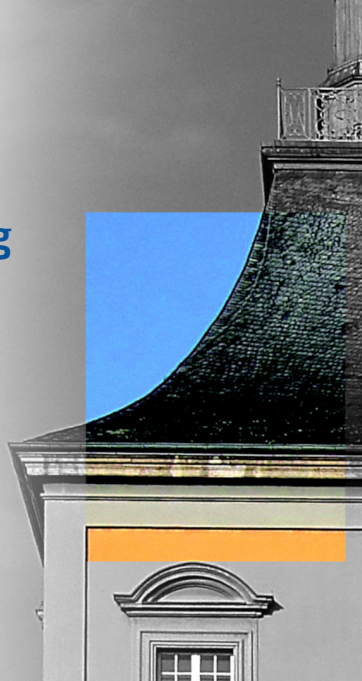
UNIVERSITÄT BONN

Algorithmen und Programmierung

Imperative Programmierung mit C++

Dr. Felix Jonathan Boes
boes@cs.uni-bonn.de
Institut für Informatik



KURZE WIEDERHOLUNG

Die Programmiersprache C++

Einschub: Text und Textausgabe

UNIVERSITÄT BONN

Strings

Einfache Texte werden als Instanzen der Klasse **std**::string repräsentiert. Strings können zu neuen Strings mithilfe von + verbunden werden. Mit der Funktion **std**::to_string(...) wird aus einem Objekt ein neuer String erzeugt.

```
#include <string> // Muss eingebunden werden um mit Strings zu arbeiten
std::string text = "Ich bin " + std::to_string(42) + " Jahre alt.";
```

Um Texte auf der Konsole auszugeben wird auf dem Objekt **std**::**cout** operiert. Dabei wird **std**::**cout** ein **std**::string mithilfe des Operators << übergeben.

```
std::cout << "Ich mag die Zahl " << std::to_string(42) << std::endl;
```

Um Texte von der Konsole einzulesen wird auf dem Objekt **std**::**cin** operiert. Dabei wird der von **std**::**cin** eingelesene Text an einen **std**::string mithilfe des Operators << übergeben.

```
std::string eingelesen;
std::cin >> eingelesen;
```

Die Programmiersprache C++

Arrays in C++

UNIVERSITÄT BONN

Arrays

Als **Array** bezeichnen wir ein zusammenhängendes Stück Speicher indem Werte desselben Typs abgelegt werden. Ein Array ist **statisch** wenn es eine feste, unveränderbare Größe besitzt und es ist **dynamisch** sonst. Dynamische C++-Arrays sind zu bevorzugen.

```cpp
#include <vector>
// int-Array mit drei Elementen
std::vector<int> x = {42, 9, 11};
// int-Array mit 77 Elementen
std::vector<int> y(77);
y[1] = 4; // Setzt das Element mit Index 1 auf 4.
y.size(); // Gibt 77 zurück.
```

UNIVERSITÄT BONN

Range-Based For-Loops

Um die Elemente eines Arrays (oder allgemein eines iterierbaren Containers) zu durchlaufen, verwenden erfahrenere Programmierer:innen Range-Based For-Loops.

Bei Range-Based For-Loops können die Elemente als Kopie; direkt mithilfe einer Referenz; oder direkt, aber nur lesbar mithilfe einer const-Referenz durchlaufen werden.

```
std::vector<int> zahlen = { /* ... */ };
for (int zahl : zahlen) {
  // zahl durchläuft alle Elemente von zahlen als Kopie
}

for (int& zahl : zahlen) {
  // zahl benennt sukzessive alle Elemente von zahlen und kann diese lesen und verändern
}

for (const int& zahl : zahlen) {
  // zahl benennt sukzessive alle Elemente von zahlen und kann diese nur lesen
}
```

Die Programmiersprache C++

Stack, Heap, Call by Value / Reference

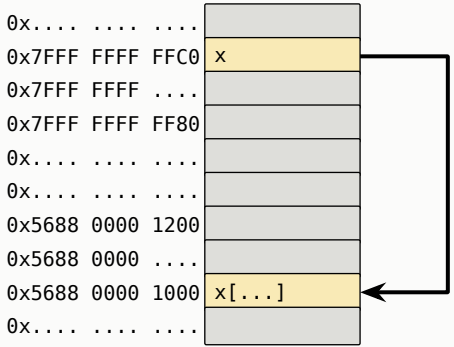
UNIVERSITÄT BONN

Call by Value im Speicher

Wir übergeben zunächst den Wert eines Array via Call by Value.

```
void ausgeben(const std::vector<int> z)
{
  ...
}

int main() {
  std::vector<int> x; ←
  ...
  ausgeben(x);
}
```



