Concepts of C++ Programming

winter term 2023/24

PD Dr. Tobias Lasser

Computational Imaging and Inverse Problems (CIIP)
Technical University of Munich

Contents

Who?

- Lecture:
  - PD Dr. Tobias Lasser
    - Computational Imaging and Inverse Problems (CIIP)
- Exercises:
  - David Frank, Jonas Jelten
    - Scientific staff at CIIP

What?

Aim of the lecture:

- study the concepts of modern C++ programming
- learn to apply those concepts in elegant and efficient solutions

Target audience:

- Bachelor students of Informatics (3rd term onwards)
- Master students of Informatics, BMC, CSE, Robotics
- anyone who is interested!

When and where?

Dates, places, and links

- Tuesday, 14:15 - 15:45, Interims II Hörsaal 2 (103)
  - Live Stream, recording: `https://live.rbg.tum.de/`
  - Tweedback: `https://tweedback.de/`
- Thursday, 17:00 - 18:30, Galileo Hörsaal (8120.EG.001)
  - Video conference: `https://bbb.rbg.tum.de/tob-w3o-4un-gp6` (no recordings!)

Generally:

- Tuesday: lecture
- Thursday: exercises

Tweedback Session-ID today: zmqt

- URL: `https://tweedback.de/zmqt`

Online resources

Chat platform

`https://zulip.cit.tum.de`, Streams #CPP, #CPP Homeworks

- discussions, answering questions

Moodle course

`https://www.moodle.tum.de/course/view.php?id=90655`

- news and announcements
- materials (slides, exercise sheets etc.)

Website

`https://ciip.in.tum.de/teaching/cpp_ws23.html`

Contact and feedback

Questions

- chat: `https://zulip.cit.tum.de`, Streams #CPP, #CPP Homeworks
- during lecture:
  - in person
  - Tweedback (today: session ID zmqt)
- email: `lasser@cit.tum.de` (in exceptional cases only)

Feedback

- chat
- email

Feedback is always welcome!

Examination

- Written exam:
  - *date not set yet*
  - in presence
  - 90 minutes duration
  - no repeat exam!
- Homework:
  - weekly programming assignments
  - passing the automated tests yields 2 points per week (total of 24 points achievable, as currently planned)
- Final grade:
  - points from written exam and from homeworks added together
  - final grade will be derived from the sum of points

Homework assignments

Programming assignments

- each week you receive an exercise sheet (via Moodle) with some programming assignments
- submit your solutions via `https://gitlab.lrz.de`
- automatic tests run on your solutions

- you will get two points per homework assignment where you pass all the automated tests within the deadline (1 week)
- total achievable points: 24 (as currently planned)
- points will be added to the points achieved in the written exam to form the final grade

- details on how to submit the homework will be presented in the first exercise session (Thursday, Oct. 19, 2023)

Homework assignments: brief details

- one homework assignment every Thursday via Moodle
- deadline for automated tests to pass: +1 week (Thursday 17:00)
- your solutions have to be uploaded to a GitLab repository
- how to get access to your GitLab repository:
  - log in to `https://gitlab.lrz.de`
  - edit your name (Profile, Account, Username) if you wish
  - in Moodle: go to "Add your GitLab username" activity
    - add your GitLab profile username
      (check `https://gitlab.lrz.de/-/profile/account` if unsure)
  - request access for the Waiting Room at
    `https://gitlab.lrz.de/cppcourse/ws2023/waiting-room`
  - your personal repository will be setup automatically
  - if there are any issues: use our chat at `https://zulip.cit.tum.de`, stream #CPP Homeworks

Lectures

Lecture format

- traditional slide-based lecture

- all materials will be available in Moodle
- feel free to ask questions any time via Tweedback!
  - I will try to make sure to pause often to answer your comments there

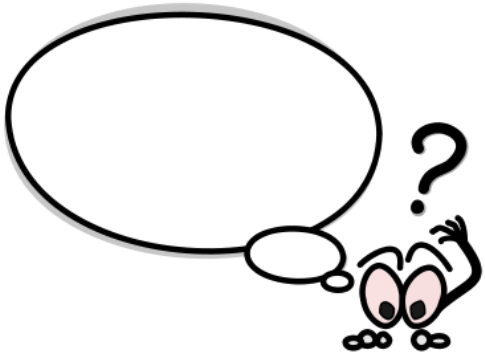
Exercises

Exercise format

- one exercise sheet per week
- exercise session will discuss
  - the previous week's assignments
  - give some pointers to the current exercise sheet

- all materials will be available in Moodle
- the level of interaction is up to you!

