CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Advanced C++ Course

C++ infrastructure, text-based resources

CSCS

C++ infrastructure

- as most statically typed languages C++ is compiled (exluding cern's ROOT environment), but compilersare not the only tool supporting it, we will give an overview of:

- compilers
- object utils
- linker
- build tools
- package managers
- Web resources
  - web tools
  - Textual/Documentation Resources
- Editors/IDEs
  - remote editing
- Remote Environements
  - containers (dev-containers)
  - gitpod
- sanitizers/analyzers
- debugger
- profilers
- Test frameworks
- CI

Compilers

- there are many compilers, for example:
  - GCC
  - clang
  - MSVC
  - Intel compilers (oneAPI DPC++, the older icc)
  - NVidia nvc++/nvcc
  - IBM XL compiler

What does a Compiler do?

- A compiler looks at a source file, and at the included header files containing the declarations of the known functions/types and produces an object file that targets an architecture (x86_64, arm64,...) and ABI (memory layout, alignemens, basic data types, calling convention, system calls, ...)
- An object file is normally a binary file that contains several segments of data, its main components are symbols (labels/addresses), the TEXT (code) segment and DATA segment. It can also have debugging information (DWARF), which can also be stored in external files.
- To inspect the files one can normally use use binutils (or llvm-) objdump, readelf, nm, ldd (and on windows DUMPBIN.EXE).

Linker

- An object file contains executable code, but normally is not executable, because it refers to functions defined in other files or libraries.
- A function call needs to jump to the address of the code to execute
- the linker replaces the reference to the function with the address of its code (often relative to the PC or some other register)
- linking can happen
  - when creating the executable (static linking)
  - at load time (dynamic linking)
    - shared libraries might be relocated (moved to another adress), require position independent code
    - ldd can show the shared libraries loaded

ABI - Calling convention

- stack layout
- which registers are caller and which ar callee saved
- dynamic linking

Build systems

- C++ is used in some very large projects with thousands of files
- creating libraries and executables manually calling the compiler is not really an option you want something that does it for you, and
  - recompiles only what is needed when you do a small change (fast incremental builds)
  - does not forget to update some dependencies (is correct)
  - simplifies cleanup and full builds (automatable)
- there are various options, if you work within a large project the best bet is probably to continue with what they are using
- we present some good options

CMake

- cmake.org is probably the most widely used C++ build system
- cross platform system that generates build scripts for other build tools like:
  - Unix Makefiles (the default)
  - Ninja (`--GNinja`) , very fast build tool an excellent choice if available (Linux, macOS, Windows),
  - XCode project files (macOS)
  - Visual Studio project files (Windows MSVC)
- specific projects might have their own CMake macros which you are encouraged to use in the CMakeLists.txt (Qt for example)

`tutorial/CMakeLists.txt` from the cmake tutorial

```
cmake_minimum_required(VERSION 3.10)
project(Tutorial VERSION 1.0) # set the project name and version
set(CMAKE_CXX_STANDARD 17) # specify the C++ standard
set(CMAKE_CXX_STANDARD_REQUIRED True)
add_subdirectory(MathFunctions) # add the MathFunctions library
# add the executable
add_executable(Tutorial tutorial.cxx)
# and link the library
target_link_libraries(Tutorial PUBLIC MathFunctions)
```

`tutorial/MathFunctions/CMakeLists.txt`

```
add_library(MathFunctions MathFunctions.cxx)

# state that anybody linking to us needs to include the current source dir
# to find MathFunctions.h, while we don't.
target_include_directories(MathFunctions
                           INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
                           )
```

create a build directory with
`cmake -S tutorial -B tutorial_build -GNinja`
build with `cmake --build tutorial_build`

Bazel

- bazel.build is the open source version of the blaze tool used by google
- it supports several other languages beside C++
- strives to be fast, correct and support multiple platforms (linux, macOS, Windows)

Example

`main/BUILD` from bazel.build/start/cpp

```
cc_library(
    name = "hello-greet",
    srcs = ["hello-greet.cc"],
    hdrs = ["hello-greet.h"],
)

cc_binary(
    name = "hello-world",
    srcs = ["hello-world.cc"],
    deps = [
        ":hello-greet",
        "//lib:hello-time",
    ],
)
```

build with `bazel build //main:hello-world`

Scons

- scons.org a python based build tool
- it supports several other languages beside C++
- strives to be fast, correct and support multiple platforms (linux, macOS, Windows)

