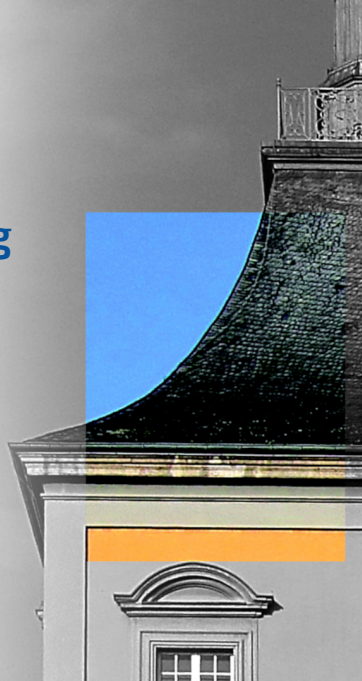
UNIVERSITÄT BONN

Algorithmen und Programmierung

Imperative Programmierung mit C++

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



KURZE WIEDERHOLUNG

Die Programmiersprache C++

Wiederkehrenden Programmcode zusammenfassen in Bibliotheken

UNIVERSITÄT BONN

Bibliotheken in C++

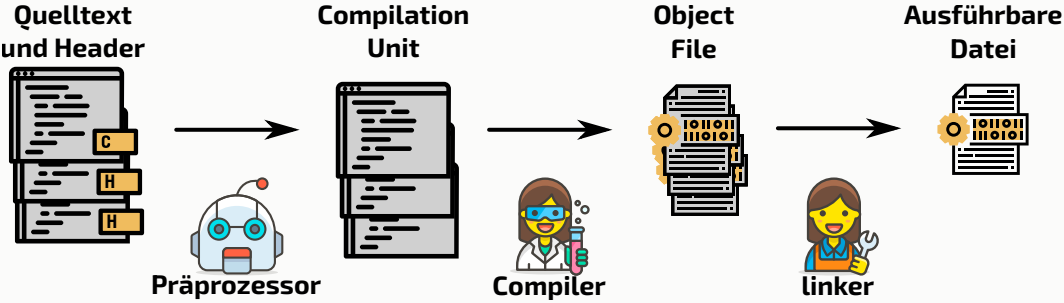
Für C++ Bibliotheken hat sich folgende Aufteilung bei gut lesbaren, wartbaren und verständlichen Bibliotheken etabliert.

- 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 und Funktionen mithilfe des Namespace zu benennen.

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.

Die Programmiersprache C++

Präprozessordirektiven und Makros

UNIVERSITÄT BONN

Präprozessordirektiven I

Mit **Präprozessordirektiven** stehen uns Befehle für den Präprozessor zur Verfügung. Dazu gehört das Einbinden von Bibliotheksheadern, das Definieren und Nutzen von Präprozessorkonstanten sowie das Definieren und Nutzen von Präprozessormakros.