Questions?



Contents

Overview of topics

- basic concepts
  - syntax, strong typing, type deduction
  - compilation units, ODR
- concepts for build systems and dependency management
  - automatic compilation, linking, and dependencies
  - continuous testing and integration
- concepts for procedural programming
  - functions, parameter passing, lambdas, overloads, error handling
- concepts for containers and iterators
  - containers in STL, iterators, algorithms, other STL types
- concepts for object-oriented programming
  - classes, inheritance, polymorphism, RTTI

Overview of topics (cont.)

- concepts for generic programming
  - templates, variadic templates, fold expressions
  - CRTP and expression templates
- concepts for resource management
  - RAII, pointers, universal references, ownership, copy/move
- concepts for compile-time programming
  - constexpr, template recursion, SFINAE, type traits
- concepts for parallel programming
  - threads, atomics, async and futures, parallel STL
- outlook to future concepts (C++23 etc.)
  - coroutines, executors, networking, modular standard library

Please note

This course is still in flux

→ any feedback from you is welcome!

Survey

- What is your experience in modern C++? $\longrightarrow$ Tweedback quiz

Contents

What is C++?

C++ is a multi-paradigm, general-purpose programming language, supporting:

- procedural programming
- object-oriented programming
- generic programming
- functional programming

Key characteristics:

- static typing
- compiled language
- facilities for low-level programming

Some C++ history

Early development:

- 1979: Bjarne Stroustrup begins work on C with Classes
- 1983: name changed to C++
- 1985: reference book The C++ Programming Language
  - also: release of the first commercial implementation
- 1998: standardization as ISO/IEC 14882:1998, C++98
- 2003: minor revision of ISO standard, C++03

Some C++ history (cont.)

Development of modern C++:

- 2011: new ISO standard (major revision), C++11
- 2014: minor revision of ISO standard, C++14
- 2017: revision of ISO standard, C++17
- 2020: revision of ISO standard, C++20
- 2023 (almost done): revision of ISO standard, C++23

Although modern C++ is (mostly) backwards compatible, it is almost a new programming language compared to C++98/03. The focus is on safe programming techniques with zero overhead.

Why C++?

- Performance:
  - high performance for all types (including user-defined)
  - low-level hardware access is possible
  - zero overhead rule: "You don't pay for what you don't use." (B. Stroustrup)

- Flexibility:
  - flexible level of abstraction (very low-level to very high-level)
  - supports several programming paradigms
  - excellent scaling to very small and very large systems
  - interoperability with other programming languages
  - extensive ecosystem (tool chains, libraries)

Why C++? Examples

```
1  template <class... Ts>
2  constexpr auto average (Ts... nums) requires (sizeof...(nums) > 0)
3  {
4      return (nums + ...) / sizeof...(nums);
5  }
```

computes the average of basically anything

Why C++? Examples

```
1  auto z = (log(c * exp(-1/2 * square(xn - mu) / sq))).sum();
```

this emits CUDA code at compile-time to compute Gaussian log likelihood (using the correct library)

Why C++? Examples

```
1  for (auto n : std::views::iota(101, 200)
2                | std::views::filter([](auto v) { return v % 7 == 0; })
3                | std::views::take(3) )
4      std::cout << n << "\n";
```

prints the last three numbers divisible by 7 in the range $[101, 200]$

Why C++? Examples

You can still do regular C:

```
1  int main(int b,char**i){long long n=B,a=I^n,r=(a/b&a)>>4,y=atoi(*++i),
2  _=(((a^n/b)*(y>>T)|y>>S)&r)|(a^r);printf("%.8s\n",(char*)&_);}
```

(Source: winner of 2020 International Obfuscated C Code Contest)

Contents

A first C++ program

File first.cpp:

```cpp
#include <print>

int main() {
    std::println("Hello world!");
}
```

A first C++ program (cont.)

