

Setup & Debugging Guide für Visual Studio Code

Oliver Stoll

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

1 VSCode downloaden

Installiere VSCode von <https://code.visualstudio.com/download>.

Tipp: Sämtliche Terminal-Befehle aus diesem Guide könnt ihr direkt im Terminal von VSCode 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>.

Wenn ihr nun VSCode öffnet, werdet ihr gefragt ob ihr die wsl-Integration installieren wollt. Klickt ja und verbindet euer Terminal mit wsl. So könnt ihr direkt alle Konsolen-Befehle im VSCode Terminal über wsl ausführen. Ihr solltet nun unten rechts *WSL:;Euer-Distro* sehen.

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 daher statt **sudo apt** nachfolgend **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++
```

3 Projekt einrichten

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

Tipp: Sämtliche Tool-Windows könnt ihr mit Shortcuts (Explorer: `Ctrl+Shift+E`, Debug: `Ctrl+Shift+D`, Terminal: `Ctrl+Shift+ö`) öffnen.

Projekt öffnen / erstellen

Erstellt, falls ihr noch keines habt, ein leeren Ordner als neues Projekt, oder öffnen ein Ordner als Projekt über den *Explorer*. Unter Windows müsst ihr den Ordner nun erneut mit wsl öffnen. Nutzt dazu den Command-Shortcut `Ctrl+Shift+P` und sucht nach *Remote-WSL: Reopen Folder in WSL*. Außerdem müsst ihr für Windows die *C/C++ Configuration* unten Rechts im Status-Bar auf Linux ändern (diese müsst ihr zunächst erstellen, dafür einfach unter *Add Configuration* Linux eingeben).

Code ausführen (mit Argumenten)

Nachdem ihr die C/C++ Extension installiert habt, könnt ihr nun euren Code ausführen. In eurem Code-Fenster könnt ihr nun oben rechts auf den grünen Pfeil klicken, um euer Programm auszuführen. Dies buildet euer executable allerdings **nicht automatisch** im build Ordner. Wenn ihr Argumente übergeben möchtet, benötigt ihr eine Run-Configuration sowie eine Build-Configuration welche im Ordner `.vscode` als `launch.json` sowie `tasks.json` angegeben werden. In der Build-Configuration kann auch gleich angegeben werden, wo die Executables gebuildet werden sollen.

Eine beispielhafte `launch.json` & `build.json` die CMake nutzen sind nachfolgend angegeben, die ihr in euer Projekt kopieren könnt.

Unter *Run & Debug* könnt ihr damit eure Run-Configuration auswählen und ausführen.

Listing 1: `.vscode/launch.json`

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "DEBUG WEBSERVER",
      "request": "launch",
      "program": "${workspaceFolder}/build/webserver",
      "args": [],
      "cwd": "${fileDirname}",
      "type": "cppdbg",
      "MIMode": "gdb",
      "preLaunchTask": "BUILD WEBSERVER (CMAKE)"
    }
  ]
}
```

Listing 2: .vscode/tasks.json

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "BUILD WEBSERVER (CMAKE)",
      "command": "cmake -B build -D CMAKE_BUILD_TYPE=Debug
        && make -C build",
      "type": "shell",
    }
  ],
}
```

Externe Executables ausführen

Falls ihr ein bereits gebuildetes executable ausführen & debuggen möchtet, wie beispielsweise einen mit `make all` gebildeten Test, könnt ihr dies auch mittels einer Run-Configuration tun. Wählt dafür das gebildete executable als **"program"** aus. Nun könnt ihr noch ein build task angeben und Argumente übergeben.

Git einrichten

Selbstverständlich könnt ihr auch Git direkt in VSCode nutzen, um euer Projekt mit einem Klick upzudaten, zu committen und zu pushen oder zwischen Branches zu wechseln. Wählt hierfür **Source Control > Publish to GitHub** um ein neues Github Projekt zu erstellen oder in der Konsole `git clone <url>` um ein vorhandenes Projekt zu clonen.

Unter Source Control (Ctrl+Shift+G) findet ihr nun alle Git-Operationen die ihr benötigt.

4 Debuggen

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

Breakpoints setzen & Debugger starten

Jetzt könnt ihr euer Programm debuggen. Das **Debug-Window** (**Alt+5**) sollte sich öffnen sobald euer Programm bei einem Breakpoint stoppt, die ihr in eurem Code gesetzt habt. Diese könnt ihr mit einem Klick links neben 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 **Variables**, den **Call Stack** (welche Funktionen voneinander aufgerufen wurden, unten ist der äußerste Funktionsaufruf) und in dem integrierten **Terminal** den Output eures Programms.

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.

Call Stack nutzen (Multithreading)

Mithilfe des **Call Stacks** 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 Call Stack aus. Die entsprechenden lokalen Variablen werden so angezeigt.

So können wir auch komplexere Programme mit mehreren Threads debuggen.

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.