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 und 1 (bzw. 0) 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
```

Ausblick: Anfänge der objektorientieren Programmierung

Ziel

Sie sollen einen ersten Einblick in die objektorientiere Programmierung gewinnen

Sie üben das hier Gelernte während den kommenden, theoriefokussierten Vorlesungen ein

Die wichtigsten Konzepte der objektorientieren Programmierung erlernen Sie im Anschluss

UNIVERSITÄT BONN

Erinnerung

Unser erstes, größeres Ziel ist es sowohl genügend Theorie als auch Praxis zu erlernen um Sortieralgorithmen zu studieren.

Genauer gesagt sollen Sie verstehen, wie man Objekte eines festen Typs sortieren. Zum Beispiel sollen Sie Telefonbucheinträge nach Vor- oder Nachnamen sortieren. Oder Sie sollen Ihre Memes absteigend nach der Anzahl ihrer Bewertung sortieren.

Um generelle Objekte zu sortieren verwendenden (rein) imperative Sprachen und objektorientiere Sprachen unterschiedliche Ansätze. Im Allgemeinen (also nicht nur für Sortieralgorithmen) führt der objektorientiere Ansatz zu nachvollziehbarerem, wartbarerem Programmcode.

UNIVERSITÄT BONN

Von imperativ zu objektorientiert

Computerprogramme verarbeiten Informationen um eine gegebene Aufgabe zu bearbeiten. Schlussendlich werden diese **Informationen** durch Bytefolgen im Speicher dargestellt. Diese **Daten** werden durch den **Programmcode** verarbeitet.

Bei (rein) **imperativen Programmiersprachen** liegen Daten und Programmcode im wesentlichen getrennt vor. Die vorliegenden Daten werden an Funktionen übergeben, die diese Daten verarbeiten.

Bei **objektorientieren Programmiersprachen** werden Daten und dazu gehöriger Programmcode zu **Objekten** zusammengefasst. Die Datenverarbeitung geschieht durch die gezielte Interaktion mit den vorliegenden Objekten.

UNIVERSITÄT BONN

Objekt

Wir fassen Daten und dazugehörigen Programmcode in einem **Objekt** zusammen. Dabei soll der lesende, schreibende und datenverarbeitende Zugriff ausschließlich durch definierte **Memberfunktionen** (manchmal auch **Nachrichten** genannt) geschehen.

Dadurch wird eine **Interaktionsschnittstelle** definiert welche von anderen Programierer:innen genutzt werden kann ohne die Implementierungsdetails kennen zu müssen. Das erhöht die Lesbarkeit und Wartbarkeit von Programmen enorm.

UNIVERSITÄT BONN

Objekttyp

In objektorientieren Programiersprachen wie C++, Python oder Java hat jedes Objekt einen definierten **Typ**.

Der Typ und die Funktionsweise eines jeden nutzerdefinieten Objekts wird durch eine zugehörige **Klasse** definiert.

Klasse

Durch die Definition einer **Klasse** wird ein zugehöriger Typ definiert. Desweiteren legt man durch die Klassendefinition fest aus welchen Daten der Typ zusammgesetzt ist, wie man Objekte aus dieser Klasse erzeugt und welche Memberfunktionen den erzeugten Objekten zur Verfügung stehen.

Intuitiv gesprochen dienen Klassen als Bauplan für nutzerdefiniete Objekte. Jedes Objekt wird zur Laufzeit erzeugt und wird als **Instanz** der zugehörigen Klasse bezeichnet[1].

[1] In C++ sind Objekte Instanzen von Klassen oder auch primitiven Datentypen. In Java sind Objekte Instanzen von Klassen (aber nicht von primitiven Datentypen).

Ausblick: Anfänge der objektorientieren Programmierung

Objekte in C++ verwenden

UNIVERSITÄT BONN

Definitionen am Programmbeispiel I

Wir verwenden die soeben erlernten Begriffe in einem trivialen Beispiel.

```
// Die Klasse std::vector<int> beschreibt wie Objekte von diesem Typ erzeugt werden und
// wie man mit Objekten dieses Typs interagieren kann.
// Die Definition befindet sich in der C++-Standardbibliothek.
#include <vector>

// x ist eine Instanz der Klasse std::vector<int>
// Insbespndere ist x ein Objekt vom Typ std::vector<int>
// Die Definiton der Klasse std::vector<int> legt fest wie Objekte dieses Typs erzeugt werden
std::vector<int> x(10);

// In der Definiton der Klasse std::vector<int> wird festgelegt welche weiteren Memberfunktion
// existieren und wie diese implementiert sind
x[3] = 42;
```

UNIVERSITÄT BONN

Definitionen am Programmbeispiel II

Die Memberfunktion dynamischer Arrays vom Typ `std::vector<Typ>` findet man beispielsweise in der (von uns empfohlenen) C++-Referenz unter

`https://en.cppreference.com/w/cpp/container/vector`

Hier ein Auszug

```
// Arrays erzeugen
std::vector<int> x;                 // Erzeugt ein Array der Länge 0.
std::vector<int> y{1, 5, 7, 9, 6};  // Erzeugt Array der Länge 5 mit den Elementen 1, 5, 7, 9, 6.
std::vector<int> z(100, 42);        // Erzeugt Array der Länge 100 mit den Elementen 42, 42, ...