File first.cpp:

```
1  #include <print>
2
3  int main() {
4      std::println("Hello world!");
5  }
```

In order to run it, we have to compile it:

```
$ c++ -std=c++23 -Wall -Werror first.cpp -o first
```

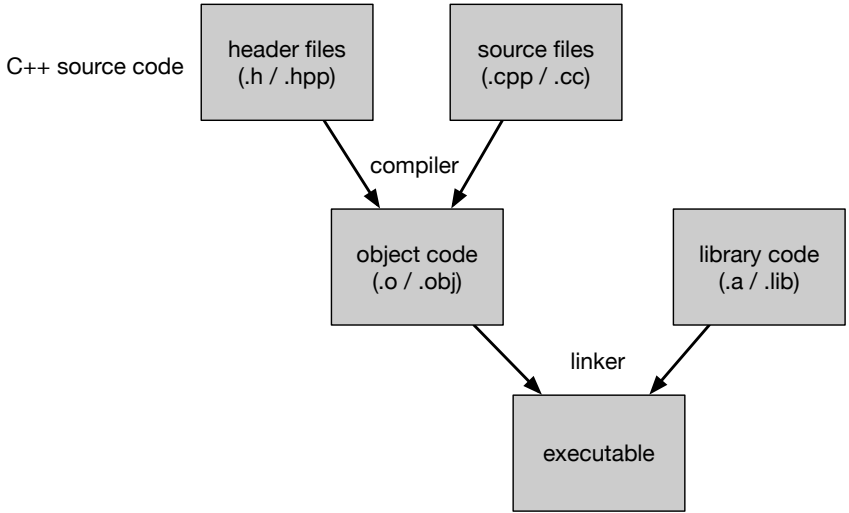
and then run it:

```
$ ./first
```

which produces:

```
Hello world!
```

Compilation and linking



Translation units

- the input of the compiler is called translation unit
- the output is the object code (.o / .obj)

- typically, a translation unit consists of:
  - a source file (.cpp / .cc)
  - some header files (.h / .hpp) that are `#include`'d

- `#include` is done via the preprocessor
  - in fact, the included file is literally included into the source code (which explains some of the long compilation times)
  - *side note:* this relic of C was finally superseded in C++20 (modules), but support of compilers / build systems is still lacking

Header and source files: example

sayhello.h:

```
1  // declaration
2  void sayHello();
```

sayhello.cpp:

```
1  #include <print>
2
3  #include "sayhello.h"
4
5  // definition
6  void sayHello() {
7      std::println("Hello world!)";
8  }
9
10 int main() {
11     sayHello();
12 }
```

The purpose of the header/source split is to have

- declarations in header files
  - these are easy to include
  - also in other translation units
- definitions in source files

One definition rule

One definition rule (ODR)

- all objects (functions, variables, templates etc.) have *at most* one definition in any translation unit
- all objects or non-inline functions that are used have *exactly* one definition in the entire program
- some things like types, templates, and inline functions can be defined in more than one translation unit, but they must be effectively identical

For all subtleties and exceptions, please see the standard.

The job of compiler and linker

- the compiler uses the declaration to know the signature of the declared object (function, variable etc.)
- the compiler does **not** resolve the symbols if they are defined externally (i.e. in another object file or library)
- the linker does resolve these symbols
- this often results in errors, for example:
  - missing symbols when forgetting to link an object file or library
  - multiple definitions of symbols when violating the ODR

ODR examples

In same translation unit, compiler catches ODR violations:

```
1  int i{5}; // OK: declares and defines i
2  int i{6}; // ERROR: redefinition of i
```

Separate declaration and definition is required to break circular dependencies:

```
1  void bar(); // declares function bar
2  void foo() { // declares and defines function foo
3      bar();
4  }
5
6  void bar() { // (re-)declares and defines function bar
7      foo();
8  }
```

ODR examples (cont.)

