

# Programmieren in C++

---

Christian Lang (Lac)

20. September 2019

# Build-System

---

- Kompilierung im Vergleich
- Build-Probleme
- Build-Tools
- CMake
- Tool Integration
- CMake Targets
- Kompilierung ausführen
- Build-Systeme sind produktiver Code

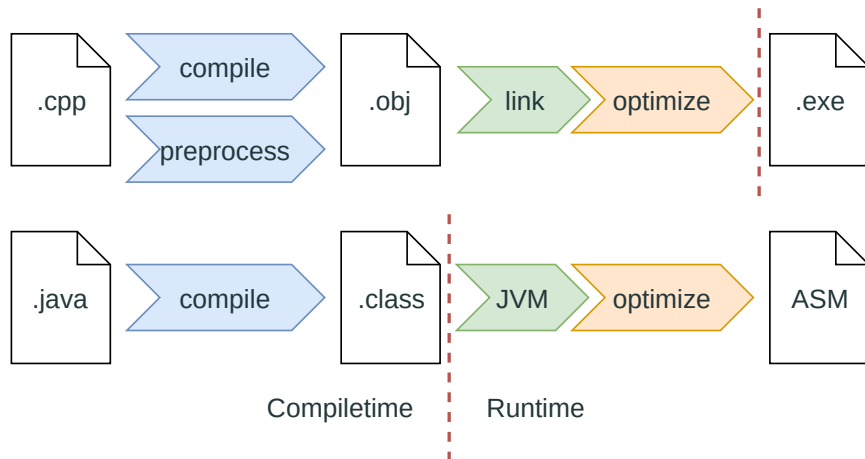
## Java

- kompiliert zu Bytecode
- Multipass Compiler
- Optimierungen zur Runtime
- Linking zur Runtime

## C/C++

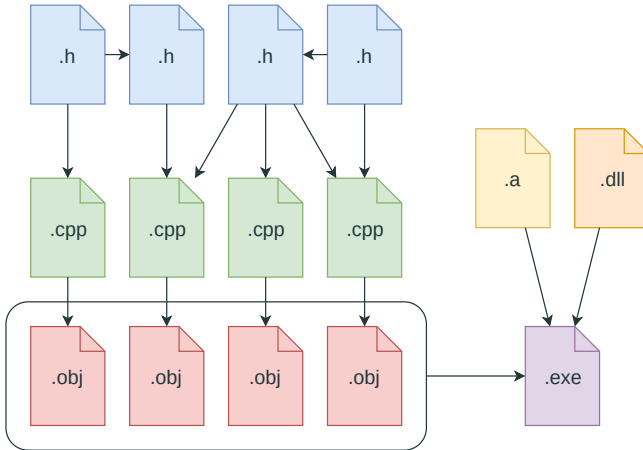
- kompiliert zu Objectcode
- Onepass Compiler Design (In Realität aber Multipass)
- Optimierungen zur Compiletime
- Linking zur Compile- oder Runtime

# Kompilierung im Vergleich

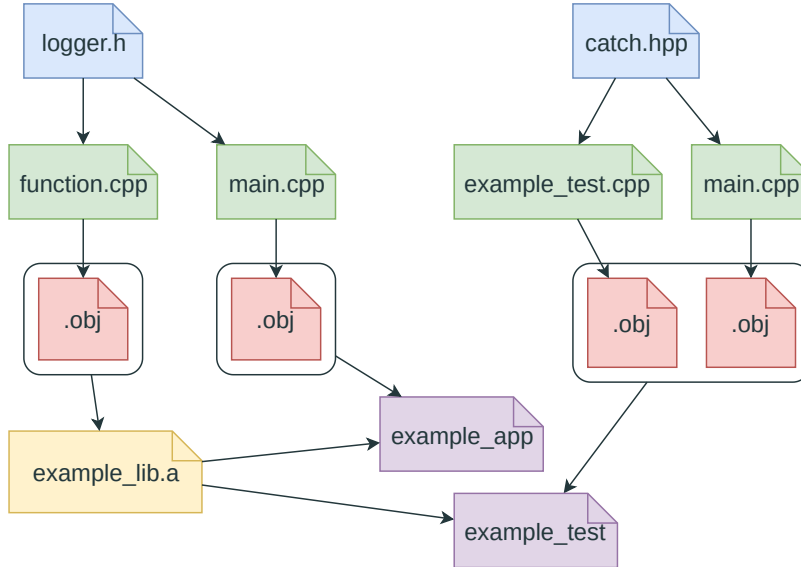


# Build-Probleme

- Inkrementelle Builds
- Korrekte Binaries
- Korrekte Abhängigkeiten



## Beispiel: example-Projekt



## Compiler

- gcc
- clang/llvm
- msvc

## Build-Systeme

- make
- ninja
- Visual Studio

## Meta-Build-Systeme

- cmake
- meson



- Definiert **High-Level**-Abhängigkeiten
- Generiert **rohes** Build-System
- rohes Build-System **überwacht** Änderungen
- Modernes CMake ab **3.x**