// Rein lesender Zugriff auf Elemente
x.at(32);   // Gibt eine konstante Referenz auf das Element am Index 32 zurück

// Lesender oder Schreibender Zugriff auf Elemente
x[44];      // Gibt eine Referenz auf das Element am Index 44 zurück
x.front();  // Gibt eine Referenz auf das erste Element zurück
x.back();   // Gibt eine Referenz auf das letzte Element zurück
```

UNIVERSITÄT BONN

Beispiel: Objekte in C++

Die Memberfunktion dynamischer Arrays vom Typ `std::vector<Typ>` findet man beispielsweise in der (von uns empfohlenen) C++-Referenz unter

`https://en.cppreference.com/w/cpp/container/vector`

Hier ein Auszug

```
// Kapazitäten
x.size();   // Gibt die Anzahl der enthaltenen Elemente zurück
x.empty();  // Gibt (durch einen bool) an, ob das Array leer ist

// Array verändern
x.clear();        // Entfernt alle Elemente des Arrays
x.push_back(44);  // Fügt das Element 44 am Ende des Arrays ein
x.pop_back();     // Entfernt das letzte Element des Arrays (ohne Rückgabe)
```

Haben Sie Fragen?

Zusammenfassung

Objekte sind Instanzen von Klassen

Klassen definieren woraus ein Objekt besteht, wie es Erzeugt wird und wie mit ihm agiert werden kann

Ausblick: Anfänge der objektorientieren Programmierung

Eigene Klassen in C++ definieren

Offene Frage

Wie definiert man eigenen Klassen?

UNIVERSITÄT BONN

Eigenen Klassen entwerfen

Beim Entwurf einer Klasse wird definiert aus welchen Bestandteile die zugehörigen Objekte bestehen, wie es erzeugt wird und wie die Kommunikation mit diesen Objekten gestaltet ist.

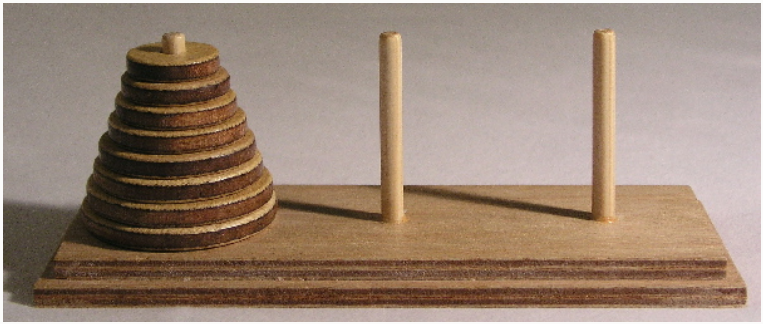
Der Entwurf von Klassen, die sowohl eine nachvollziehbare Kommunikationsschittstelle haben also auch über den Entwicklungszeitraum eines Projekts nicht angepasst werden müssen, erfordert viel Erfahrung und ist unter Umständen unmöglich.

In jedem Fall sollen Klassen mit ausreichend Zeit und Weitsicht entworfen werden. Dazu lernen Sie später auch UML-Diagramme kennen.

UNIVERSITÄT BONN

Beispiel: Türme von Hanoi

Wir betrachten beispielsweise das Spiel **Türmen von Hanoi**.



Ziel des Spiels ist es, den Turm bestehend aus n Scheiben von einem Pin auf einen anderen Pin in möglichst wenig Zügen zu bringen. Dabei darf immer nur eine Scheibe bewegt werden und kleinere Scheiben dürfen nicht unter größeren Scheiben gelegt werden.

UNIVERSITÄT BONN

Modellierung

Wir wollen die Türme von Hanoi modellieren. Ziel ist es eine Klasse zu entwickelen sodass sich die daraus instanzierten Objekte wie die Türme von Hanoi verhalten.

Mit diesem Ziel vor Augen überlegen wir zunächst was die Türme von Hanoi ausmacht wie sie erzeugt werden und wie man mit den Türmen von Hanoi interagiert.

Jede Konfiguration der Türme kann durch drei Stapel von natürlichen Zahlen modelliert werden. Dabei haben die Stapel eine Reihenfolge (links, mitte rechts) und keine Zahl kommt mehr als einmal vor.

Wir möchten die Türme von Hanoi in einer vorgegebenen Ausgangskonfiguration erstellen. Dabei reicht es zunächst die Anzahl der Scheiben vorzugeben. Außerdem wollen wir „versuchen“ eine oben liegende Scheibe auf einen anderen Stapel zu legen (falls das erlaubt ist). Zuletzt möchten wir die aktuelle Konfiguration ausdrucken.

UNIVERSITÄT BONN

Eigene Klassen im Projekt anlegen

Klassen sollen nach folgendem Schema in unseren Projekten angelegt werden.