Example

`SConstruct` see scons doc

```
env = Environment()
hello_lib = env.SharedLibrary('#/lib/hello', ['libhello.c'])
exe = env.Program('main', ['main.c'], LIBS=hello_lib)

env.Install('/usr/lib', hello_lib)
env.Install('/usr/bin', exe)
env.Alias('install', '/usr/bin')
env.Alias('install', '/usr/lib')
```

build with `scons`

Don't invent your own

- In general any code you do not have to write is a win
- libraries can help you with code that has been used and reviewed already by others.
- One often underestimates the time needed until a code works well and solves the issue.
- try to use stdlib or existing libraries

Package Managers

- package managers can help you install the libraries you require.
- we will look at a couple of HPC or C++ package managers

Spack

- spack.io is a package manager for supercomputers, Linux and macOS.
- It makes it easy to install specific versions of scientific libraries that use a specific compiler, or architecture specific compilation flags.
- can be installed as user without root privilegies
- easy to package your own software with it

Getting started installing a library (libelf), as described in the spack documentation:

```
$ git clone -c feature.manyFiles=true https://github.com/spack/spack.git
$ cd spack/bin
$ ./spack install libelf
```

- packaging your tool/library supports git or github as sources, which makes creating packages for your tool relatively easy as described in the packaging guide.

Conan

- conan.io is an open source, decentralized C/C++ package manager.
- it is well integrated with buildtools (cmake, scons,...)
- it is easy to create packages

`conanfile.txt` from the conan tutorial

```
[requires]
zlib/1.2.11
[tool_requires]
cmake/3.22.6
[generators]
CMakeDeps
CMakeToolchain
```

and its companion `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.15)
project(compressor C)
find_package(ZLIB REQUIRED)
message("Building with CMake version: ${CMAKE_VERSION}")
add_executable(${PROJECT_NAME} src/main.c)
target_link_libraries(${PROJECT_NAME} ZLIB::ZLIB)
```

```
cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake \
  -DCMAKE_BUILD_TYPE=Release
```

Vcpkg

- vcpkg.io is a free C/C++ package manager for acquiring and managing libraries
- Maintained by the Microsoft C++ team and open source contributors.

```
git clone https://github.com/Microsoft/vcpkg.git
```

Make sure you are in the directory you want the tool installed to before doing this.

```
./vcpkg/bootstrap-vcpkg.sh
```

Install libraries for your project

```
vcpkg install [packages to install]
```

Using vcpkg with CMake

```
cmake -B [build directory] -S . \
-DCMAKE_TOOLCHAIN_FILE=[path to vcpkg]/scripts/buildsystems/vcpkg.cmake
```

Web Resources

- godbolt.org (Compiler Explorer)
  - *many* different compilers
  - nicely annotated assembly output
  - supports several popular libraries (ranges, {fmt}, ...)
  - code round-trip between Compiler Explorer, C++ Insights and Quick Bench possible
  - we will use it quite a bit
- quick-bench.com a site to easily a quickly benchmark C++ code (and inspect disassembly)
- cppinsights.io lets you see the C++ code that clangs generates for lambdas, range for-loops, structured bindings,...
- gcc-explorer.com makes error logs clickable, openening an IDE
- build-bench.com compare the build time of code snippets with various compilers
- www.onlinegdb.com lets you debug a program using gdb

Textual/Documentation Resources

- The isocpp.org site is main entry point for the C++ standard and related things (news, articles, blogs,...)
  - The C++ Core Guidelines are a set of tried-and-true guidelines, rules, and best practices about coding in C++
- C++ reference covers the language and standard library
- cplusplus.com has tutorials articles
- hackingcpp.com a good collection of various C++ learning resources
  - with an excellent Tools Ecosystem Overview.

Editors and IDEs

- C++ needs to be written, there are various editors that can be used, from the classic vim/neovim, emacs and micro to Kate, Geany QtCreator and Visual Studio Code.
- to help editing, completion, go to definition, find all references,... can be very useful.
- They can be provided by a Language Server. As long as you use an editor that supports the Language Server Protocol it you can have good support in it.
- clangd provides excellent support for C++ in an editor independent way.
  - ```
    cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=1; ln -s ~/myproject/compile_commands.json ~/myproject-build/
    ```
    enables good completion with cmake, you can use bear for the other build tools
  - ccls is an alternative language server
- Enabling clangd in your editor
  - vim: YouCompleteMe or LanguageClient-neovim
  - nvim: coc.nvim and coc-clangd or LanguageClient-neovim
  - emacs eglot or lsp-mode
- If you have no preferred Editor Visual Studio Code is flexible editor with good support.
    ```

