CMake

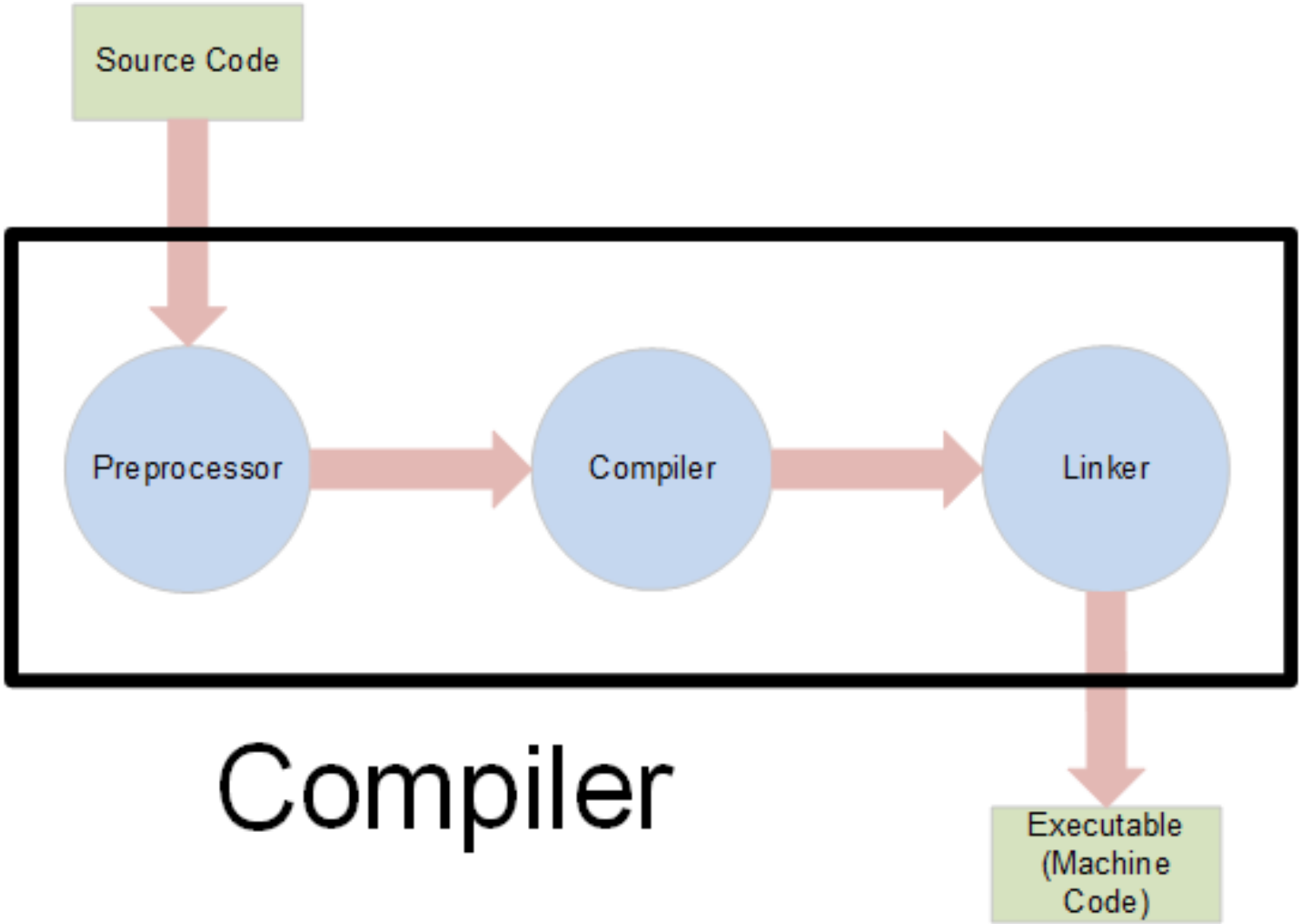
A Cross Maker

Benjamin Maier

January 21, 2020

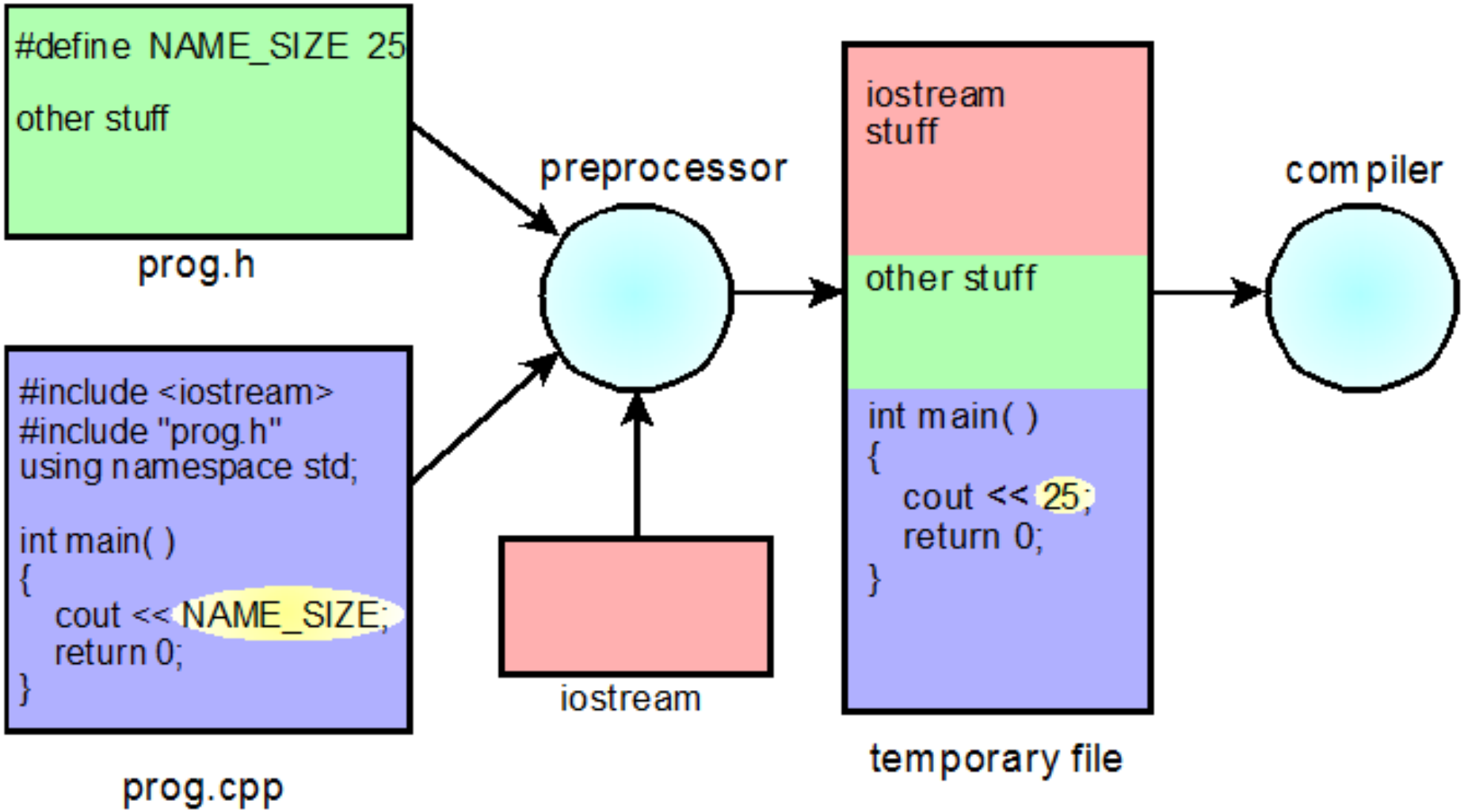