```
+--docs
|
+--examples
|   |
|   +--main.cpp
|
+--include
|   |
|   +--mein_namespace
|       |
...     + meine_klasse.hpp
|
+--src
|   |
|   +--meine_klasse.cpp
...
|
CMakeLists.txt
```

UNIVERSITÄT BONN

Klassen deklarieren

In der zugehörigen Headerdatei wird die Klasse deklariert. Dabei sollen sich der Name der Klasse und der Name der Headerdatei entsprechen.

Die Klassendeklaration legt fest wie mit den zugehörigen Objekten interagiert werden kann. Dies geschieht durch die **Deklaration der Memberfunktionen**.

Außerdem werden die (meist von außen unzugängliche) Repräsentation der Objektdaten festgelegt. Dies geschieht durch die **Deklaration der Membervariablen**.

Bevor wir mehr Konzepte der objektorientieren Programmierung studieren sind alle Memberfunktion **`public`**, also für den „Zugriff von außen“ bestimmt.

Außerdem sind zunächst alle Membervariablen **`private`**, also nicht für den „Zugriff durch andere Typen von außen“ bestimmt.

```
#ifndef __NAME_DER_KLASSE_HPP__
#define __NAME_DER_KLASSE_HPP__

// Nötige Includes
...

namespace mein_namespace {
  class NameDerKlasse {
  public:
    // Konstruktor(en) deklarieren:
    NameDerKlasse();
    ...
    // Deklaration der Memberfunktionen
    RueckgabetypABC MemberfunktionABC( /* Parameter */ );
    RueckgabetypDEF MemberfunktionDEF( /* Parameter */ );
    ...
  private:
    // Deklaration der Membervariablen
    ...
  }; // <-- Nicht das Semikolon vergessen!
}

#endif
```

UNIVERSITÄT BONN

Türme von Hanoi - Headerdatei

```cpp
#ifndef __HANOI_HPP__
#define __HANOI_HPP__

#include <vector>
#include <cstddef>

namespace hanoi{
  class Hanoi{
  public:
    Hanoi(size_t anz);    // Erstellt eine Startkonfiguration mit anz vielen Scheiben
    bool bewege(size_t von, size_t nach);  // Verschiebt die oberste Scheibe (falls erlaubt)
      // und gibt true zurück genau dann wenn die Bewegung erlaubt war
    void drucke() const;  // Druckt die aktuelle Konfiguration auf der Konsole aus

  private:
    // Wir haben drei Stapel. Also bietet sich ein Array von Zahlenarrays an.
    std::vector<std::vector<size_t>> stapel;
  };
}
#endif
```

UNIVERSITÄT BONN

Klasse definieren

Nachdem die Klassendeklaration abgeschlossen wurde, werden die Memberfunktionen in einer separaten Quelldatei definiert. Diese Funktionsdefinitionen nennt man auch die **Implementierungsdetails** der Klasse.

```cpp
// Für die Implementierung der Klasse wird die Klassendeklaration benötigt
#include <mein_namespace/name_der_klasse.hpp>

// Weitere Includes
...

// Der Namespace des Projekts wird auch für die Definition der Memberfunktion verwendet
namespace mein_namespace {

  // Der volle Name der Memberfunktion ist bei der Definition anzugeben
  // Wir beginnen mit einem der Konstruktoren
  NameDerKlasse::NameDerKlasse() {
    ...
  }

  // Nun werden die restlichen Memberfunktionen implementiert.
  RueckgabetypABC NameDerKlasse::MemberfunktionABC( /* Parameter */ ) {
    ...
  }

  RueckgabetypDEF NameDerKlasse::MemberfunktionDEF( /* Parameter */ ) {
    ...
  }
}
```

UNIVERSITÄT BONN

Türme von Hanoi - Quelldatei

```
#include <hanoi/hanoi.hpp>
...

namespace hanoi{
  Hanoi::Hanoi(size_t anz) {
    // Die Variable stapel ist ein Array von drei Stapeln (welche wiederum Zahlenarrays sind)
    stapel = std::vector<std::vector<size_t>>(3);
    // Der erste Stapel soll den vollen Turm enthalten
    stapel[0] = std::vector<size_t>(anz);
    size_t i = anz;
    for(auto& x : stapel[0]) {
        x = i;
        i--;
    }
  }

  ...

