

Setup & Debugging Guide für CLion

Oliver Stoll

Dieser Guide soll als Hilfestellung für die Einrichtung von CLion und als kurzer Überblick über die wichtigsten Funktionen dienen. IDEs sind sehr mächtige Tools, die euch den Entwicklungsprozess stark erleichtern können, wenn ihr Zeit investiert zu lernen sie richtig einzusetzen.

1 CLion downloaden

Installiere CLion von <https://www.jetbrains.com/clion/download>.

Ihr könnt zunächst die 30-Tägige Testversion nutzen, oder direkt die Gratis Educational License auf <https://www.jetbrains.com/shop/eform/students> mit eurer TU-Email beantragen.

Tipp: Sämtliche Konsolen-Befehle aus diesem Guide könnt ihr direkt im Terminal von CLion ausführen, nachdem ihr ein anfängliches Projekt erstellt habt.

2 Entwicklungstools installieren

In diesem Bereich installieren wir alle weiteren notwendigen Tools für die C-Entwicklung. Teilweise sind diese Schritte nur für Windows- oder Mac-User relevant, dies wird jeweils angegeben.

WSL installieren (nur Windows)

Installiere WSL (Windows Subsystem for Linux) von <https://learn.microsoft.com/de-de/windows/wsl/install>.

Jetzt könnt ihr mit `wsl` eure Konsole in eine Linux-Konsole umwandeln und alle weiteren Konsolenbefehle damit ausführen.

Brew installieren (nur Mac)

In nächsten Punkt werden wir Pakete mit `apt` installieren. Dies ist ein Paketmanager für Linux, der auf Mac nicht funktioniert. Als Mac-User müsst ihr stattdessen `brew` verwenden.

Installiert `brew` mit folgendem Terminal-Befehl (ohne den Zeilenumbruch):

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

C-Tools installieren (Alle)

Installiert alle notwendigen Tools für die C-Entwicklung mit folgendem Terminal-Befehlen:

```
sudo apt update
sudo apt upgrade
sudo apt install -y build-essential cmake clang gdb g++
```

CLion Toolchain einstellen (nur Windows)

Nachdem ihr WSL, CMake und GDB installiert habt, müsst ihr noch WSL als Toolchain in CLion einstellen. Dies könnt ihr unter **File > Settings > Build, Execution, Deployment > Toolchains** machen.

Klickt auf das +-Symbol und wählt WSL aus. Hier sollten nun nach etwas Zeit "Build Tool, C Compiler, C++ Compiler, Debugger" erkannt werden.

Nun könnt ihr eure vorherige Toolchain (MinGW) mit dem --Symbol entfernen.

Nun müsst ihr nurnoch unter **Build, Execution, Deployment > Build Tools > Make** den Pfad zu eurem wsl **make** hinzufügen ("`//wsl$/Ubuntu/usr/bin/make`", abhängig von eurer wsl Distribution).

3 Projekt einrichten

In diesem Abschnitt richten wir euer Projekt für die Entwicklung ein.

Tipp: Sämtliche Tool-Windows könnt ihr mit Shortcuts (Project: **Alt+1**, Debug: **Alt+5**, Terminal: **Alt+F12**) öffnen und wieder schließen. Hier könnt ihr genauso wie in einem anderen Terminal **wsl** verwenden

Projekt erstellen

Erstellt, falls ihr das noch nicht getan habt, ein neues Projekt in CLion mit **New Project**. Wählt hierfür **C Executable** aus und gebt eurem Projekt einen Namen. Fügt nun Vorgabe-Dateien über das **Project-Window** (**Alt+1**) eurem neu erstellten Projekt hinzu.

Code ausführen (mit Argumenten)

In eurem Code könnt ihr nun links neben der **main**-Funktion auf den grünen Pfeil klicken, um euer Programm auszuführen. Dies erstellt automatisch eine Run-Konfiguration, die ihr oben rechts im Dropdown-Menü oder unter **Run > Edit Configurations** einsehen könnt. In dieser Run-Konfiguration könnt ihr nun zusätzlich die beim Aufruf übergebene Argumente unter **Program Arguments** angeben.

CLion buildet zum ausführen ein eigenes executable im internen build directory (standardmäßig **cmake-build-debug**), und speichert die Regeln zum builden in der Projekt-internen **CMakeLists.txt**.

Achtung: Dieses interne build directory müsst ihr für Systemprogrammierung in der Einstellungen unter CMake zu **build** umbenennen, um die so kompilierten Dateien abgeben zu können.

Externe Executables ausführen

Falls ihr ein bereits gebuildetes executable ausführen & debuggen möchtet, wie beispielsweise einen mit **make all** gebuildeten Test, könnt ihr dies auch mittels einer Run-Konfiguration tun. Wählt dafür in **Run > Edit Configurations > Add new Configuration > Custom Build Application** das gebuildete executable als **Executable** aus. Nun könnt ihr noch das **Working Directory** auswählen (vermutlich euer Projektordner), ein build target angeben (**make all**) und Argumente übergeben.