Outline

Vom Quellcode zur ausführbaren Datei



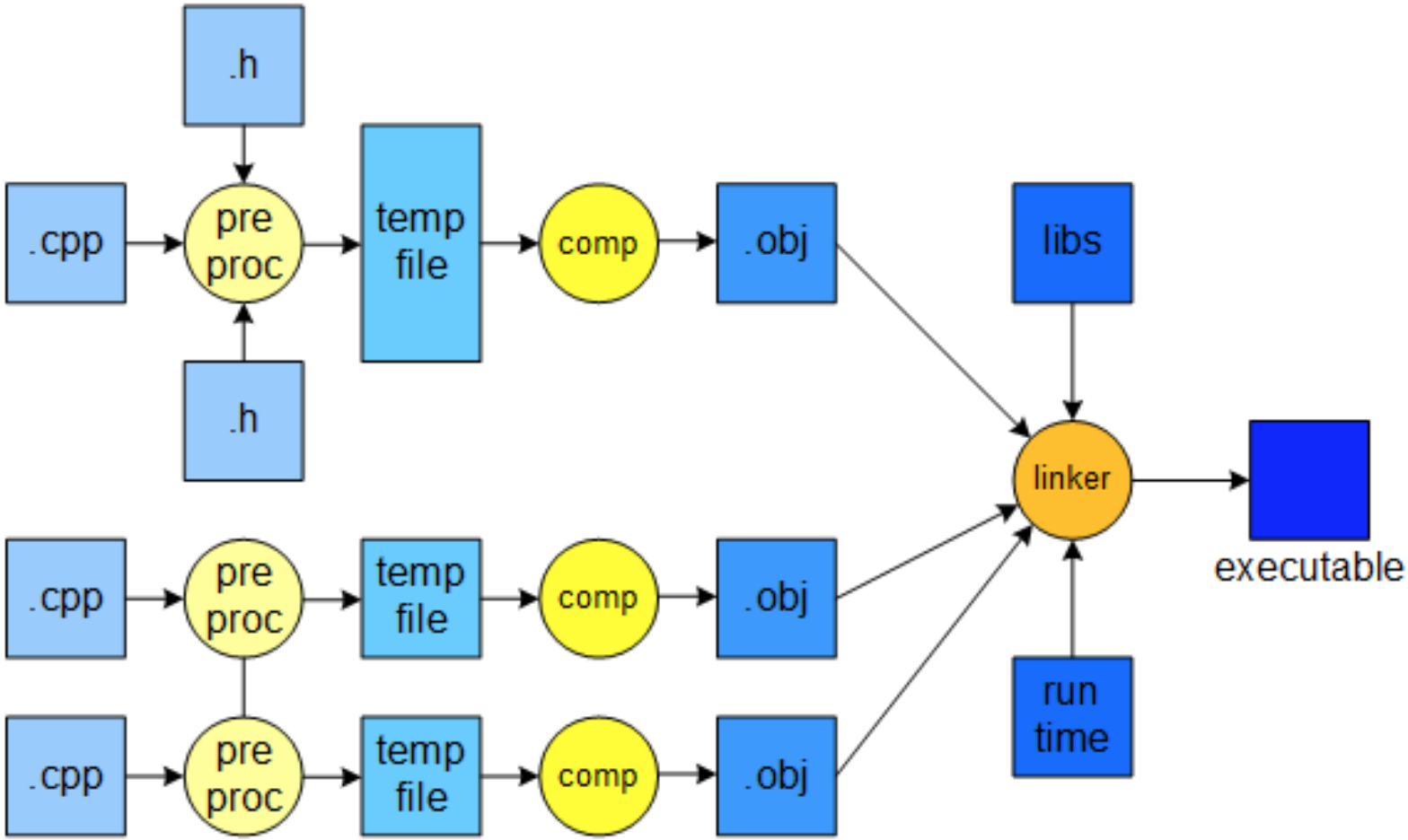
Quelle: `http://icarus.cs.weber.edu/~dab/cs1410/textbook/1.Basics/compiler_op.html`

Preprocessor



Quelle: `http://icarus.cs.weber.edu/~dab/cs1410/textbook/1.Basics/compiler_op.html`