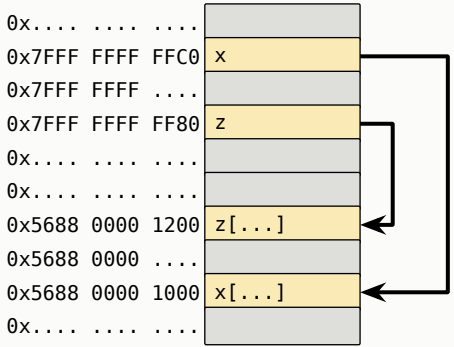
UNIVERSITÄT BONN

Call by Value im Speicher

Wir übergeben zunächst den Wert eines Array via Call by Value.

```
void ausgeben(const std::vector<int> z)
{
  ...          ←
}

int main() {
  std::vector<int> x;
  ...
  ausgeben(x); ←
}
```



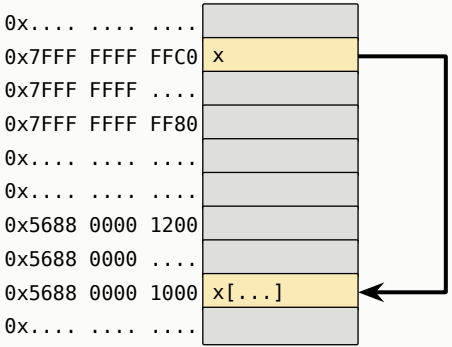
UNIVERSITÄT BONN

Call by Reference im Speicher

Wir übergeben nun eine Arrayvariable via Call by Reference.

```
void ausgeben(const std::vector<int>& z)
{
  ...
}

int main() {
  std::vector<int> x; ←
  ...
  ausgeben(x);
}
```



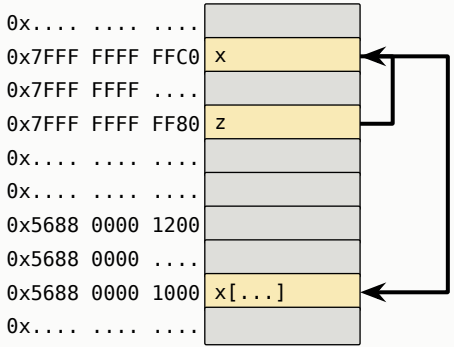
UNIVERSITÄT BONN

Call by Reference im Speicher

Wir übergeben nun eine Arrayvariable via Call by Reference.

```
void ausgeben(const std::vector<int>& z)
{
  ...          ←
}

int main() {
  std::vector<int> x;
  ...
  ausgeben(x); ←
}
```



Die Programmiersprache C++

Wiederkehrenden Programmcode zusammenfassen in Bibliotheken

Übergeordnetes Ziel

Sie lernen wie man wiederkehrende Codeabschnitte verständlich und wiederverwendbar im Bibliotheken zusammenfasst

Dazu verstehen wir wie Präprozessor, Linker und Compiler zusammenarbeiten

UNIVERSITÄT BONN

Motivation

Wir haben bereits gelernt, dass Codeabschnitte die **in Ihrem Programm** wiederkehrenden, zu Funktionen zusammengefasst werden sollen.

Sie lernen nun, dass Codeabschnitte die **in vielen Programmen** wiederkehren, zu **Bibliotheken / Modulen** zusammengefasst werden sollen. Eine Bibliothek stellt den Programmierer:innen Konstanten, Funktionen und Klassen zur (Wieder-)verwendung zur Verfügung.