File a.cpp:

```
1  int foo() {
2      return 1;
3  }
```

File b.cpp:

```
1  int foo() {
2      return 2;
3  }
4
5  int main() {}
```

Now the linker will catch the ODR violation:

```
$ c++ -c a.cpp -o a.o
$ c++ -c b.cpp -o b.o
$ c++ a.o b.o
duplicate symbol 'foo()' in:
    a.o
    b.o
ld: 1 duplicate symbol for architecture x86_64
clang: error: linker command failed with exit code 1
```

Header guards

- a file may transitively include the same header several times
- this can lead to violations of the ODR:

File headerI.h:

```
1  int i = 1;
```

File headerJ.h:

```
1  #include "headerI.h"
2
3  int j = i;
```

File main.cpp:

```
1  #include "headerI.h"
2  #include "headerJ.h" // ERROR: i is defined twice
```

- solution: header files ensure that they are included at most once in a translation unit via header guards

Header guards: the portable way

File headerI.h:

```
1  // use any unique name
2  // usually composed from filename
3  #ifndef H_headerI
4  #define H_headerI
5
6  int i = 1;
7
8  #endif
```

File headerJ.h:

```
1  #ifndef H_headerJ
2  #define H_headerJ
3
4  #include "headerI.h"
5
6  int j = i;
7
8  #endif
```

Header guards: the elegant way

File headerI.h:

```
1  #pragma once
2
3  int i = 1;
```

File headerJ.h:

```
1  #pragma once
2
3  #include "headerI.h"
4
5  int j = i;
```

- drawback: this `#pragma once` is not in the C++ standard
- but all the big compilers support it

Outlook: C++20 modules

- C++20 modules replace the C-preprocessor based `#include` with proper modules using import/export
- example:

File sayhello.cpp:

```
1  // declare and export module
2  export module sayHello;
3
4  // import print library
5  import <print>;
6
7  // export the function
8  export void sayHello() {
9      std::println("Hello world!");
10 }
```

File main.cpp:

```
1  // import our module
2  import sayHello;
3
4  int main() {
5      sayHello();
6  }
```

Outlook: C++20 modules (cont.)

- modules also support the split of interface/implementation:

File sayhello.cppm:

```
1  // declare and export module
2  export module sayHello;
3
4  // export the function
5  export void sayHello();
```

File sayhello.cpp:

```
1  // define module implementation
2  module sayHello;
3
4  // import print library
5  import <print>;
6
7  void sayHello() {
8      std::println("Hello world!");
9  }
```

File main.cpp:

```
1  // import our module
2  import sayHello;
3
4  int main() {
5      sayHello();
6  }
```

Contents

Getting a C++ compiler

Depending on your operating system, there are many options.

Here are some popular ones:

- Linux: get gcc or clang via your favorite package manager
  - get a current version, e.g. gcc 13.2 or clang 17
- macOS:
  - installing Xcode will net you a clang version
    - unfortunately, those clang versions not very recent
  - alternatively, get gcc or clang via managers such as brew
    - to get the current versions
- Windows:
  - get Visual Studio with Visual C++ (no cost for TUM students)
  - get Cygwin or MinGW with gcc (current version!)
  - or use WSL to install gcc or clang (current versions!)

Getting a development environment

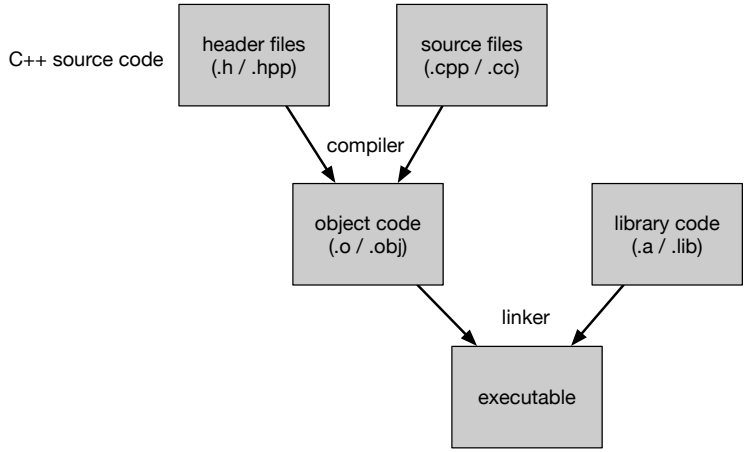
- any text editor (e.g. vim, Emacs) and a command line will do
- if you want more comfort, get an IDE, such as
  - QtCreator
  - Visual Studio Code
  - CLion (no cost for TUM students)