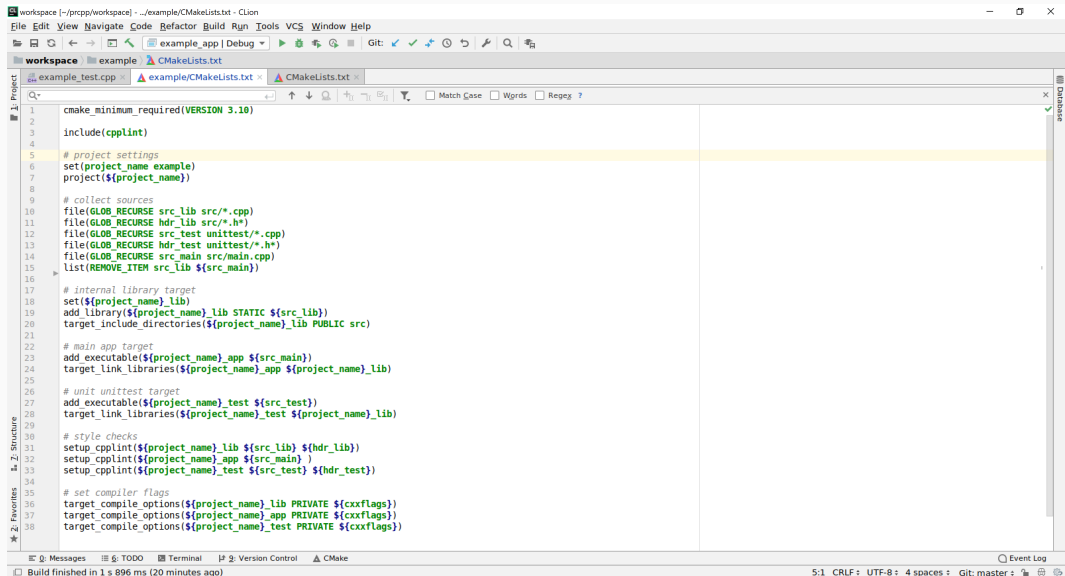
## Links

- [An Introduction to Modern CMake](#)
- [Effective Modern CMake](#)
- [C++Now 2017: Daniel Pfeifer - Effective CMake](#)

# CMake: example-Projekt

```
1  cmake_minimum_required(VERSION 3.10)
2  project(example)
3
4  # collect sources
5  file(GLOB_RECURSE src src/*.cpp)
6
7  # main app target
8  add_executable(example_app ${src})
9  target_link_libraries(example_app
10     PRIVATE pthread)
11
12  # define compile flags (warnings are fatal / add more checks)
13  set(cxxflags -Werror -Wall -Wextra -Wconversion -Wpedantic)
14
15  # set compiler flags
16  target_compile_options(example_app PRIVATE ${cxxflags})
```

# CMake: example-Projekt



The screenshot shows an IDE window titled "workspace [-/prcpp/workspace] - .../example/CMakeLists.txt - CLion". The menu bar includes File, Edit, View, Navigate, Code, Refactor, Build, Run, Tools, VCS, and Window Help. The toolbar contains icons for file operations, navigation, and development tools. The main editor displays the CMakeLists.txt file with the following content:

```
1 cmake_minimum_required(VERSION 3.10)
2
3 include(cppLint)
4
5 # project settings
6 set(project_name example)
7 project(${project_name})
8
9 # collect sources
10 file(GLOB_RECURSE src_lib src/*.cpp)
11 file(GLOB_RECURSE hdr_lib src/*.h*)
12 file(GLOB_RECURSE src_test unittest/*.cpp)
13 file(GLOB_RECURSE hdr_test unittest/*.h*)
14 file(GLOB_RECURSE src_main src/main.cpp)
15 list(REMOVE_ITEM src_lib ${src_main})
16
17 # internal library target
18 set(${project_name}_lib)
19 add_library(${project_name}_lib STATIC ${src_lib})
20 target_include_directories(${project_name}_lib PUBLIC src)
21
22 # main app target
23 add_executable(${project_name}_app ${src_main})
24 target_link_libraries(${project_name}_app ${project_name}_lib)
25
26 # unit unittest target
27 add_executable(${project_name}_test ${src_test})
28 target_link_libraries(${project_name}_test ${project_name}_lib)
29
30 # style checks
31 setup_cpplint(${project_name}_lib ${src_lib} ${hdr_lib})
32 setup_cpplint(${project_name}_app ${src_main} )
33 setup_cpplint(${project_name}_test ${src_test} ${hdr_test})
34
35 # set compiler flags
36 target_compile_options(${project_name}_lib PRIVATE ${cxxflags})
37 target_compile_options(${project_name}_app PRIVATE ${cxxflags})
38 target_compile_options(${project_name}_test PRIVATE ${cxxflags})
```