UNIVERSITÄT BONN

Beispiele für Funktionalitäten die Bibliotheken zur Verfügung stellen

Es gibt sehr viele Bibliotheken für wiederkehrende Aufgabenkomplexe. Beispiele:

- Konstante, Funktionen und Klassen die der „Standard“ der Programmiersprache definiert
- Netzwerkkommunikation zwischen Programmen
- Effiziente Datenverarbeitung
- Darstellen und Arbeiten mit 3D Modellen
- Abspielen von Ton

UNIVERSITÄT BONN

Bibliotheken in C++

Für C++ Bibliotheken hat sich folgende Aufteilung bei gut lesbaren, wartbaren und verständlichen Bibliotheken etabliert.

- Eine aussgekräftige **Dokumentation** zur Installation und Handhabung der Bibliothek (als Manpage, Webseite, ...)
- Einer oder mehrer **C++-Header-Dateien**. Pro Header sind die verfügbaren Klassen (später mehr), Konstanten und Funktionen sowie deren Nutzweise (technisch sehr anspruchsvoll) skizziert.
- Menschenlesbare **C++-Quelldateien** oder die zugehörigen maschienenlerbaren **Bibliothekdateien**.
- Ein eindeutiger **Namespace** um alle in der Bibliothek definierten Klassen, Funktionen und Konstanten mithilfe des Namespace zu benennen (später mehr)

UNIVERSITÄT BONN

Codeaufteilung

Der Programmcode von Bibliotheken wird in **Header** und **Quelldateien** aufgeteilt. Die Headerdatei enthält alle **Klassendeklarationen** (später mehr), **Konstanten**, **Funktionsdeklarationen**, **Präprozessorkonstanten** und **Präprozessormakros** die den Nutzer:innen der Bibliothek direkt zur Verfügung stehen sollen. Die Headerdateien werden durch die Nutzer:innen beim Programmieren eingebunden.

Die Quelldateien enthalten alle **Klassendefinitionen** (später mehr), **Funktionsdefinitionen** und **Implementierungsdetails**, die den Nutzer:innen der Bibliothek nicht oder nur indirekt zur Verfügung stehen sollen. Die Quelldateien werden durch den Compiler zu einer (dynamischen) Bibliotheksdatei zusammengesetzt und stehen den Nutzer:innen beim Linken und Ausführen als Bibliotheksdatei zur Verfügung.

UNIVERSITÄT BONN

Beispiel

Wir schauen uns hier eine beispielhafte Implementierung der Systembibliothek `cmath` an. In der Headerdatei werden Konstanten definiert[1] und Funktionen deklariert. In der Quelldatei werden Funktionen definiert.

meine_cmath.hpp

```
const double mein_pi = 3.141592;

// Aussagekräftiger Kommentar
double mein_sin(double x);

// Aussagekräftiger Kommentar
double mein_cos(double x);

// Aussagekräftiger Kommentar
double mein_tan(double x);
```

meine_cmath.cpp

```
#include <meine_cmath.hpp>

// Aussagekräftiger Kommentar
double mein_sin(double x) {
  ...
}

...
```

[1] Noch besser ist es, die konstanten Variablen in der Headerdatei nur als „extern“ zu deklariert und sie in der zugehörigen Quelldatei zu definieren.

UNIVERSITÄT BONN

Arbeiten mit Bibliotheken
Guter Workflow

Um Bibliotheken zu nutzen, gehen Sie wie folgt vor.

- Dokumentation lesen.
- Prüfen ob die in der Dokumentation beschriebe Funktionalitäten ausreichen um das gewünschte Problem zu lösen.
- Bibliothek installieren (so wie in der Dokumentation beschrieben)
- Bibliotheksheader im eigenen C++-Quellcode einbinden (so wie in der Dokumentation beschrieben).
- Buildsystem einstellen / IDE einstellen / Compilerparameter anpassen (um die Bibliotheksdatei verwenden zu können).
- Programmieren, kompilieren, freuen.

UNIVERSITÄT BONN

Bibliotheksdateien

Beim Installieren von Bibliotheken werden so gut wie immer Header und Bibliotheksdateien auf dem System gespeichert. Der Bibliotheksquellcode ist beim Nutzen der Bibliothek nicht nötig.

Solche (dynamischen) Bibliotheksdateien haben folgende Vorteile:

- Wenn sich die Implementierungsdetails der Bibliothek ändern, zum Beispiel weil ein unbeabsichtigter Fehler behoben wurde, reicht es aus die Bibliotheksdatei zu aktuallisieren. Alle Programme die die Bibliothek verwenden, müssen nicht erneut kompiliert werden. Das spart Zeit.
- Eine geladene Bibliothek liegt nur einmal im Arbeitsspeicher und kann von allen Prozessen gleichzeitig verwendet werden. Das spart Ressourcen.