}
```

UNIVERSITÄT BONN

Türme von Hanoi - Implementierungstest

```
#include <hanoi/hanoi.hpp>

int main() {
    hanoi::Hanoi h(5);
    h.drucke();

    h.bewege(0,1);
    h.bewege(0,1); // Illegale Bewegung
    h.bewege(0,2);
    h.drucke();

    h.bewege(1,2);
    h.drucke();

    h.bewege(0,1);
    h.bewege(2,0);
    h.bewege(2,1);
    h.bewege(0,1);
    h.drucke();
}
```

Das Buildsystem cmake

Bei Programmierprojekten mit einer wachsenden Anzahl von Klassen wird das Kompilieren mit einem Langen Befehl der Form

```
clang++ -std=c++17 -Wall -I./include src/meine_klasse.cpp src/meine_klasse2.cpp ...
```

schnell unübersichtlich.

Bei größeren Programmierprojekten verwendet man **Buildsysteme**. In dieser Vorlesung **nutzen** wir das häufig verwendete Buildsystem `cmake`. Im Modul *Praktikum Objektorientiere Softwareentwicklung* **lernen** wir genauer wie Buildsysteme funktionieren und wie sie konfiguriert werden.

Haben Sie Fragen?

Zwischenfazit

Klassen werden zuerst auf dem Papier entworfen

Die Klasse wird in der Headerdatei deklariert

Die Klasse wird in der Quelldatei definiert

Die Implementierung sollte stets getestet werden

Zum Bauen eines Projekts werden Buildsysteme verwendet

Ausblick: Anfänge der objektorientieren Programmierung

Weiterführendes

Ziel

In diesem Abschnitt besprechen wir erste, weiterführende (Stil)fragen

Das hier Besprochene wird nach den kommenden, theoriefokussierten Vorlesungen vertieft

UNIVERSITÄT BONN

Memberfunktion überladen

Wir wissen bereits, dass „normale“ Funktionen überladen werden können. Bei Memberfunktionen ist das Überladen ebenfalls möglich.

```
class K {
public:
...
  void drucke(); // Druckt das Objekt aus.
  void drucke(std::string preambel); // Druckt erst den Preambel und dann das Objekt aus.
...
};
```

UNIVERSITÄT BONN

Weitere Bemerkung zu Konstruktoren I

Auch Konstruktoren können überladen werden. Damit können Objekte auf verschiedene Weisen erzeugt werden. Beispiel `std::vector<int>`

```
std::vector<int> a;          // Erstellt leeres Array
// Ausgeführter Konstruktor:
// std::vector<int>()

std::vector<int> b(10);      // Erstellt Array mit 10 Elementen die den Wert 0 haben
// Ausgeführter Konstruktor:
// std::vector<int>(size_type)

std::vector<int> c(10,44);   // Erstellt Array mit 10 Elementen die den Wert 44 haben
// Ausgeführter Konstruktor:
// std::vector<int>(size_type, const T& value, const Allocator& alloc = Allocator())

std::vector<int> d(a);       // Erstellt eine Kopie des Arrays a
// Ausgeführter Konstruktor:
// std::vector<int>(const std::vector<int>& other);
```

UNIVERSITÄT BONN

Weitere Bemerkung zu Konstruktoren II

Beim Erzeugen eines Objekts wird stets ein ausgewählter Konstruktor verwendet. Bevor der Konstruktor ausgeführt wird, werden alle Membervariablen des Objekts initialisiert. Dazu werden (wenn nichts weiteres angegeben ist) die jeweiligen Standardkonstruktoren dieser Membervariablen ausgeführt.

Man kann die Initialisierung der Membervariablen auch explizit vorbestimmen. Dazu gibt man die Initialisierung der Membervariablen zwischen dem Konstruktorfunktionsnamen und dem Konstruktorfunktionskörper an.