Compiler und Linker

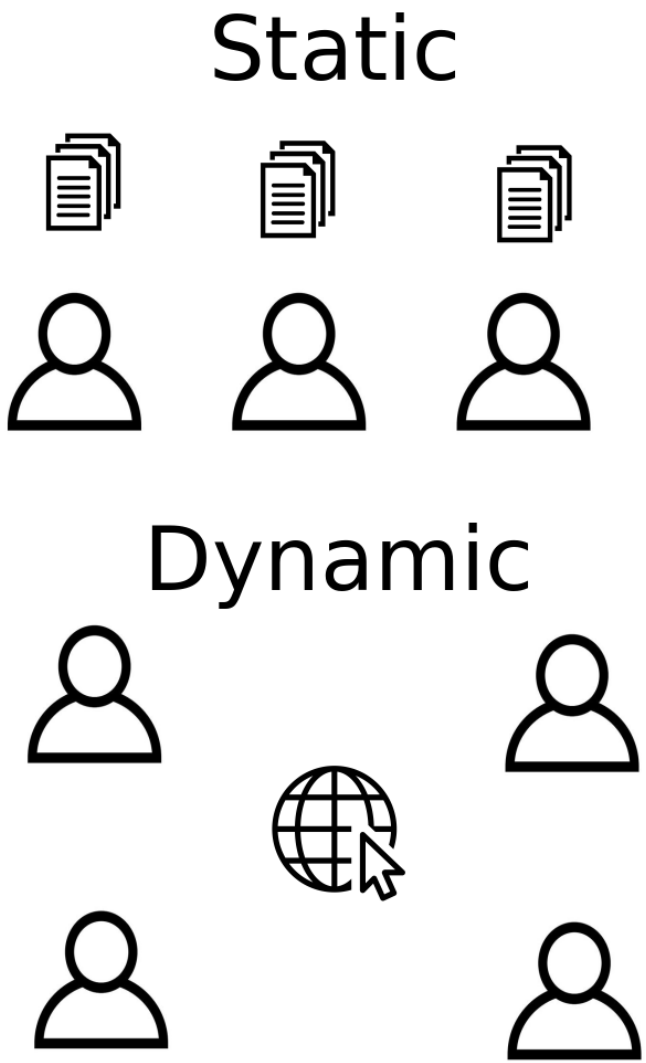


Quelle: `http://icarus.cs.weber.edu/~dab/cs1410/textbook/1.Basics/compiler_op.html`

Bibliotheken

- ermöglicht Projekte in Module einzuteilen
- Bibliotheken lassen sich leicht an andere Programmierer verteilen
- Code lässt sich leicht wiederverwenden
- Gewisse Problemstellungen wurden schon mal gelöst und müssen nicht neu erfunden werden
- Man unterscheidet zwischen statischen und dynamischen Bibliotheken

Statische versus dynamische Bibliotheken



Was kann CMake?

- Anstatt den Build eines Programmes in mehreren Schritten über die Konsole zu steueren (siehe Anhang), wurde Make eingeführt
- Build mittels formaler Beschreibung automatisch zu regeln und somit zuverlässiger und zeitsparender zu machen
- Am Ende ist nur noch ein Befehl nötig: make
- CMake (Cross-platform Make) ist ein plattformübergreifendes Tool, vorher war es nötig für jede Plattform, Compiler und Linker ein eigenes Makefile zu schreiben
- CMake erstellt das passende Makefile für die jeweilige verwendete Plattform
- Kann aber auch zusätzliche Projekte für verschiedene IDEs anlegen, beispielsweise eine Solution für Windows Visual Studios