UNIVERSITÄT BONN

Dynamische Bibliothek bauen und einbinden

Es ist „eigentlich“ nicht kompliziert, dynamische Bibliotheken aus Quellcodedateien zu bauen und diese dann zu verwenden. Allerdings gibt es einige Details zu beachten, die vom Compiler und vom Betriebbsystem abhängig sind.

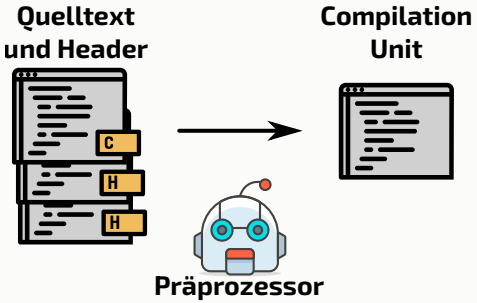
Zum Beispiel werden Bibliothek je nach Betriebbsystem unterschiedlich benannt. Desweiteren benötigt man ein wenig zusätzliches Wissen über C++-Spracherweiterungen und C-Spezifika um zu verstehen wie Bibliotheken eingebunden werden.

Wir verzichten hier auf ein explizites Beispiel.

UNIVERSITÄT BONN

Vom Quellcode zur Ausführbaren Datei

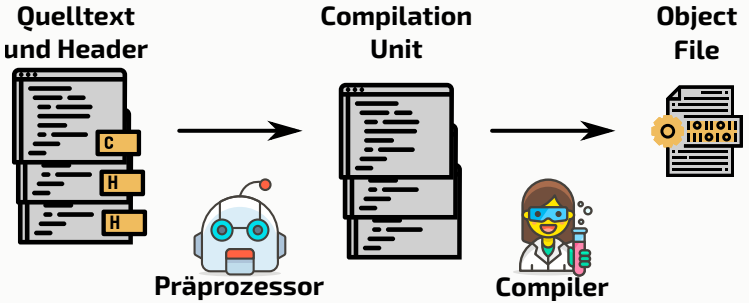
Bei der Überstzung von Quellcode werden die folgenden Schritte durchlaufen.



Der Präprozessor und der Compiler benötigen nur die Bibliotheksheader. Prozesse startet und wie die Bibliotheken zum Prozessstart eingebunden werden.

Vom Quellcode zur Ausführbaren Datei

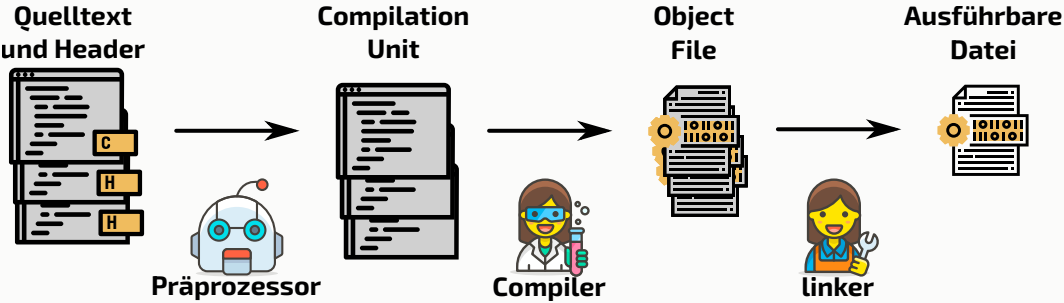
Bei der Überstzung von Quellcode werden die folgenden Schritte durchlaufen.



Der Präprozessor und der Compiler benötigen nur die Bibliotheksheader. Prozesse startet und wie die Bibliotheken zum Prozessstart eingebunden werden.

Vom Quellcode zur Ausführbaren Datei

Bei der Überstzung von Quellcode werden die folgenden Schritte durchlaufen.



Der Präprozessor und der Compiler benötigen nur die Bibliotheksheader. Prozesse startet und wie die Bibliotheken zum Prozessstart eingebunden werden.

