

Programmieren (T3INF1004)

DHBW Stuttgart

Christian Bader

christian.Bader@lehre.dhbw-stuttgart.de

Christian Alexander Holz

christian.holz@lehre.dhbw-stuttgart.de



Recap / Vorstellungen

Live

Kapitelübersicht

- 1 Einführung
- 2 Objektorientierung
- 3 Vertiefung C++**
- 4 Die Standard Template Library (STL)
- 5 Clean Code
- 6 Test-driven Development

Kapitel 3: Vertiefung C++



31. Weitere C++ Keywords

32. Namespaces

33. Objektorientiertes Design

34. Übungsprojekt

Initialisierung

- Die Cpp spezifische Initialisierung durch { } nennt man *List initialization*
- Bietet folgende Vorteile:
 - Verhindert unbeabsichtigten type cast (z. B. von double nach int)
 - Sicher vor „narrowing“: Es wird kein Wertverlust erlaubt
 - Compiler kann viele Fehler frühzeitig erkennen
 - Klare und einheitliche Syntax

```
1 void initialization(double val, int val2) {  
2  
3     int x2 = val;    // if val == 7.9, x2 becomes 7 (bad)  
4  
5     char c2 = val2;  // if val2 == 1025, c2 becomes 1 (bad) char hat 8 Bit  
6  
7     int x3{val};     // error: possible truncation (good)  
8  
9     char c3{val2};   // error: possible narrowing (good)  
10  
11    char c4{24};      // OK: 24 can be represented exactly as a char (good)  
12  
13    char c5{264};     // error (assuming 8-bit chars): 264 cannot be  
14    | | | |           // represented as a char (good)  
15  
16    int x4{2.0};       // error: no double to int value conversion (good)  
17 }
```

List Initialisierung { } sollte (fast) immer bevorzugt werden.

Initialisierung

Ausnahme:

- Sie wollen (sollten) deutlich machen, dass eine Funktion / Methode aufgerufen wird.

```
02_lecture > 2_Objektorientierung > 21_Grundlagen > 212_Konstrukturen > Rectangle > Rectangle.cpp > main()
1  #include <iostream>
2
3  > class Rectangle...
44
45  int main()
46  {
47      Rectangle myRectangleClearInit(2.0, 3.0); // clear: we call a methode
48      Rectangle myRectangleStrangeInit{2.0, 3.0}; // works also but looks strange
49
50      std::cout << myRectangleClearInit.calcArea() << std::endl;
51
52      return 0;
53  }
54
```

List Initialisierung { } sollte (fast) immer bevorzugt werden.

const

- Zur Definition von Variablen die nicht mehr verändert werden sollen (z.B. Konstanten)
- Keyword gibt an, dass die jeweilige Variable zur Laufzeit nicht mehr geändert werden kann

const

- OOP: Methoden können `const` sein → Sie dürfen keine Member ändern
- OOP: Membervariablen (Argumente) können `const` sein → Diese dürfen von der Funktion/Methode nicht geändert werden

const

- `const`-Membervariablen müssen im Header initialisiert werden (wir können ihnen ja nichts später zuweisen)
- nicht `const`-Membervariablen sollten in der `.cpp` gesetzt werden.

Wann sollten Sie `const` / `constexpr` verwenden?

Wann immer möglich!

static

- Alle bisher genutzten Methoden/Variablen existierten auf Objektebene
- Zum Aufruf der Methoden muss vorher ein Objekt erzeugt sein
- Die Werte der Variablen können in jedem Objekt dieser Klasse unterschiedlich sein
- Mithilfe von `static` lassen sich **Variablen/Methoden definieren, die objektlos aufgerufen werden können**
- Dieser Wert ist dann in allen Objekten dieser Klasse gleich
- Die Methode hat nie Abhängigkeiten zu den Daten eines spezifischen Objekts

static

- Alle bisher genutzten Methoden/Variablen existierten auf Objektebene
- Zum Aufruf der Methoden muss vorher ein Objekt erzeugt sein
- Die Werte der Variablen können in jedem Objekt dieser Klasse unterschiedlich sein
- Mithilfe von `static` lassen sich **Variablen/Methoden definieren, die objektlos aufgerufen werden können**
- Dieser Wert ist dann in allen Objekten dieser Klasse gleich
- Die Methode hat nie Abhängigkeiten zu den Daten eines spezifischen Objekts

