Concepts of C++ Programming

Lecture 1: Overview and Hello World

Alexis Engelke

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

Winter 2024/25

Module "Concepts of C++ Programming" (CIT323000)

Goals

- Write good and modern C++ code
- Apply widely relevant C++ constructs
- Understand some advanced language concepts

Non-Goals

- Become experts in C++
- Fancy language features
- Apply involved optimizations

Prerequisites

- Fundamentals of object-oriented programming EIDI, PGdP
- Fundamentals of data structures and algorithms GAD
- Beneficial: operating systems, computer architecture GBS, ERA

Lecture Organization

- Lecture: Mon 14:30 – 17:00, MW 0001
  - Lecturer: Dr. Alexis Engelke `engelke@in.tum.de`
  - Live stream and recording via RBG: `https://live.rbg.tum.de/`
  - Tweedback for questions during lecture
- Exercises: Tue 14:15 – 15:45, Interims II HS 3
  - Florian Drescher, Mateusz Gienieczko

- Material: `https://db.in.tum.de/teaching/ws2425/cpp/`
- Zulip-Streams: #CPP, #CPP Homeworks, #CPP Random/Memes

- Exam: written exam *on your laptop*, on-site, 90 minutes
  - Open book, but no communication/AI tools allowed
  - Same submission system as for homework

Homework

- 1–2 programming tasks as homework every week
  - Released on Monday, deadline next Sunday 11:59 PM
- Automatic tests and grading, points only for completely solved tasks
  - Typically all[1] tests provided with the assignment

- Container environment provided, no support for other setups
- Submission via git+ssh only

- Grade bonus: 0.3 for 75% of exercise points
  - Applies **only** for the main exam, not for the retake
- Cheating in homework $\rightsquigarrow$ 5.0U in final grade

[1] We may add extra cases to prevent hard-coding of test cases.

Literature

Primary

- **C++ Reference Documentation.** (`https://en.cppreference.com/`)

- Lippman, 2013. *C++ Primer (5th edition)*. Only covers C++11.
- Stroustrup, 2013. *The C++ Programming Language (4th edition)*. Only covers C++11.
- Meyers, 2015. *Effective Modern C++. 42 specific ways to improve your use of C++11 and C++14.*.

Supplementary

- Aho, Lam, Sethi & Ullman, 2007. *Compilers. Principles, Techniques & Tools (2nd edition)*.
- Tanenbaum, 2006. *Structured Computer Organization (5th edition)*.

What is C++?

- Multi-paradigm general-purpose programming language
  - Imperative programming
  - Object-oriented programming
  - Generic programming
  - Functional programming
- Key characteristics
  - Compiled
  - Statically typed
  - Facilities for low-level programming

Some C++ History

Initial development

- Bjarne Stroustrup at Bell Labs (since 1979)
    - Originally “C with classes”, renamed in 1983 to C++
- In large parts based on C
- Inspirations from Simula67 (classes) and Algol68 (operator overloading)
- Initially developed as a C++-to-C converter (Cfront)

First ISO standardization in 1998 (C++98)

- Further amendments in following years (C++03/11/14/17/20)
- Current standard: C++23

C++ Standard vs. Implementations

- C++ *standard* specifies requirements for C++ *implementations* about language features and standard library
- "Implementation" consists of: compiler, standard library impl, OS, ...

- Some things are specified rigidly in the standard
- Some things are *implementation-defined*
  - Standard specifies options, implementation chooses one and documents that
  - Example: size of an `int`
- Implementations can offer extensions[2]

[2] https://clang.llvm.org/docs/LanguageExtensions.html

Why Study C++?

- Performance
  - Very flexible level of abstraction
  - Direct mapping to hardware capabilities easily possible
  - Zero-overhead rule: "What you don't use, you don't pay for."
- Scales to large systems (with some discipline)
- Interoperability with other languages, esp. C

- *Huge* amount of legacy code needs developers/maintainers
  - compilers, databases, simulations, ...

This Lecture

- Go bottom-up through important language constructs
  - Some things (e.g. standard library) appear rather late
  - Cyclic dependencies are unavoidable
- Focus: widely used constructs and important cases
  - Topic selection based on relevance real-world projects
  - Many special cases not discussed, lecture will be inaccurate at times
  - Use the C++ reference!

Hello World!

```cpp
#include <print>
int main() {
  std::println("Hello␣World!");
  return 0;
}
```

On the command line:

```
$ clang++ -std=c++23 -o hello hello.cpp
$ ./hello
Hello World!
```

Hello World, explained[3]

```cpp
// Make print and println available
#include <print>

// Definition of function main().
// Program execution starts at main.
int main() {
  // std:: is a namespace prefix. std is for the C++ standard library
  std::println("Hello␣World!");

  // End program with exit code 0. (zero = everything ok, non-zero = error)
  return 0;
}
```

[3] A bit hand-wavy, but we have to start somewhere.