UNIVERSITÄT BONN

Der Präprozessor

Der Präprozessor ist sehr fleißig. Die Aufgabe des Präprozessor ist es den C++-Quellcode für den Compiler vorzubereiten. Die typischen Aufgaben des Präprozessor sind Folgende.

- Kommentare, mehrfach Leerzeichen, Tabulatoren, Neuezeilen usw. entfernen.
- Headerdateien per Textersetzung einfügen.
- In Abhängigkeit von „Präprozessorbedingungen“ C++-Code generieren, hinzufügen oder entfernen.

UNIVERSITÄT BONN

Der Compiler

Der Compiler ist sehr klug. Die Aufgabe des Compilers ist es vorbereiteten C++-Code in möglichst effizienten Maschienencode umzusetzen.

Der dabei entstehende Maschinencode ist noch nicht ausführbar. Der Compiler weiß zu diesem Zeitpunkt zum Beispiel noch nicht mal ob der Maschinencode zu einer ausführbaren Datei oder zu einer Bibliothek zusammengesetzt werden soll.

Der Linker

Der Linker kennt sich sehr aus. Die Aufgabe des Linkers ist es Objektdateien zu einer ausführbaren Datei zusammenzusetzen. Außerdem hilft er dem Betriebssystem beim Starten von Programmen.

Dabei muss der Linker alle verwendeten Bibliotheken kennen und er muss ganz genau wissen wie das Betriebssystem Prozesse startet und wie die Bibliotheken zum Prozessstart eingebunden werden.

Präprozessor, Compiler und Linker

In dieser Vorlesung lernen wir den Präprozessor noch ein wenig besser kennen.

Dem Compiler würde man eine vollständige Vorlesungreihe widmen.

Den Linker treffen Sie in Modulen die betriebssystemnahe oder hardwarenahe Themen behandeln wieder.

Haben Sie Fragen?

Zusammenfassung

Sie haben gelernt wie man wiederkehrende Codeabschnitte verständlich und wiederverwendbar im Bibliotheken zusammenfasst

Dazu haben Sie verstanden wie Präprozessor, Linker und Compiler zusammenarbeiten

Die Programmiersprache C++

Präprozessordirektiven und Makros

Übergeordnetes Ziel

Sie lernen wesentliche Befehle kennen die der Präprozessor verarbeitet

UNIVERSITÄT BONN

Präprozessordirektiven I

Wir haben bereits gelernt, dass der Präprozessor den C++-Code für den Compiler vorbereitet. Mit **Präprozessordirektiven** stehen uns Befehle für den Präprozessor zur Verfügung. Wir haben unter anderem schon gelernt, wie man Bibliotheksheader einbindet.

UNIVERSITÄT BONN

Präprozessordirektiven II

Es gibt unter anderem folgende Präprozessordirektiven.

```
#include  // Fügt Headerdatei hinzu.
          // Mit <> werden Systembibliotheken eingebunden
          // Mit "" werden lokale Bibliotheken eingebunden

#define   // Definiert Präprozessorkonstanten und Makros
#undef    // Vergisst Präprozessorkonstanten und Markros

#ifdef    // Präprozessor Fallunterscheidungen:
#ifndef   // Code soll (nicht) eingebunden werden falls eine
          // Präprozessorkonstante definiert ist.

#if       // Präprozessor Fallunterscheidungen:
#elif     // Z.B. #if "CPU ist neu genug"
#else     //          Hier ist optimierter C++ der neusten CPU Features nutzt
#endif    //      # else
          //          Hier ist unoptimierter C++ Code

#pragma   // Steueranweisungen an den Compiler
```

UNIVERSITÄT BONN

Präprozessorkonstanten

Wir unterscheiden hier methodisch zwei Arten von Präprozessorkonstanten.

Zum einen gibt es Konstanten die das Vorhandensein einer Eigenschaft ausdrücken. Zum Beispiel welcher Compiler verwendet wird oder ob spezieller Programmcode zum Fehlerauffindung verwendet werden soll.

```
#define DRUCKE_STATUS_INFOS 1
...
#ifdef DRUCKE_STATUS_INFOS
  // Das vom Compiler zur Verfügung gestellte Makro __LINE__ druckt die aktuelle Zeile aus
  drucke("Wir befinden uns in Zeile: ", __LINE__);
#endif
```

Zum anderen gibt es Konstanten, die wir ausschließlich zum Zeitpunkt des Kompilierens kennen wollen. Das wird oft aus Optimierungsgründen verwendet.