Investigate tools

- start getting familiar with important tools, such as
  - git for source code management
  - CMake for build automation and dependency management
  - a debugger (such as lldb or gdb) for debugging

Contents

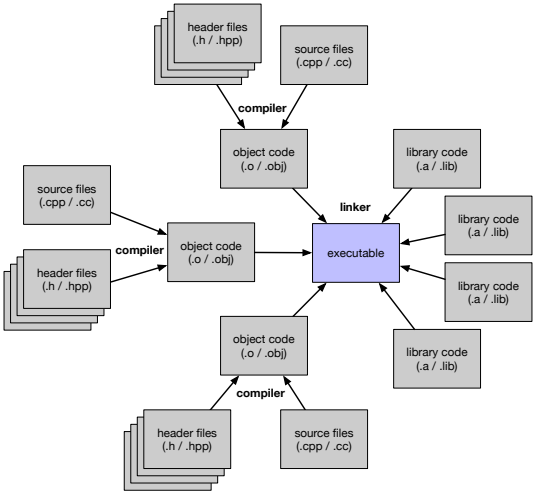
Compilation and linking



To compile and link:

```
$ c++ -std=c++23 -Wall -Wextra -O3 -c -o source.o source.cpp
$ c++ -std=c++23 -Wall -Wextra -O3 -o executable source.o library.a
```

More realistic example



To compile and link: many `$ c++ ...` calls

Automatic compilation and linking

- C++ projects usually consist of many source files (.cpp) and header files (.h) that need to be compiled and linked
  - tedious to do this manually
  - when one of the source files (.cpp) changes, only the corresponding code should be recompiled (not everything)
  - when one of the header files (.h) changes, only the source files that `#include` it should be recompiled

$\Rightarrow$ automate using a build system!

- examples are:
  - make, MSBuild, ninja, and many others
- we will use CMake, a build system that is also independent of the platform and the compiler

Contents

CMake

- CMake is a meta build system
  - independent of platform and compiler
  - specifies the build in CMakeLists.txt files
  - using its own language
  - generates output for other build systems, for example make files, MSBuild files, ninja files, and many others
- CMake is very commonly used for C++ projects
- it is natively supported by most IDEs (such as QtCreator, CLion, Visual Studio Code, or Visual Studio)
- documentation is available online, for example:
  - reference: `https://cmake.org/cmake/help/latest/`
  - guide: `https://cliutils.gitlab.io/modern-cmake/`

A basic CMakeLists.txt file

CMakeLists.txt:

```
1  cmake_minimum_required(VERSION 3.27)
2  project(exampleProject LANGUAGES CXX)
3
4  add_executable(exampleProject main.cpp)
```

Building:

```
$ mkdir build; cd build   # create separate build directory
$ cmake ..                # generate make files
-- The CXX compiler identification is Clang 17.0.1
[...]
-- Configuring done
-- Generating done
-- Build files have been written to: /home/X/cmake_example/build
$ make                    # build the project
Scanning dependencies of target exampleProject
[ 50%] Building CXX object CMakeFiles/exampleProject.dir/main.cpp.o
[100%] Linking CXX executable exampleProject
[100%] Built target exampleProject
```

Important CMake commands

- `cmake_minimum_required(VERSION 3.27)`
  require (at least) a specific version (mandatory)
- `project(exampleProject LANGUAGES CXX)`
  define a C++ project with name "exampleProject"
- `add_executable(myProgram a.cpp b.cpp)`
  define an executable named "myProgram" (a target) built from the source files a.cpp and b.cpp
- `add_library(myLibrary c.cpp d.cpp)`
  define a library named "myLibrary" (a target)
- `target_include_directories(myProgram PUBLIC inc)`
  where to look for include files for the target "myProgram"
- `target_link_libraries(myProgram PUBLIC myLibrary)`
  which libraries to add to target "myProgram"

CMake variables

CMake uses variables to adapt/customize the build process.
Variables are set:

- in CMakeLists.txt: `set(FOO bar)`
  (set the variable `${FOO}` to "bar")
- or on the command line: `cmake -D FOO=bar`
- or you can also use `ccmake` or `cmake-gui` to adjust variables manually