Program Arguments

- `main` can take two paramters to hold command-line arguments
  - `int argc`: number of arguments
  - `char** argv`: the actual arguments, ∼array of strings
  - First argument is the program invocation itself (e.g., `./hello2`)

```
#include <print>
int main(int argc, char** argv) {
  std::println("Hello␣{}!", argv[1]); // DON’T DO THIS
  return 0;
}
```

```
$ clang++ -std=c++23 -o hello2 hello2.cpp
$ ./hello2 World
Hello World!
$ ./hello2
Segmentation fault
```

Debugging 101

- Pass -g to Clang to enable debug info generation
- Run gdb `./hello2`

```
$ clang++ -g -std=c++23 -o hello2 hello2.cpp
$ gdb ./hello2
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
(gdb) backtrace
// ...
#16 in main (argc=0x1, argv=0x7fffffffe868) at hello2.cpp:3
(gdb) up 16
(gdb) print argc
1
(gdb) quit
```

Debugging 102

- Print debugging.

```
#include <print>
int main(int argc, char** argv) {
  std::println("argc={}", argc);
  std::println("Hello␣{}!", argv[1]);
  return 0;
}
```

```
$ clang++ -std=c++23 -o hello2 hello2.cpp
$ ./hello2 World
Hello World!
$ ./hello2
Segmentation fault
```

Program Arguments, attempt 2

```
#include <print>
int main(int argc, char** argv) {
  if (argc >= 2)
    std::println("Hello␣{}!", argv[1]);
  else
    std::println("Hi␣there!");
  return 0;
}
```

```
$ clang++ -std=c++23 -o hello2 hello2.cpp
$ ./hello2 World
Hello World!
$ ./hello2
Hi there!
```

Compiler Flags

Compiler invocation: `clang++ [flags] -o output inputs...`

- `-std=c++23` — set standard to C++23
  - Always specify the version of the C++ standard!
- `-g` — enable debugging information
- `-Wall` — enable many warnings
- `-Wextra` — enable some more warnings
  - Always compile with `-Wall -Wextra`! Warnings often hint at bugs.
- `-O0` — no optimization, typically good for debugging
- `-O1`/`-O2`/`-O3` — enable optimizations at specified level

Build Systems: CMake

- Frequent use of long compiler commands is tedious and error-prone
- Manual work doesn't scale to larger projects
- Different systems may require different flags

- CMake: build system specialized for C/C++
  - Widely used by large projects and supported by many IDEs
- `CMakeLists.txt` specifies project, files, etc.

- Reference: `https://cmake.org/cmake/help/latest/`

CMake Example

CMakeLists.txt:

```
# Require a specific CMake version, here 3.20 for C++23 support
cmake_minimum_required(VERSION 3.20)
# Set project name, required for every project
project(hello2)
# We use C++23, basically adds -std=c++23 to compiler flags
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
# Compile executable hello2 from hello2.cpp
add_executable(hello2 hello2.cpp)
```

On the command line:

```
$ mkdir build; cd build # create separate build directory
$ cmake ..
$ cmake --build .
$ ./hello2
```

Further CMake Commands and Variables

- `add_executable(myprogram a.cpp b.cpp)`
  Define an executable to be built from the source files `a.cpp` and `b.cpp`
- `add_compile_options(-Wall -Wextra)`
  Add `-Wall -Wextra` to compiler flags

- `set(CMAKE_CXX_COMPILER clang++)`
  Set C++ compiler to `clang++`
- `set(CMAKE_BUILD_TYPE Debug)`
  Set "build type" `Debug` (other values: `Release`, `RelWithDebInfo`); affects optimization and debug info

Variables can be set on the command line invocation of CMake:

```
cmake ..  -DCMAKE_BUILD_TYPE=RelWithDebInfo
```

Overview and Hello World – Summary

- C++ is a compiled, widely-used, multi-paradigm language
- Program execution typically starts at `int main()`
- Command line arguments accessible via `argc`/`argv`
- Basic debugging techniques: GDB and print debugging
- Important compiler options for warnings and optimizations
- Basic usage of CMake for building C++ projects

Overview and Hello World – Questions

- What are key characteristics of the C++ language?
- Why is C++ one of the most important languages today?
- How to access program arguments?
- What are important flags for compiling C++ code with Clang?
- How to debug a compiled C++ program with GDB?
- What is a segmentation fault?
- What are advantages of using a build system like CMake?