auto

- Kann anstelle des konkreten Typs bei einer Zuweisung angegeben werden, wenn direkt ein Wert zugewiesen wird
- Compiler ermittelt selbst den Typ
- Typsicherheit bleibt erhalten, d.h. Compiler warnt weiter wenn Typen nicht zusammen passen
- C++ Standard: AAA: Almost Always Auto
- Aber: Gefahr Lesbarkeit zu reduzieren!
- Besser: Auto nur wenn Typ auf rechten Seite klar definiert
- Auto nur wenn Lesbarkeit erhöht wird, z.B. bei zu langen Klassennamen

Welche Verwendung von auto ist sinnvoll?

```
auto numberOfUsers{5};  
int measValue{5.5};  
...  
auto humCalculator{HumidityCalculator(20, 50)};  
auto humCalculator = HumidityCalculator(20, 50);  
...  
auto measValue02{measValue}
```

Friend class

- Variablen, die in der Regel private sein sollen, werden **von einer einzelnen spezifischen Klassen** benötigt
- hierzu wäre es nicht sinnvoll diese komplett als `public` zu definieren
- Die andere Klasse kann als `friend` deklariert werden
- `friend` Beziehungen gelten nur in eine Richtung
- Die Klasse `FriendClass` bekommt Zugriff auf die private Variablen der Klasse `Class`
- **Z.B. in Tests sinnvoll**

```
class Class
{
public:
    Class()
    : m_privateMember(5)
    {};

    ~Class()
    {};

    friend class FriendClass; //define FriendClass to be friend

private:
    int m_privateMember; //private member variable
};

class FriendClass
{
public:
    void testAccess(const Class& testObj)
    {
        std::cout << testObj.m_privateMember << std::endl; //has access to private member of Class
    }
};
```

Aufgaben I: C++ keywords

- 1) Nehmen Sie das die Aufgabe constEdit (Git Repo) und setzen Sie alles `const` was `const` gesetzt werden kann (und auch sollte).
Versuchen Sie nicht zu raten sondern zu wissen.
- 2) Erstellen Sie eine Klasse User.
 - a) Dieser User hat einen Namen und eine einzigartige ID.
 - b) Der Name wird im Konstruktor übergeben, die ID im Konstruktor erstellt.
 - c) Die Namensklasse erstellt die ID selbst.
 - d) Der Nutzer hat eine Print Methode, die den Namen und die ID ausgibt. Die ID soll für die Nutzer hochzählend sein, beginnend bei 0.
 - e) Legen Sie mehrere Nutzer an und geben Sie Namen und ID aus.

Aufgaben II: C++ keywords

- 1) Erstellen Sie eine Verwaltung von Bankkonten
 - a) Ein Konto definiert sich über eine 6-stellige Kontonummer (zufällig generiert von der Klasse selbst), einen Kontotyp (auswählbar aus Girokonto, Tagesgeldkonto und Bausparvertrag), eine BLZ (unveränderlich), ein aktuelles Guthaben und einen Zinssatz. Zur Generierung der Zufallszahl kann `std::mt19337` genutzt werden (aus `<random>`)

```
std::random_device device;  
std::mt19937 generator(device());  
std::uniform_int_distribution<int> distribution(100000, 999999);  
m_accountNumber = distribution(generator);
```

- b) Jedes Konto hat eine Methode, die alles nennenswerte ausgibt und eine Methode, mit der man Geld einzahlen (positiv) und auszahlen (negativ) kann.
 - c) Es soll gezählt werden wie viele Giro/Tagesgeld und Bausparverträge insgesamt erstellt wurden (also drei Counter).