Remote editing

- Several editors can access code on a remote machine from the editor running on your local machine via ssh, and give you a mostly native experience.
- Visual Studio Code Remote development extension has
  - Remote - SSH - Connect to any location by opening folders on a remote machine/VM using SSH.
  - Dev Containers - Work with a separate toolchain or container-based application inside (or mounted into) a container.
  - WSL - Get a Linux-powered development experience in the Windows Subsystem for Linux.
  - Remote - Tunnels - Connect to a remote machine via a secure tunnel, without configuring SSH.
- Gitlab Web IDE and github.dev provides vscode based editing in the Webbrowser

Remote containers

- Setting up the whole environment might take time an effort, using a Container can simplify it, providing a reporoducible environment with all the tools.
- DevContainerscontainers.dev is a specifications for containers that are developed to be used to complile/work on code.
- The code is on the local machine and gets mounted in the DevContainer
- The container contains all the tools to compile and run/debug the code
- The Visual Studio Code extension can support their use

Remote containers

- As further step aways from your local machine containers can be run remotely, and provide an remote environment. Some commercial services do exactly that
- gitpod.io keeps both the code and the container to compile/run/debug it running in the cloud
- codespaces is a similar service provided by github

Code Formatters

- Well and consistently formatted code makes the code easier to read, which is important, but manually formatting the code can take time, and it is difficult for many developers to be all consistent with each other. A code formatter and maybe a pre-commit hook can take care of that
- clang format (of llvm/clang) can be customized by a `.clang-format` file that you can add to your repository (you can generate it with the online clang-format-configurator), and `git clang-format` reformats the changes you have staged, so that you can review them
  part of the llvm/clang project
- other options:
  - Artistic Style
  - Uncrustify

Git

- you should always use some version control software
- git is the de-facto industry standard for version control
- little reason not to use it, we assume you do
- set a `.gitignore`
- commit hooks for example in a `.githooks` directory and then explain how to activate them (`git config --local core.hooksPath .githooks/`)

CI

- you shoudl always try to have automatic tests, indeed you should try to use Test Driven Development (TDD).
- automatic test on each commit (CI) help making sure that the project stays working, and failure/regressions are catched quickly
- both gitlab and github let you set up pipelines that ensure that

Sanitizers

- Sanitizers add extra checks to the compiled code
- this is normally a small overhead with respect to the default execution
- a run of the tests with sanitizers enabled is an excellent CI action
- with cmake consider using ECMEnableSanitizers (documentation or sanitizer-cmake

- ASAN (Address Sanitizer)
  g++ -fsanitize=address
  clang++ -fsanitize=address
  cl.exe /fsanitize=address
- UBSAN (Undefined Behavior Sanitizer)
  g++ -fsanitize=undefined
  clang++ -fsanitize=undefined
- LeakSanitizer
  g++ -fsanitize=leak
  clang++ -fsanitize=leak
- ThreadSanitizer
  g++ -fsanitize=thread
  clang++ -fsanitize=thread

Analyzers

- Static analyzers might find potential issues, this can be useful, but soethime you have quite a bit of noise (false positives), making it more costly for large codebases
  * Clang-Tidy
  clang-based linter tool, part of the extra clang tools
  diagnoses programming errors, style violations, interface misuse
  valgrind
- SonarSource code analyzer with an opensource core with commercial extensions
- snyk.io opensource code analyzers with commercial suite to find vulnerabilities

Profiling

- Profile the real thing (unoptimized code might behave *very* differently)
- gprofng
  - modern non intrusive sampling based
  - supports optimized multithreaded applications
  - HTML report (if wanted)
  - simular to the older perf
- Valgrind (cachegrind, callgrind)
  - Simulates the processor, 100s times slower, but exact behaviour of optimized code
  - --tool=memcheck leak, invalid read/write detection
  - --tool=callgrind runtime profiling
  - --tool=cachegrind cache profiling
  - --tool=massif heap memory profiling
  - integration into various IDEs or visual tools, for example {q/k}cachegrind

Profiling 2

- VTune Profiler commercial, but free usage available
- Apple Instruments various tools, quite nice, also non intrusive sampling based methods
- Coz – Causal Profiler
  - unique approach to profiling
  - creates causal profile: "optimizing function X will have effect Y"
  - profile is based on performance experiments
  - program is partitioned into parts based on progress points (that are set in source code)
  - no additional instrumentation of source code required

Single Process Debuggers

- debuggers let one inspect a running program and find issues
  - GDB the GNU debugger
  - lldb the LLVM project's debugger
  - rr records program state over time
    - replay & debug same recording many times
    - reverse execution
    - chaos mode for catching intermittent bugs
- frontends
  - DDD official GNU debugger frontend
  - seer Qt based debugger frontend
- Several IDEs integrate the debugger, for example QtCreator, Visual Studio Code

Parallel debuggers (mpi,...)

