

PROGRAMMING IN C++

SHEET 0

Submission date: 21.09.2025

First Things First

Before you can start programming in C++, you need to install and set up your development environment. In this exercise, we'll guide you through the installation and setup process for all major operating systems.

Important: Please follow the provided order of operations and aim to complete this task **as soon as possible**. Some of you may encounter issues that could take time to resolve. Unfortunately, we won't have the opportunity to assist with these problems once the course has begun.

Windows-Installation

Install WSL 2 with Ubuntu 22.04:

Run as Admin: (Win+R (run) "cmd" Ctrl+Shift+Enter)

```
1 wsl --install -d Ubuntu-22.04
```

If already installed, make sure that you're running the latest version:

```
1 wsl --update
```

Reboot afterwards.

Make sure you're running WSL Version 2:

Run as Admin: (Win+R (run) "cmd" Ctrl+Shift+Enter)

```
1 wsl --set-version Ubuntu-22.04 2
```

If you encounter issues during the installation, have a look at [the official instructions](#). Also, make sure that you have *virtualization* support enabled in your BIOS.

You can now open your lightweight Ubuntu 22.04 installation from the start menu. On the first run, you will be asked to set a username and password. Also, install the latest updates using the following set of commands:

```
1 sudo apt-get update
2 sudo apt-get upgrade
```

You can access your "C:\\" drive as `/mnt/c/`. Do not directly work in this directory though. Instead, copy the projects to your home directory, e.g. like this:

```
1 mkdir ~/cpp_course
2 cp -r /mnt/c/path/to/the/unpacked/project/directory ~/cpp_course/
```

Now, follow the **Linux** guide for step **0.1** - **0.5** using this Ubuntu terminal.

Note: You can use right-click to paste into your Ubuntu Terminal.

Note: In general, we encourage you to try using a full Linux desktop. It's what we use every day.

Linux-Users:

In the following instructions, we assume a Ubuntu/Debian-based Linux. If you run another distribution, you may have to adjust package names and installation commands for the package manager.

0.1 Installing clang

C/C++ code needs to be translated into machine language by a compiler. The main two options are LLVM (`clang/clang++`) and GCC (`gcc/g++`). For cross-platform compatibility, we decided to use **clang-15**. This will make sure that compiler errors and the associated error messages are the same for everyone. (If not available, Clang ≥ 12 or GCC ≥ 10 are required for C++20 support.)



Linux In most Linux distributions you can install `clang` directly via your package manager.

```
1 sudo apt-get install clang-15 clangd-15
```

In addition to the compiler, you also need the C++ standard library and support for OpenMP:

```
1 sudo apt-get install libstdc++-12-dev libomp-15-dev
```

macOS Simply install XCode from the AppStore. On M1/M2-Macs, you will not have OpenMP. This means that you will have to test the multithreading exercise on a different machine.

0.2 Installing gdb

C++ code can be debugged using specialized debuggers, which allow the user to control the execution steps of the program. A debugger collects all sorts of useful debug information about the program state and allows the user to interactively explore the code. We will use `gdb` (The GNU Project Debugger). `gdb` itself is a command line program, which can be used independently from any editor. This can be useful, however, it is also hard to learn. We chose an editor that in the background uses `gdb` when the program is running in debug-mode.

It does not matter which `gdb` version you install, which makes the installation relatively simple.

Linux Any `gdb` version will do, so go for whatever version you get from your package manager. E.g. for Ubuntu:

```
1 sudo apt-get install gdb
```



macOS Skip this step. You can use `lldb` instead, which should already be installed as a part of XCode.

0.3 Installing ninja

In theory, a compiler and a text editor are all you need to develop C++ programs. However, using a compiler directly can become complicated quickly. Especially when the project must be compiled and linked from multiple source files. For this reason, there are *build systems*: Programs that not only call the compiler for you but also speed up compilation by automatically checking which parts of the project need to be recompiled and which parts can be reused from a previous build. The most popular build system is called `make`. For our exercises, we will however use `ninja`, which is a more modern, lightweight, and slightly faster build system.

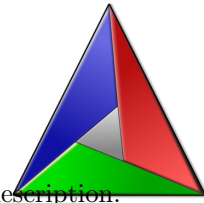
Linux You can install `ninja` from the package manager.

```
1 sudo apt-get install ninja-build
```

macOS Install `ninja` using `brew`.