Aufgaben III: C++ keywords

- Das Girokonto:
 - Hat einen Dispo. Es kann auch ins Negative gehen.
- Ein Tagesgeldkonto:
 - Hat eine Mindestlaufzeit (unveränderbar)
 - Dieses Konto kann nichts ins Negative gehen.
- Der Bausparvertrag:
 - Hat eine Bausparsumme (unveränderbar)
 - Geld ist einzahlbar, solange die Bausparsumme nicht erreicht ist.
- Testen Sie die Konten, indem Sie mehrere jeder Art anlegt und mit verschiedenen Werten befüllt und verschiedene Operationen aufruft. Die Werte der Objekte dürfen nur über Methoden geändert werden. Code-Duplikation soll vermieden werden.

Kapitel 3: Vertiefung C++

31. Weitere C++ Keywords



32. Namespaces

33. Objektorientiertes Design

34. Übungsprojekt

Namespaces - Motivation

- Annahme:
Wir implementieren ein Algebra Programm und wollen darin eine Klasse `Vector` implementieren
- Problem: `Vector` gibt es schon! → Compiler Fehler
- Lösung: Namespaces! `Vector` der C++ Standard Template Library (STL) liegt im Namespace `std` und wird aufgerufen mit `std::vector` → Nur `std::vector` ist belegt, wir können unsere Klasse `Vector` nennen.
- Für große Projekte und oft wieder verwendeten Code unverzichtbar

Vorteile von Namespaces:
Verhindern von Namenskollisionen
Verbesserte Lesbarkeit (z.B. `algebra::vector`)

Namespaces - Beispiel

- Bei großen Projekten: ALLES in Namespaces mit Namenskonventionen für die Namespaces, z.B. jedes Projekt einen eigenen Namespace
- Meine Empfehlung: In Namespaces nicht einrücken, da sonst oft ganze Files einfach ein oder zweimal eingerückt werden
→ weniger Platz zum Vergleichen
- Aber Ende des Namespaces kommentieren, wofür die geschlossene Klammer steht

```
1  #include <vector>
2
3
4  namespace Algebra
5  {
6
7      class Vector
8      {
9
10     };
11
12 } // end namespace algebra
13
14
15 int main()
16 {
17     // create STL vector
18     std::vector<int> vec;
19     // create algebra vector
20     Algebra::Vector vec;
21
22     return 0;
23 }
```

Erster Buchstabe groß. Nach includes, vor class

Im Namespace nicht eingerückt

Ende kommentiert

Namespaces – hpp and cpp files

```
3_Vertiefung_Cpp > namespaces > C++ main.cpp > main()
1  #include <vector>
2
3  #include "classInNamespace.hpp"
4
5
6  int main()
7  {
8
9      MyNamespace::ClassInNamespace myObject;
10
11      using MyNamespace::ClassInNamespace;
12
13      ClassInNamespace myObject;
14
15      return 0;
16  }
17
18

3_Vertiefung_Cpp > namespaces > h++ classInNamespace.hpp > ...
1
2  namespace MyNamespace
3  {
4
5      class ClassInNamespace
6      {
7
8      };
9
10 } // end namespace MyNamespace

3_Vertiefung_Cpp > namespaces > C++ classInNamespace.cpp > {} MyNamespace
1  #include "classInNamespace.hpp"
2
3  namespace MyNamespace
4  {
5
6      ClassInNamespace::ClassInNamespace()
7      {
8          // Implementation of Default Constructor
9      }
10
11 } // end namespace Mynamespace
```

- Namespaces können sich über mehrere Dateien erstrecken
- Wenn zusätzliche Klassen und Funktionen in einen Namespace eingefügt werden sollen oder die Implementierungen in der .cpp Datei, dann geht dies einfach über das erneute definieren des Namespaces.

using namespace

- Ganze Namespaces können über `using` in den aktuellen Namespace eingebunden werden (z.B. `using namespace std;`)

using namespace: Nur mit viel Vorsicht!
Und niemals in Headern!

- Auch Vorsicht vor `using namespace std;`
 - STL wird häufig genutzt: erscheint sinnvoll
 - Aber STL ist im stetigen Wandel
 - Namenskollisionen können entstehen obwohl eigener Code nicht geändert wurde
- Alternative: einzelne oft genutzte Ausdrücke aus `namespace` befreien durch z.B. `using std::cout;` (geht auch mit eigenen langen Namespaces)