Unsere erste CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)
project(minimal_example)

add_executable(hello_world main.cpp)
```

Bauen mit CMake

- Zunächst lässt man CMake die benötigten Makefiles erstellen
- $ cmake "path_to_the_CMakeFile"
- Anschließend lässt sich das Programm mit make bauen
- $ make

Machen wir es etwas komplizierter

- Projekte möchte man in Module unterteilen
- Man möchte mehrere Executables erstellen
- Man möchte Bibliotheken erstellen
- Man möchte andere Skripte ausführen
- Vieles, vieles mehr

Machen wir es etwas komplizierter

- Als kleines Beispiel habe ich eine Mathematikbibliothek geschrieben, welche ich mit CMake bauen möchte
- Code ist unter https://github.com/Xalanot/Tools_CMake

Top-level CMakeLists

- In der top-level CMakeLists wird einmal das Projekt angelegt, die Outputpfade gesetzt und die Unterordner eingebunden

```
cmake_minimum_required(VERSION 3.12)
project(simple_mathlib)

set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)

add_subdirectory(mathlib)

option(BUILD_TESTS "set this if you want to build the tests")

if(BUILD_TESTS)
        enable_testing()
        add_subdirectory(test)
endif()
```

Setzen der Outputpfade

```
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}
```

- Archive setzt den Outputpfad für die statischen Bibliotheken
- Library für dynamische
- Runtime für unsere executables

Hinzufügen eines Unterordners

```
add_subdirectory(mathlib)
```

- Der hinzugefügte Order muss selbst über eine CMakeLists.txt verfügen

Hinzufügen einer Flag

```
option(BUILD_TESTS
        "set this if you want to build the tests")
```

- CMake erlaubt es eigene Variablen zu verwenden, welche man beim Aufruf von cmake auf der Kommandozeile übergeben kann
- Diese lassen sich auch über CMake-GUI setzen
- In diesem Fall ist der default Wert False bzw. OFF, man kann aber auch einen anderen Wert wählen

CTest erlauben

```
enable_testing()
```

- CTest ist ein automatisches Testsystem von cmake, welches mit diesem Aufruf aktiviert wird
- Später zeige ich, wie man hierfür Tests hinzufügt

Hilfsfiles einbinden

```
include("cmake_files.cmake")
```

- Es kann manchmal hilfreich sein, Teile eines CMakeFiles auf mehrere Files aufzuteilen
- Wichtig ist die Fileendung ".cmake"

Bibliothek erstellen

```
add_library(mathlib_basic
        STATIC
        ${mathlib_basic_SOURCES}
        ${mathlib_basic_HEADERS}
)
```

- Wichtig ist die Angabe, ob man eine statische oder eine dynamische Bibliothek anlegen möchte
- Als nächstes muss man CMake sagen, welche Source Dateien zu der Bibliothek gehören
- Eigentlich reicht es nur die .cpp Dateien einzubinden, allerdings erscheinen die Header dann nicht in einer IDE

Includes für eine Bibliothek setzen

```
target_include_directories(mathlib_basic
        PUBLIC
        $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}
        /include>
)
```

- Build Interface ist dabei eine Generatorexpression, welche angibt, dass wir uns im Bildprozess befiden
- CMAKE_CURRENT_SOURCE_DIR beschreibt dabei das aktuelle Source Verzeichnis

Bibliotheken verlinken

```
target_link_libraries(mathlib_advanced
        PRIVATE
        mathlib_basic
)
```

- Private gibt dabei an, dass nur die Funktionen der mathlib advanced Funktionen von mathlib basic verwenden können, nicht aber zum Beispiel eine Bibliothek die wiederum mit mathlib advanced gelinkt wird

Fremdbibliotheken einbinden

```
find_package(Catch2 (REQUIRED))
```

- Der optionale Parameter REQUIRED gibt dabei an, ob ein Error geworfen werden soll, wenn das Paket nicht gefunden wird
- Damit ein Paket mittels find_package gefunden werden kann, muss ein CMake target von der Bibliothek erstellt werden
- Hat man die Bibliothek selbst nur gebaut und nicht installiert, so muss man CMake sagen, an welcher Stelle die find cmake Datei liegt

Tests hinzufügen

```
add_test(NAME catch2_test
         COMMAND mathlib_test_catch2)
```

- Wichtig ist dabei die Angabe von NAME und COMMAND