```
class K{
public:
  K() : x(100), y(20,10) {
    std::cout << "x:" << x << " y[0]:" << y[0] << std::endl; // Druckt 'x:100 y[0]:10'
  }
private:
  int x;
  std::vector<int> y;
};
```

UNIVERSITÄT BONN

Besser Programmieren mit Const Correctness

Das Schlüsselword `const` kann auch verwendet werden um Memberfunktionen zu modifizieren. Ein `const` hinter dem Memberfunktionsnamen sorgt dafür dass der Funktionskörper die Membervariablen des Objekts nicht verändern darf.

```
class K{
public:
  void ich_tue_nichts() const {
    x = 42; // Das verbietet der Compiler!
  }
private:
  int x;
};
```

Die Verwendung von `const` in diesem Kontext sorgt wie auch sonst dazu die Nachvollziehbarkeit des Programms zu erhöhen und Programmierfehler zu vermeiden.

UNIVERSITÄT BONN

Sichtbarkeitsbereiche I

Variablen und Typen haben einen Namen. Um Variablen und Typen zur verwenden müssen sie deklariert werden. Der Sichtbarkeitsbereich eines Namens wird als **Scope** bezeichnet.

Variablen die innerhalb eines Compoundstatements definiert werden, sind in diesem Compoundstatement sichtbar. Sie sind nicht außerhalb des jeweiligen Compoundstatements sichtbar.

Variablen die als Funktionsparameter eingeführt werden, sind innerhalb der Funktion sichtbar. Sie sind nicht außerhalb der jeweiligen Funktion sichtbar.

Private Membervariablen eines Objekts vom Typ T,sind nur innerhalb der Memberfunktionen von Objekten des Typs T sichtbar.

UNIVERSITÄT BONN

Sichtbarkeitsbereiche II

Mit **Namespaces** wird die Sichtbarkeit von Typnamen eingeschränkt. Typen die innerhalb desselben Namespace deklariert werden sehen sich gegenseitig. Typen die nicht innerhalb desselben Namespace deklariert werden müssen den Typnamen um den jeweiligen Namespace ergänzen.

```
// Voller Name des Typ MeineKlasse im Namespace mein_namespace
mein_namespace::MeineKlasse
```

Es ist als guter Stil angesehen aussagekräftige Namespaces zu verwenden um mehrfach vorkommende Typnamen zu erschweren.

Mit ausreichend Programmiererfahrung (und nur dann) kann man (unter Umständen) Namespaces mithilfe von **`using namespace`** `mein_namespace` einbinden.

UNIVERSITÄT BONN

Einige Faustregeln zum Erreichen von gutem Stil

Wir geben hier eine Auswahl von Faustregeln um gutem Stil näher zu kommen.

- Die Verwendung von Projektnamespaces und die Trennung von Klassendeklaration und Klassendefinition erleichtert die Interoperabilität zwischen Entwicklerteams.
- Wir überlegen uns vorrangig, welche Interaktionsmöglichkeiten mit dem / Operationen auf dem Objekt gewünscht sind. Diese werden dann vollständig umgesetzt. Diese gute Sichtweise führt auch bei sehr großen Projekten zu leserlichem, wartbaren, fehlerfreiem Code.
- Memberfunktionen sind **`public`** und Membervariablen **`private`**. Falls man auf die Membervariablen direkt auslesen / schreiben möchte, verwendet man dafür Memberfunktionen. Diese Faustregel wird oft genutzt um den Zustand des Objekts sicherzustellen.
- Memberfunktionen die den Zustand des Objekts nicht verändern sollen werden mit dem Schlüsselwort `const` versehen.

Haben Sie Fragen?

Zwischenfazit

Der Entwurf eigener Klassen geschieht am Papier und muss eingeübt werden

Eigene Klassen nachvollziehbar und möglichst fehlerfrei zu implementieren muss eingeübt werden

Die genannten Faustregeln helfen beim Ausprägen von gutem Programmierstil

UNIVERSITÄT BONN

Kurzer Ausblick

Nach den kommenden, theoriefokussierten Vorlesungen erwarten Sie die folgenden Konzepte des objektorientieren Programmierens

- Vererbung und Typbeziehungen
- Sichtbarkeit durch Zugriffsmodifikatioren
- Klassenmember
- Interfaces und abstrakte Klassen
- Repräsentierung von Objekten und Vererbung im Speicher
- Generisches Programmieren