Important variables are:

- `CMAKE_CXX_COMPILER`
  set e.g. to "clang++" to use the clang compiler
- `CMAKE_BUILD_TYPE`
  set to "Debug" or "Release", which affects compiler optimization and debug information

Target properties

Variables are global. It is preferred to use target-specific properties, such as:

- `target_include_directories(myProgram PUBLIC inc)`
  where to look for include files for "myProgram"
- `target_link_libraries(myProgram PUBLIC myLibrary)`
  which libraries to link to "myProgram"
- `target_compile_features(myProgram PUBLIC cxx_std_23)`
  enable the C++23 standard for "myProgram"

Properties can be PUBLIC or PRIVATE, determining whether those properties get propagated to other targets using this target (PUBLIC) or not (PRIVATE).

CMake and subdirectories

- larger projects are usually split up into subdirectories
- CMake expects the main CMakeLists.txt in the top-level folder to contain the `project()` command
  - the subdirectories are explicitly added using the command `add_subdirectory(subdir)`
  - the subdirectories contain their own CMakeLists.txt file (without `project()`)

CMake example directory structure

```
ExampleProject
├── CMakeLists.txt
├── myLibrary
│   ├── CMakeLists.txt
│   ├── myLibrary.h
│   ├── myLibrary.cpp
│   ├── awesomeFunctions.h
│   └── awesomeFunctions.cpp
└── src
    ├── CMakeLists.txt
    └── main.cpp
```

- the root CMakeLists.txt contains the `project()` statement
- it includes myLibrary using `add_subdirectory()`
- it includes src using `add_subdirectory()`

- for a bigger example, check out the homework assignments in GitLab!

Contents

Version control

- version control is essential in projects with many developers to be able to synchronize the code
- it has many advantages even when working alone
- git is currently the most popular version control system
  - open source
  - decentralized (i.e. no server required)
  - very efficient using branches and tags
- documentation available at:
  - reference: `https://git-scm.com/docs`
  - book: `https://git-scm.com/book/en/v2`
  - live explorer: `http://git-school.github.io/visualizing-git/`

Continuous integration

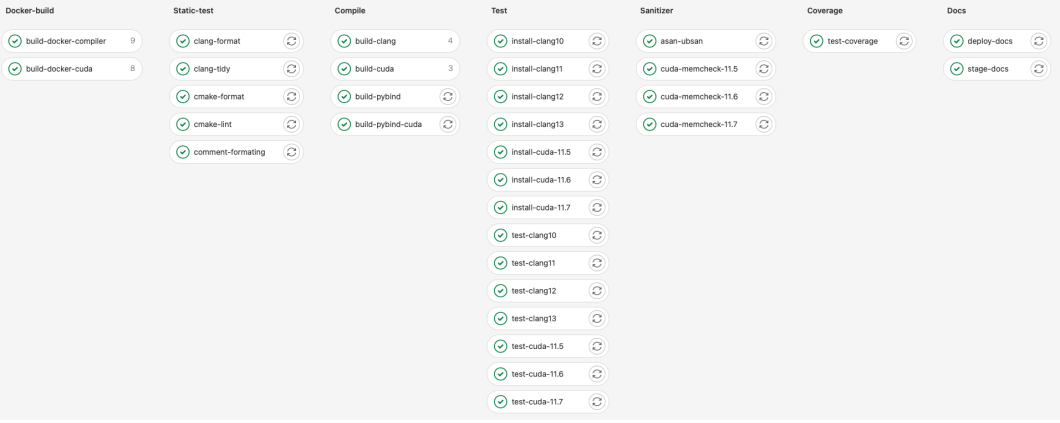
When you develop and push your changes to the git repository:

- you want to be sure that your code compiles successfully (ideally with different compilers)
- you want to be sure that all your tests pass

→ continuous integration runs automated builds/tests each time you push to the repository

- commonly used tools: Travis CI, Jenkins, GitHub Actions or GitLab runners
- steps often defined in a YAML file in the repository
- we use this (with Gitlab runners) for your homework!

CI: example



You can check this out in more detail yourself at `https://gitlab.com/tum-ciip/elsa`.

Summary