0.4 Installing cmake

ninja still needs to know about how our projects are structured in order to build executables. We will use a `CMakeList.txt` file to explain how such an executable can be built from the source files in a rather generic, platform-independent way. **cmake** is a program that generates the more detailed descriptions of all the platform-dependent dependencies which are needed by **ninja** from this high-level description.



Linux You can install **cmake** from your package manager. E.g.

```
1 sudo apt-get install cmake
```

macOS Install **cmake** using **brew**.

Note: Please make sure that your installed **cmake** installation has **at least version 3.15** by running `cmake --version` on a shell.

0.5 Installing the Thread Building Blocks library

In one of the later exercises we will use constructs that rely on the Thread Building Blocks library **tbb**. You may have this library already installed on your system, if not install it like this:

Linux Use your package manager to install **tbb**.

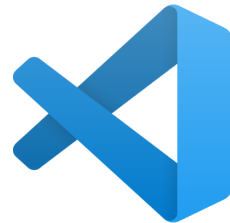
```
1 sudo apt-get install libtbb-dev
```

macOS Install the **tbb** library via **brew**: `brew install tbb`

0.6 Installing VisualStudio Code

All you need to develop C++ code is a simple text editor. However, a decent text editor with syntax-highlighting, linting, debugger interface, **cmake** plugin, ... can help to prevent common mistakes and take away some of the monotonous tasks. In this course, we will use VisualStudio Code, a simple yet modern and easily extensible editor, which is available for all common operating systems.

Download the matching installer for your system from <https://code.visualstudio.com/download> and install it.




Linux Download and install the **.deb** (Debian-based, e.g. Ubuntu) or **.rpm** (Red Hat-based) package. On Debian-based systems like Ubuntu, run `sudo apt-get install ./code-<version>.deb` to install the package to your system. Alternatively, the **.tar.gz** contains a standalone version of VisualStudio Code, which needs no installation. You can start the editor by running the `code` executable. (Open a terminal in the folder and run `./code`)



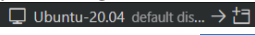
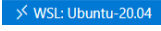
macOS Download the **.zip** file and install the VisualStudio Code.app that you find inside. You might have to change some security settings for this to work.

Windows Download and run the System Installer. You can also download and unpack the **.zip** version. In that case, directly run VisualStudio Code by running the `Code.exe` file.

0.7 Set up VisualStudio Code for the exercises

When VisualStudio Code is started first you are led through some setup steps.

- a) Select whatever theme you like  `Choose the look you want.`

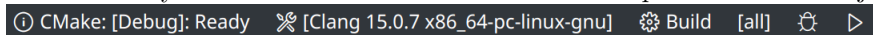
- b) In  click “Browse Language Extensions” and use the search field in the Extensions: Marketplace on the top left to install the following extensions:
WSL Windows only. Install this extension first. Select the Remote Explorer  extension on the left. Connect to WSL by clicking on the arrow of this icon on the newly opened WSL Targets menu:  In the bottom left corner, it will indicate that you are connected to WSL:  While connected, install the following extensions.

C/C++ The C/C++ for Visual Studio Code extensions adds some basic features to VisualStudio Code like the aforementioned debugging interface with `gdb`.

clangd This clangd plugin provides the typical features that you would expect from an IDE like code completion, jumping to definitions, code formatting and refactoring. Add the arguments `--clang-tidy` and `--header-insertion=never` in the settings of the extension. Mac users will also have to add the argument `--query-driver=/usr/bin/clang++`. (Otherwise, clangd will assume that you want to program in C, not C++).


CMake The CMake For VisualStudio Code plugin provides auto-completion when developing CMakeList.txt files, which could be useful during the exercises.

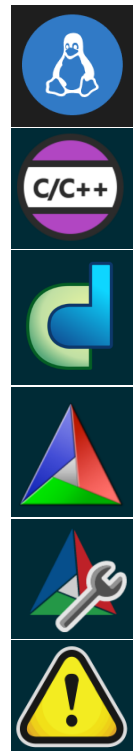
CMake Tools The CMake Tools plugin is what we will interact with whenever you want to configure, compile, debug or run your program. You will find it on the very bottom of the screen as soon as we open a CMake Project.



