugging level is 2.**
- `-g3`: Extra information, including macro definitions.
- `-Og`: Special optimization option. Enables optimizations without interfering with debugging.

# Main commands of `gdb` / `lldb`

- `run` : Run the program.
- `break` : Set a breakpoint at a line/function.
- `where` : Show location and backtrace.
- `print` : Display variable/expression value.
- `list n` : Show lines around line n.
- `next` : Go to the next instruction, proceeding through subroutines.
- `step` : Go to the next instruction, entering called functions.
- `continue` : Continue executing.
- `backtrace` : Print memory stack after program aborts.
- `quit` : Exit the debugger.
- `help` : Displays help information.

# Other debugging tools

`valgrind`, a suite of tools for debugging and profiling. It can find memory leaks, unassigned variables, or check memory usage:

**Find memory leaks:**

```
valgrind --tool=memcheck --leak-check=yes --log-file=file.log executable
```

**Check memory usage:**

```
valgrind --tool=massif --massif-out-file=massif.out --demangle=yes executable
ms_print massif.out > massif.txt
```

`massif.txt` indicates memory usage during the program execution.

# Profiling

# Profilers

A **profiler** in software development is a tool or set of tools designed to analyze the runtime behavior and performance of a computer program. It provides detailed information about resource utilization, execution times, and function calls during the program's execution.

## Key objectives

- **Performance analysis:** Profilers offer insights into how much time a program spends in different functions, helping identify performance bottlenecks.
- **Resource usage:** They measure memory consumption, CPU utilization, and other system resources, aiding in optimizing resource-intensive operations.
- **Function call tracing:** Profilers track the sequence of function calls, enabling developers to understand the flow of execution.

# `gprof`

`gprof` is the GCC simple profiler. In order to use it, compile the code with the `-pg` option at both the compilation and linking stages.

When executing the code, it generates a file called `gmon.out`, which is then utilized by the profiler:

```
gprof --demangle executable > file.txt
```

Then `file.txt` will contain valuable information about the program execution.

# Main options of `gprof`

`gprof` offers a range of options. The main ones are:

- `--annotated-source[=symspec]`: Prints annotated source code. If `symspec` is specified, print output only for matching symbols.
- `-I dirs`: List of directories to search for source files.
- `--graph[=symspec]`: Prints the call graph analysis.
- `--demangle`: Demangles mangled names (essential for C++ programs).
- `--display-unused-functions`: As the name says.
- `--line`: Line-by-line profiling (but maybe better use gcov).

# `callgrind`

`callgrind` is a tool of `valgrind` that you may call, for instance, as:

```
valgrind --tool=callgrind --callgrind-out-file=grind.out --dump-line=yes ./myprog
```

Compile the program with `-g` and **optimization activated**. The option `--dump-lines` is used for line-by-line profiling.

Afterward, post-process the binary file `grind.out`, e.g., using `kcachegrind`:

```
kcachegrind grind.out
```

It opens a graphical interface.

# Other profilers

There are alternative profilers, some useful in a parallel environment:

- perf : Lightweight CPU profiling.
- gperftools : Formerly Google Performance Tools.
- TAU (Tuning and Analysis Utilities) : Profiling and tracing toolkit for parallel programs.
- Scalasca : Performance analysis for parallel applications on distributed memory systems.

# Testing

# Verification vs. validation

## Verification: Ensuring correct implementation

## Validation: Confirming desired behavior

- **Verification:** Conducted during development, tests **individual components** separately. Specific tests demonstrate correct functionality, covering the code and checking for memory leaks.
- **Validation:** Performed on the **final code**. Assesses if the code produces the intended outcome - convergence, reasonable results, and expected computational complexity.

# Types of testing

- **Unit testing:** Testing individual components (functions, methods, or classes) to ensure each behaves as expected. It focuses on a specific piece of code in isolation.
- **Integration testing:** Verifying that different components/modules of the software work together as intended. It deals with interactions between different parts of the system.
- **Regression testing:** Ensuring recent code changes do not adversely affect existing functionalities. It involves re-running previous tests on the modified codebase to catch unintended side effects.

# Importance of testing

- **Early detection of bugs:** Testing allows early detection and fixing of bugs, reducing the cost and time required for debugging later in the development process.
- **Code reliability:** Testing ensures the code behaves as expected and provides reliable results under different conditions.
- **Documentation:** Test cases serve as documentation for how different parts of the code are expected to work. They help other developers understand the intended behavior of functions and classes.

# Unit testing in C++

In C++, unit testing often uses frameworks like Google Test, Catch2, or CTest itself (from the CMake ecosystem).

Here's a simple example using `gtest`:

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

TEST(MyLibrary, AddTwoNumbers) {
  EXPECT_EQ(add(2, 3), 5);
}

int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

In this example, we test the `add` function from the `mylibrary` module.

# Test-Driven Development (TDD)

**TDD** is a software development approach where tests are written before the actual code. The cycle is writing a test, implementing the code to pass the test, and refactoring.

## Advantages

TDD encourages modular and testable code, ensuring all parts of the codebase are covered by tests. It also starts by thinking at how code should be used (bottom-up strategy), possibly guiding the design of the interface exposed.

## Process

1. Write a test defining a function or improvements succinctly.
2. Run the test to ensure it fails, showing it doesn't pass.
3. Write the simplest code to make the test pass.
4. Run the test and verify it passes.
5. Refactor the code for better structure or performance.

# Continuous Integration (CI) and testing

- **CI:** Frequently integrating code changes into a shared repository. Automated builds and tests ensure new changes don't break existing functionalities.
- **Benefits:**
  - Early detection of integration issues.
  - Regular validation of code against the test suite.
  - Confidence in the stability of the codebase.
- **Popular CI Tools:**
  - Jenkins
  - Travis CI
  - GitHub Actions
  - GitLab CI

# Coverage

Code coverage is a metric used in software testing to measure the extent to which source code is executed during the testing process. It provides insights into which parts of the codebase have been exercised by the test suite and which parts remain untested.

## Key concepts

- **Lines of code**: Code coverage is often expressed as a percentage of lines of code that have been executed by tests. The goal is to have as close to 100% coverage as possible.
- **Branches** and **paths**: In addition to lines, code coverage can also consider branches and execution paths within the code. This provides a more detailed analysis of the code's behavior.

# Coverage with gcov

`GCC` supports program coverage with `gcov`. Compile with `-g -fprofile-arcs -ftest-coverage` and no optimization. For shared objects with `dlopen`, add the option `-Wl,--dynamic-list-data`.

Run the code, producing `gcda` and `gcno` files. Use `gcov` utility:

```
gcov [options] source_file_to_examine [or executable]
```

Text files with code and execution counts for each line are created.

### Main options of gcov

`gcov` offers various options:

- `--demangled-names`: Demangle names, useful for C++.
- `--function-summaries`: Output summaries for each function.
- `--branch-probabilities`: Write branch frequencies to the output file.

# `lcov` and `genhtml`: nice graphical tools for `gcov`

The `gcov` output is verbose. With `lcov` and `genhtml`, you get a graphical view:

Compile with `gcov` rules, then:

```
lcov --capture --directory project_dir --output-file cov.info
genhtml cov.info --output-directory html
```

`project_dir` is the directory with `gcda` and `gcno` files. In the `html` directory, open `index.html` in your browser.

# Introduction to Python.