UNIVERSITÄT BONN

Präprozessormakros

Einfach gesagt sind **Präprozessormakros** Textersetzungsregeln mit Parametern. Mit Makros kann man sehr komplexen C++-Programmcode generieren. Makros die keine Präprozessorkonstanten definieren werden eigentlich nur in fortgeschrittenen Programmen oder Bibliotheken verwendet.

Man findet eine gute Übersicht mit Beispielen hier:

`https://gcc.gnu.org/onlinedocs/cpp/Macros.html`

pragma und #ifndef-#define-#endif

Wenn Sie eine Headerdatei öffnen, so beginnt diese (neben Lizenztexten) oft mit dem Programmtext **#pragma** once oder sieht wie folgt aus.

mein_name.hpp

```
#ifndef __MEIN_NAME_HPP_
#define __MEIN_NAME_HPP_
...
#endif
```

Ein Grund ist Folgender. Bei größeren, etwas unübersichtlichen Projekten kann es passieren, dass Sie eigene Header schreiben, die sich zirkulär einbinden. Der Präprozessor würde dann sehr fleißig wiederholend dieselben Bibliotheksheader einbinden. Diese Endlosschleife des Einbindens wird mit **#pragma** once oder der obigen Konstruktion verhindert.

UNIVERSITÄT BONN

Beispiel: Die fmt-Library I

Mit der Standardausgabe `std::cout` lassen sich formatierte Text nicht sehr nachvollziehbar erzeugen. Um formatierte Texte[2] komfortable ausdrucken zu können, verwenden wir die Bibliothek *{fmt}*, auch *fmt-Library* genannt[3].

Die fmt-Library kann untypisch simpel in ein eigenes Projekt eingebunden werden. Dazu reicht es aus die Header-Dateien von *{fmt}* in das eigene Projekt zu legen. Genauer legt man das in *{fmt}* befindliche Verzeichnis `include/fmt` in das Verzeichnis `external` des eigenen Projekts. Anschließend kann *{fmt}* in unserem Projekt wie folgt verwendet werden.

```
// Compile with clang++ -I./include -I./external
#define FMT_HEADER_ONLY  // Dieses Makro ist nötig, um die Bibliothek simpel einzubinden
#include <fmt/core.h>

...
```

[2] In wenigen Monaten oder Jahren wird die in C++-20 eingeführte *Format Library* von allen modernen Betriebbsystemen und Compilern vollständig unterstützt.

[3] Siehe `https://fmt.dev/`

UNIVERSITÄT BONN

Beispiel: Die fmt-Library II

Die Bibliothek *{fmt}* stellt Funktionen bereit, um Strings zu erstellen oder zu drucken. Dem Compiler muss mitgeteilt werden, dass in `./include` Header liegen.

```
// Compile with clang++ -I./include -I./external ...
#define FMT_HEADER_ONLY  // Dieses Makro ist nötig, um die Bibliothek simpel einzubinden
#include <fmt/core.h>    // Muss eingebunden werden um {fmt} zu verwenden
#include <fmt/ranges.h>  // Muss zur Ausgabe von Arrays eingebunden werden
#include <vector>        // Stellt dynamische Arrays zur Verfügung

int main() {
  std::vector<int> v = {42, 5, 1};
  fmt::print("Das Array ist: {} und wir finden das gut.\n", v);
  // Druckt: 'Das Array ist: [42, 5, 1] und wir finden das gut'.
  fmt::print("{1} {0} {2}\n", fmt::join(v, " -> "), "Array: ", " Juhu!");
  // Druckt 'Array:  42 -> 5 -> 1  Juhu!'
}
```

Die Verwendung von *{fmt}* ist unter `https://fmt.dev/latest/syntax.html` kurz dokumentiert.

Haben Sie Fragen?

Ziel dieses Abschnitts

Sie haben hier die wichtigsten Details zu Präprozessordirektiven und Makros kennen gelernt

Die Bibliothek {fmt} wird zur einfachen Erzeugung von formatiertem Text verwendet und ist sehr simpel einzubinden

Sie lernen nun weitere, grundlegende Eigenschften von GIT kennen