Error Lens Highlights errors in your code more clearly.

You can find the Extension Marketplace at any time on the left side or by pressing `Ctrl+Shift+X`.

- c) Next on the Welcome page is a hint  on how to open the *Command Palette*, namely via `Ctrl+Shift+P`. The *Command Palette* is where you find every single command that you can give to VSCode or one of the installed plugins. If you are searching for something you'll probably find it there.
- d) The last step is “Open up your code”. We will address this in the next subtask.



0.8 Importing a CMake project in VisualStudio Code

Open the project

The exercises will be handed out as CMake projects with a simple skeleton, which needs to be filled and extended to solve the exercise. Before you can do that you have to open the project. For this, you can just open the folder that contains the `CmakeList.txt` file of the provided skeleton.

You find this option in `File -> Open Folder...` or just by pressing `Ctrl+K` followed by `Ctrl+O`.

Open the `first_project` directory.

When using WSL, remember to copy the project directory from `/mnt/c/...` to your home directory first as explained above. If first compile the project in one place and copy the project directory afterward make sure to delete the `.cache/` directory. The files in there are used by the compiler to speed up compilation and can contain absolute paths to the old build-directory.

For our exercises, you can select that you do trust the authors. VSCode might automatically run some parts of the project (`cmake`), which could be problematic with malicious code.

Handling the Notifications

VSCode will automatically realize that this folder contains a CMake project and will give you some notifications, which will disappear rather quickly. You can reshown them by clicking the little *bell icon* on the bottom right to reshown them.

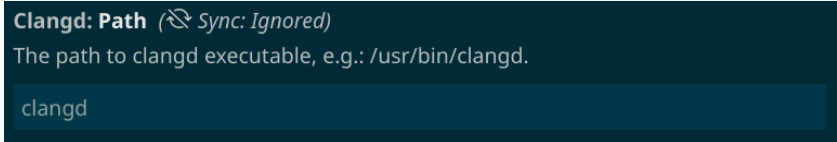
Most of those notifications will only show the first time you open one of the exercises due to some problems, that we will now fix.

You might get one or multiple of the following notifications:

You have both the Microsoft C++ extension and the clangd extension enabled... You can just disable IntelliSense. Clangd is way superior.

The 'clangd' language server was not found on your PATH... The clangd extension does not detect the installed clangd-15. Simply install the latest version when prompted.

On Linux, you can also set the correct path, e.g. clangd-15, in the settings (Ctrl + ,) instead:



Would you like to configure project 'first_project'? The Cmake Tools plugin asks you if it should directly run cmake to configure the project. We didn't set the kit yet, which refers to the toolkit that should be used. For C++ this means you first have to select the compiler. When agreeing to configure you will be presented with a dropdown menu to select the Kit - see the next section for details.

You might get some more notifications from VSCode during the exercises. Typically the suggestions given by VSCode are really good. Just make sure to read and think about what you agree to before doing so.

Selecting the Kit

In the bottom click **No Kit Selected**, or press Ctrl+Shift+P and search for **CMake: Select a Kit**. The presented drop down menu will show all compilers that are installed on your system. Please select the Clang version that you installed, e.g. **Clang 15.0.7**.

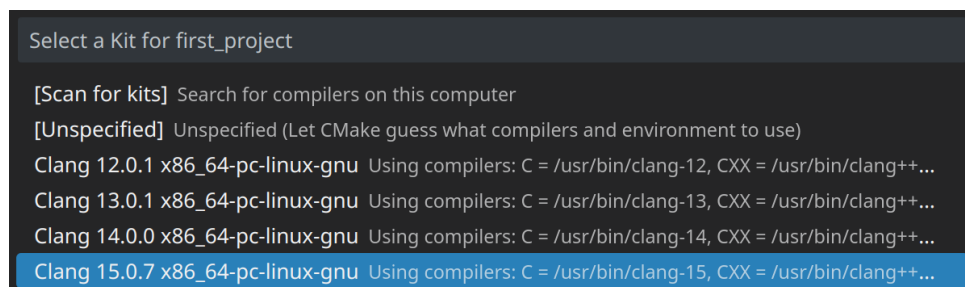


Figure 1: Selecting the installed Clang version from the list.

If Clang 15 does not show up in the list make sure that you follow the instructions from Task 0.1.