Debugging parallel applications is not easy, often one resorts to a printf approach.
There are commercial debuggers that support it, look at what is supported where you run.

- Linaro Forge (DDT) (which used to be Allinea DDT, and then Amd forge) is a parallel debugger that is also available at CSCS
- totalview is another parallel commercial parallel debugger

Testing frameworks

- Tests and CI are important, they let you modify and refactor the code more agressivly because you know that it still works
- Indeed in the TDD (Test Driven Development) you start by writing the tests even before you write the code
- There are various frameworks that help you write tests, but the crucial thing are the tests, not the framework
- If you work in a project that already uses a framework, consider using it (will probably integrate already well with CI)
- Here we present some good options if you start from scratch

Catch2

- allows for well-structured, self-documenting tests
- relatively easy to set up
- very good and concise documentation
- data generator helpers
- set of predefined matchers for comparing values
- microbenchmarking tools
- logging
- Boost Software License 1.0
  Intro/Tutorial Manual Tutorial

```cpp
#include <catch2/catch_test_macros.hpp>

unsigned int Factorial( unsigned int number ) {
    return number <= 1 ? number : Factorial(number-1)*number;
}

TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}
```

Other Good Test Frameworks

- DocTest
  - easy to set up, one header only
  - good choice for small/quick projects
  - approach allows for well-structured, self-documenting tests
  - very good and concise documentation
  - BSL-1.0
    +Tutorial Manual Quick Example
- Google Test
  - likely the most used unit test framework.
  - Robust, scales well, a solid choice
- Boost Test
  - Well structured basic unit test, cosider it if you use the Boost library

Mocking

- Mocking is useful when you have an interface, either with virtual methods or via template argument (formalized with Concepts or not), and you want to test the code that uses it.
- from Pragmatic Unit Testing
- When the real object
  - produces unpredictable results, like a stock-market quote feed, or has nondeterministic behavior.
  - is difficult to set up, or is slow.
  - has hard to trigger behavior. For example, a network error or out-of-memory condition
  - The real object has or is a user interface.
  - does not yet exist
- When the test needs to ask the real object about how it was used.
- Dangers:
  - is stateful and pushes toward stateful APIs (especially if you write test before implementing)
  - a stateless API is normally better if possible

Mocking libraries

- You can create a mock object that returns a predetermined response for given calls, and checks that given methods are called (potentially checking the sequence and the arguments of the call).
- Mocking libraries as the following help you doing that
- trompeloeil
  - A thread-safe header-only mocking framework for C++11/14
  - a good choice for example with catch2
  - Boost Software License 1.0
- Google Mock
  - a stable well rounded mocking library.
  - now part of google test, obvious choice if you use google test.

Conclusions

- C++ has a rich ecosystem, take advantage of it
- Use compilers that support the newer standard versions and more checks
- Make sure that building is possible with a single commands and iteration is quick
- consider using package managers to install your dependencies and to share your libraries
- Consult web resources to stay up to date, lookup references and quickly check code snippets
- Use an Editor/IDE that understands C++ and helps you understand and refactor your code
- use a code formatter to keep your code consistently and nicely formatted and easy to read without spending too much time on it
- write tests, they help you to be confident that your code works, and that changes do not break your program
- set up CI to continuosly and automatically
  - build and run tests
  - also with sanitizers, analyzers and benchmarks
- when required use profilers to find places to optimize and debugger to identify bugs

Exercises

- set up and editor with `clangd`, and make sure that the `compile_commands.json` file is visible to it
- add the dependencies of your project with a package manager, so that you can install all of them with a single command
- If your project is a library try to make it available via a package manager
- add some tests (using `catch2`, or another library)
- add CI to your project if it doesn't have it yet
  - add a build stage
  - execute the tests
  - do the same also with sanitizers
- build an optimized (but with debug info) single processor application (but possibly multithreaded) and profile it with
  - `gprofng` / `instruments` (realtime sampling)
  - `valgrind` and `q/kcachegrind` (exact, but *much* slower)
  - `coz` (~realtime, requires some modifications to set the range unit, you can use sample applications)