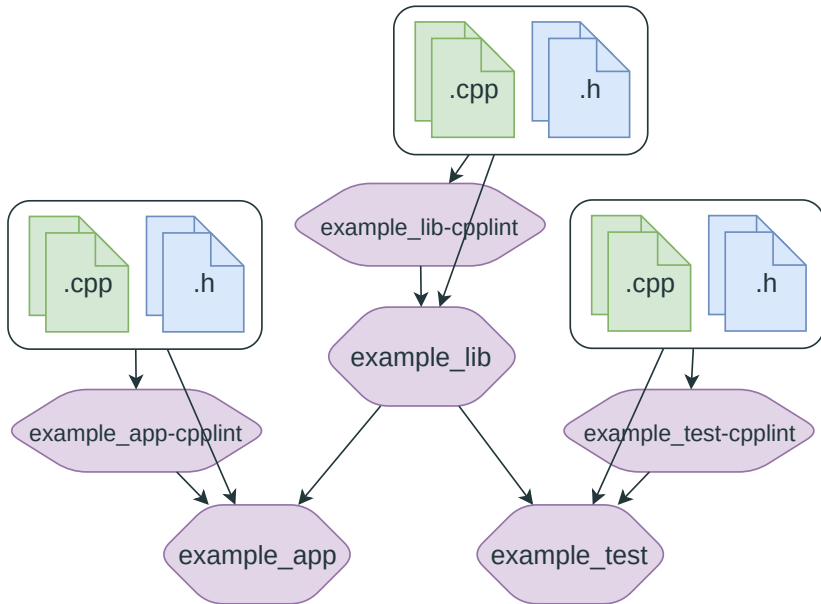
The IDE interface includes a sidebar on the left with "Project" and "Structure" views, and a bottom status bar showing "Build finished in 1 s 896 ms (20 minutes ago)". The status bar also displays "5:1 CRLF : UTF-8 : 4 spaces : Git: master :".

- CMake erlaubt Custom-Commands und Custom-Targets
- Ähnlich wie Compile-Schritt
- Kann zu Abbruch des Builds führen
- Abhängigkeiten in beide Richtungen möglich

```
1 add_custom_command(  
2     OUTPUT main.cpp.timestamp  
3     COMMAND cpplint.py main.cpp  
4         && touch main.cpp.timestamp  
5     DEPENDS main.cpp  
6     COMMENT "Linting with cpplint: main.cpp")  
7  
8 add_custom_target(example_app-cpplint  
9     DEPENDS main.cpp.timestamp
```

- **Vereinen** Kommandos
- **Abhängigkeiten** untereinander
- Sichtbar oder intern
- **Default** Target (je nach Build-System)

# CMake Targets



- Out-of-Source Builds
- Keine Durchmischung von Quell-Daten und Build-Artefakten

```
1 cd example
2 mkdir build
3 cd build
4 cmake ..
5 make -j4
```

- IDEs wie CLion abstrahieren diesen Prozess
- erzeugen z.B: cmake-build-debug

# Build-Systeme sind produktiver Code

- Das Build-System ist **Teil** des Produktes
- Muss **versioniert** werden
- Kann ebenfalls mit Funktionen etc. strukturiert werden:

```
1 include(prcpp_code)
2 prcpp_code(mein_projekt)
```

## Vorgehen im Modul prcpp

- Im Arbeitsblatt sollen Sie **einmalig** ein eigenes cmake-Projekt aufsetzen
- Verwenden Sie danach nur noch die vorgefertigte **prcpp\_code** Funktion
- Achten Sie auf die Datei-Struktur mit **src** und **unittest** Ordner