Building the project

Creating an executable by compiling and linking the source files is referred to as “building”. At this point, you should be able to build the project by either

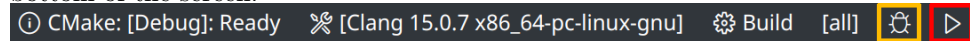
- Click the **Build** button on the bottom next to the *gear icon*.
- Press Ctrl+Shift+P, search for and run **CMake: Build**
- Press F7

The first time, or whenever the CMakeLists.txt was changed, you do this you will first see the output messages from cmake. You will also see the ninja output when building for the first time, or whenever any of the source-files were changed. If you build again when no changes have occurred, you will get something like this:

```
[build] ninja: no work to do.
```

Running the program

Building the project will create an executable file, which you could find in the `build/` directory that was generated automatically when building the project. For the first project, the executable will be called `first_project`. You can run the executable from a terminal if you want to. You can also use the CMakeTools plugin for VSCode to directly launch the program. You find the CMakeTools plugin at the bottom of the screen:



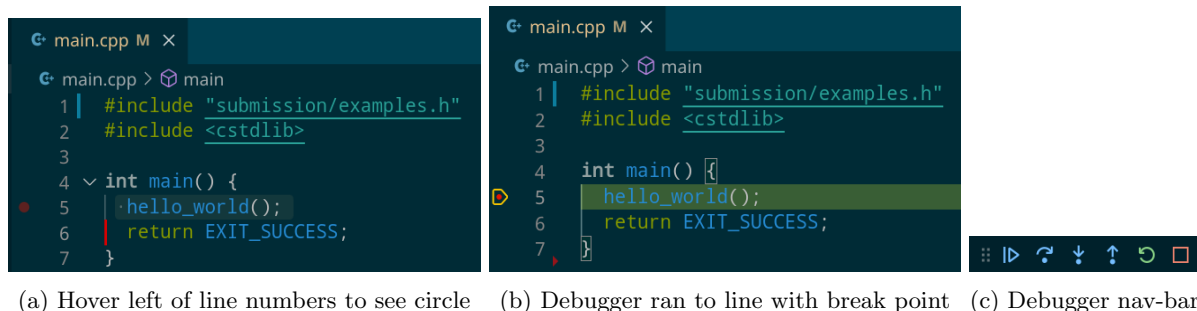
To run the program click the **play** button or `Shift+F5`. VSCode will show a **Terminal** tab and execute the executable. For the `first_project` you should see a `Hello World!` that is printed.

Running the program in debug mode

You can also run the code in debug mode. To do this click the **bug icon**. This will put VSCode into debug-mode and run the program in `gdb` (the debugger).

If you have `lldb`, the debugger from `llvm` installed it could happen that VSCode detects `lldb` as debugger instead of `gdb`. However the quick debug feature does not support `lldb`. The easiest way to fix this is to **uninstall** `lldb`.

If you didn't mark any break-points and there are no problems in your code that will cause the program to be stopped by the OS or `gdb` the program will just run through as if you ran it regularly.



(a) Hover left of line numbers to see circle (b) Debugger ran to line with break point (c) Debugger nav-bar

Figure 2: Debug interface

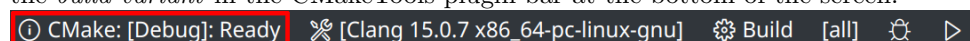
You can set *break points* by clicking the red circle that appears when you hover a line of code left to the line-numbers as shown in Figure 2a.

By clicking the red circle you specify a break-point. If you now run the program again in debug mode the debugger will interrupt the program before executing the line for which the breakpoint was set. In VSCode the line that would be executed next is marked as shown in Figure 2b.

The program execution is now paused. VSCode now shows you a navigation bar that is shown in Figure 2c. From left to right: Resume program until next break-point is encountered. Execute everything related to the current line and stop before executing the next line. Step into the scope of the function at the currently marked line. Step out of the current scope and get back to the parent. Restart the program in debug mode. Stop debugging.

All those buttons have shortcuts, which come in handy if you ever have to debug a more complex program. In debug-mode VSCode shows additional information about the current state of the program on the left side. This includes the values of all currently defined variables, watch-expressions, the current call-stack and a list of all active breakpoints. For the simple `first_project` there will be nothing to see here.