Windows: Diese Run-Konfiguration könnt ihr auch nutzen, ihr kein echtes (build-)Target ausgewählt habt. Dafür müsst ihr nur ein 'leeres' Platzhalter Target erstellen. So buildet die Run-Konfiguration euren Code allerdings nicht direkt neu. Ihr müsst also bei Code-Änderungen immer erst erneut **make all** ausführen.

Git einrichten

Selbstverständlich könnt ihr auch Git direkt in CLion nutzen, um euer Projekt mit einem Klick upzudaten, zu committen und zu pushen oder zwischen Branches zu wechseln. Wählt hierfür **VCS > Share Project on GitHub** um ein neues Git Projekt zu erstellen oder **VCS > Project from VCS** um ein vorhandenes Projekt zu clonen, und folgt dem Login-Flow um euren GitHub Account mit CLion zu verknüpfen.

Nun sollte im linken Dropdown-Menü der IDE neben eurem Projektnamen ein Git-Symbol mit dem Namen eures Branches erscheinen. Klickt hierauf für alle Git-Optionen, und verwendet den Shortcut **Ctrl+K** um eure Änderungen zu comitten & zu pushen (auch direkt mit **Ctrl+Shift+K** möglich).

4 Debuggen

In diesem Kapitel lernen wir die Funktionsweise des interaktiven Debugger von CLion kennen

Breakpoints setzen & Debugger starten

Jetzt könnt ihr euer Programm auch debuggen. Nutzt dafür einfach die gleiche Run-Konfiguration mit dem kleinen grünen Käfer, um euer Programm zu starten. Das **Debug-Window** (**Alt+5**) sollte sich öffnen und euer Programm stoppt nun bei Breakpoints, die ihr in eurem Code gesetzt habt. Diese könnt ihr mit einem Klick auf eine Zeilennummer in eurem Code erstellen / löschen. Für komplexeres Debugging könnt ihr mit Rechtsklick euren Breakpoints auch eine Condition hinzufügen (nützlich in einem Loop, wo ein Breakpoint häufig irrelevant ist).

Debug-Fenster

Sobald der Debugger an einem Breakpoint gehalten hat, seht ihr in eurem **Debug-Fenster** den aktuellen Zustand eurer **Variablen**, den **Stacktrace** (welche Funktionen voneinander aufgerufen wurden, unten ist der äußerste Funktionsaufruf) und in der integrierten **Konsole** den Output eures Programms.

Hier empfiehlt es sich, den Konsolen-Tab mit Threads & Variablen zu vereinen, um alles auf einmal sehen zu können (**Rechtsklick auf Header -> Layout > GDB & Memory View disable**n, und dann Console-Tab auf rechte Seite ins Debug-Fenster ziehen). So seht ihr alle Informationen die ihr mit **Run** auch hättet, und könnt den Debugger effektiv fast immer verwenden.

Stacktrace nutzen (Multithreading)

Mithilfe des **Stacktraces** könnt ihr nicht nur den Zustand der aktuellen Funktion des Main-Threads, sondern der von allen voneinander aufgerufenen Funktionen, und sogar der von gehaltenen Threads betrachten. Wählt dafür im Dropdown-Menü den gewünschten Thread, und darunter eine Funktion aus dem Stacktrace aus. Die entsprechenden lokalen Variablen werden so angezeigt.

So können wir auch komplexere Programme mit mehreren Threads debuggen.

Fehler Zurückverfolgen

Nun könnt ihr mit **Step Over** nach und nach einzelne Zeilen von eurem Code (bzw. mit **Step into** zusätzlich auch dem Code verwendeter libraries) ausführen, oder mit **Resume Program** euer Programm bis zum nächsten Breakpoint weiterlaufen lassen. Alternativ könnt ihr auch **Pause Program** nutzen um euer Programm zu einem beliebigen Zeitpunkt zu pausieren.

Auch wenn Fehler, wie beispielsweise ein Segmentation Fault auftreten, hält der Debugger an der Code-Zeile des Fehlers und zeigt euch den letzten Zustand der **Variablen** eures Programmes, der zu dem Fehler geführt hat.

So könnt ihr nachverfolgen welche Variablen einen anderen Zustand haben als ihr erwartet hättet, und diese Änderungen in eurem Code logisch 'rückwärts' zurückzuverfolgen.

Da wir leider in der Code-Ausführung nicht rückwärts gehen können, wird dies bei komplexeren Fehlern wahrscheinlich ein mehrfaches Ausführen des Programms erfordern, wobei wir mit jeder Ausführung neue Erkenntnisse gewinnen und unsere Breakpoints weiter nach vorne setzen können.

Fehlermeldungen Lesen

Auch wenn viele Fehlermeldungen auf den ersten Blick kryptisch wirken können, enthalten sie meist alle Informationen um die Quelle des Problems stark einzugrenzen. Achtet darauf welche Klassen/Funktionen an einem Fehler beteiligt sind. Diese können entweder direkt selbst Fehler enthalten, oder falls nicht, bekommen sie vermutlich fehlerhafte Daten zum verarbeiten geliefert. Im Zweifelsfall führt euch eine Suche nach der Fehlermeldung schnell zu einen Stack-Overflow Post, der euch das Problem weiter erklärt.