Debugging is only possible if the executable was built in the *Debug build variant*. The debug-symbols that are used by `gdb` are only included in the executable if the compiler is told to do so. You can select the *build variant* in the CMakeTools plugin bar at the bottom of the screen:



You might get warnings (...file not found...) when trying to step into some functions that are provided by standard libraries. This is expected and can only be fixed by installing the libraries with built-in debug-symbols. You can assume that problems are caused by wrong usage of the libraries, not by bugs in those libraries.

We will test your submissions in the *Release* build variation, which is optimized for speed. For development, we recommend you to use the *Debug* variant as it allows you to analyze problems in debug mode if needed.

0.9 Register on InfoMark

Note: Please check the ILIAS page for the InfoMark link regularly. It should be online about two weeks before the course starts. You can only finish this part once InfoMark is online. The exercise submissions will be managed via the **InfoMark** platform at <https://handinabi.cs.uni-tuebingen.de>. InfoMark is a submission management system, that automatically provides feedback from a set of unit-tests. So whenever you upload a solution it will be tested for correctness and you will immediately see the results. This can help you identify obvious mistakes in your code.

Please be aware that you'll only see the test results of some very simple sanity checks. The system will also run more thorough tests in the background, whose results are only shown to the tutors.

Create an account on InfoMark by registering on **InfoMark**. Please note that this InfoMark setup is independent of the one used for other courses. Please stick to the following rules:

- If possible register with your **student email address**. This should be something like this *first_name.last_name@student.uni-tuebingen.de*.
- As study subject use the English version. We need this to send your grade to the correct examination office in the end:

Informatik Computer Science

Medieninformatik Media Informatics

Bioinformatik Bio Informatics

Kognitionswissenschaften Cognitive Science

Medizininformatik Medical Informatics

Informatik Lehramt Computer Science Teaching Degree

Physik Physics

Mathematik Math

Other Use official English terms, all words starting with a capital letter. Avoid typos.

After registration, you will receive an email. You have to **confirm your account** before you can use it.

0.10 Enroll for the C++ course

Please enroll in our C++ course. This is the only course there currently is, so it should be straightforward. After enrolling you can show the content of the course.

There will be no dedicated exercise-groups for this course, so you don't have to touch the group preferences.

You will find additional material and exercise sheets here as soon as they are made available.

0.11 Submit Something

As you can see Sheet_00 is available for download. This will give this very pdf file alongside with the `first_project` directory and the `student_template_0.12` directory. We will now use the code in `student_template_0.12` to practice submitting to InfoMark and to understand the feedback provided to you by the system.

- Find the submission directory in `student_template_0.12`.
- Compress **only** the submission directory to a `submission.zip` file.
- On InfoMark scroll to Sheet_00 and click Show.
- On Task 0.12: Submit Something click Show.

- Upload your **submission.zip** in the **Submission** field either via drag and drop or via **Pick file**. Click **Upload** to upload your submission.
- You will see the message **submission received and will be tested**. Wait until you see the test results (~ 1 Minute). You should get something similar to this:

```

1 ~~~~~
2 test_program is a Catch v2.13.6 host application.
3 Run with -? for options
4
5 -----
6 Addition function is not correct!
7 -----
8 ../test.cpp:7
9 .....
10
11 ../test.cpp:8: FAILED:
12   REQUIRE( add(1, 1) == 2 )
13 with expansion:
14   0 == 2
15
16 =====
17 test cases: 1 | 1 failed
18 assertions: 1 | 1 failed

```

Apparently there is something wrong in the code. Lets go through it step step by step:

Line 1 - 4 This is the header. It tells us that the **test_program** that is doing the testing uses the **Catch** library.

Line 6 This is the conclusion that is drawn from the error that occurred. **Addition function is not correct!** means that there is apparently something wrong in the addition function that is tested.

Line 8 This is the source-file (**test.cpp**) and line in which the test case was defined (7). You have no influence on that, the tests are defined by us.

Line 11 This is the exact line in which an assertion failed. An assertion is an expectation that we formulate when testing a function.

Line 12 This is the actual assertion that we wrote down. This assertion failed. **REQUIRE(add(1, 1) == 2)** means that it is necessary for the function call **add(1, 1)** to yield 2.

Line 13, 14 This is what actually happened in the **REQUIRE()**. The assertion failed because **add(1, 1)** returned 0, but we expected $1 + 1 = 2$.

Line 17 A summary over all test cases. Here 1/1 test case failed.

Line 18 A summary over all assertions. Here 1/1 assertion failed.

- b) Let's see what else can happen. First, open the directory **exercise00** in VSCode as described before. You can run the program by selecting the Kit (**clang**) and running the program. You should see the output $1 + 1 = 0$ in the **TERMINAL** tab.

In the **Explorer** **Ctrl+Shift+E** open the file **submission->exercise_00.cpp**.

In line 13 replace **"return 0;"** with **"return asdf;"**.

If you now try to build the project you will get an error in the **OUTPUT** tab from the compiler, telling you **error: use of undeclared identifier 'asdf'**. This message might be in another language for you, depending on the locale settings of your system. You will also see warnings that **'a'** and **'b'** are **unused parameters**. Those warnings have been there before already and are a good indication that something is wrong. So keep an eye out for the **OUTPUT** tab when compiling your code.

Now let's see what you get when uploading this broken state of the submission to InfoMark. Create another .zip file with the current state and upload it. You should get the following response from the tests:

```

1 InfoMark could not compile the test project.
2 This means ninja reported a compiler or linker error.
3 Please check the following ninja log and fix the error before uploading.
4
5 [ninja output:]
6 [1/4] Building CXX object CMakeFiles/submission.dir/submission/exercise_00.cpp.o
7 FAILED: CMakeFiles/submission.dir/submission/exercise_00.cpp.o
8 /usr/bin/clang++ -Dsubmission_EXPORTS -I../lib -I../include -O3 -DNDEBUG
   -flto=thin -fPIC -Wall -Wextra -Wshadow -Wold-style-cast -Wcast-align
   -Wunused -Woverloaded-virtual -Wpedantic -Wconversion -Wsign-conversion
   -Wnull-dereference -Wdouble-promotion -Wformat=2 -Wmisleading-indentation
   -fopenmp=libomp -std=gnu++20 -MD -MT
   CMakeFiles/submission.dir/submission/exercise_00.cpp.o -MF
   CMakeFiles/submission.dir/submission/exercise_00.cpp.o.d -o
   CMakeFiles/submission.dir/submission/exercise_00.cpp.o -c
   ../submission/exercise_00.cpp
9 ../submission/exercise_00.cpp:12:12: error: use of undeclared identifier 'asdf'
10     return asdf;
11         ^
12 ../submission/exercise_00.cpp:5:13: warning: unused parameter 'a'
13     [-Wunused-parameter]
14     int add(int a, int b) {
15         ^
16     ../submission/exercise_00.cpp:5:20: warning: unused parameter 'b'
17     [-Wunused-parameter]
18     int add(int a, int b) {
19         ^
20 2 warnings and 1 error generated.
21 [2/4] Building CXX object CMakeFiles/test_program.dir/test.cpp.o
22 ninja: build stopped: subcommand failed.

```

So InfoMark gives you the same compiler errors that you have seen when trying to compile the code. In line 1-3 you are kindly requested to fix the error. **We will not grade submissions that do not compile**, therefore make sure to at least make the program compile. A response like the one shown above in your final submission will give you 0 points for the task. If you have really no idea what a function should do just return a constant so your submission compiles, as was done here in the `add()` function.

- c) From your experience in task a) you know that the test for the `add()` function checks whether the returned value is 2.

Lets try to trick InfoMark: Change `"return asdf;"` to `"return 2;"`, save, zip and upload.

And indeed you will get the output:

```

1 =====
2 All tests passed (1 assertion in 1 test case)

```

However **this will not help you** succeed with the exercises. As mentioned before, InfoMark runs hidden tests that are only shown to the tutors. For the same submission, a tutor would still see e.g. something like this:

```

1 ...
2 REQUIRE( add(2, 1) == 3 )
3 with expansion:

```

```
4 2 == 3
5 ...
```

d) Finally lets fix the function by replacing “`return 2;`” with “`return a + b;`”.

Compiling this version should get rid of all warnings in the `OUTPUT` tab. Upload this final version to InfoMark. With this submissions now both you and the tutor will see that all tests passed.