Kapitel 12 - Einführung in C++

Peter Ulbrich

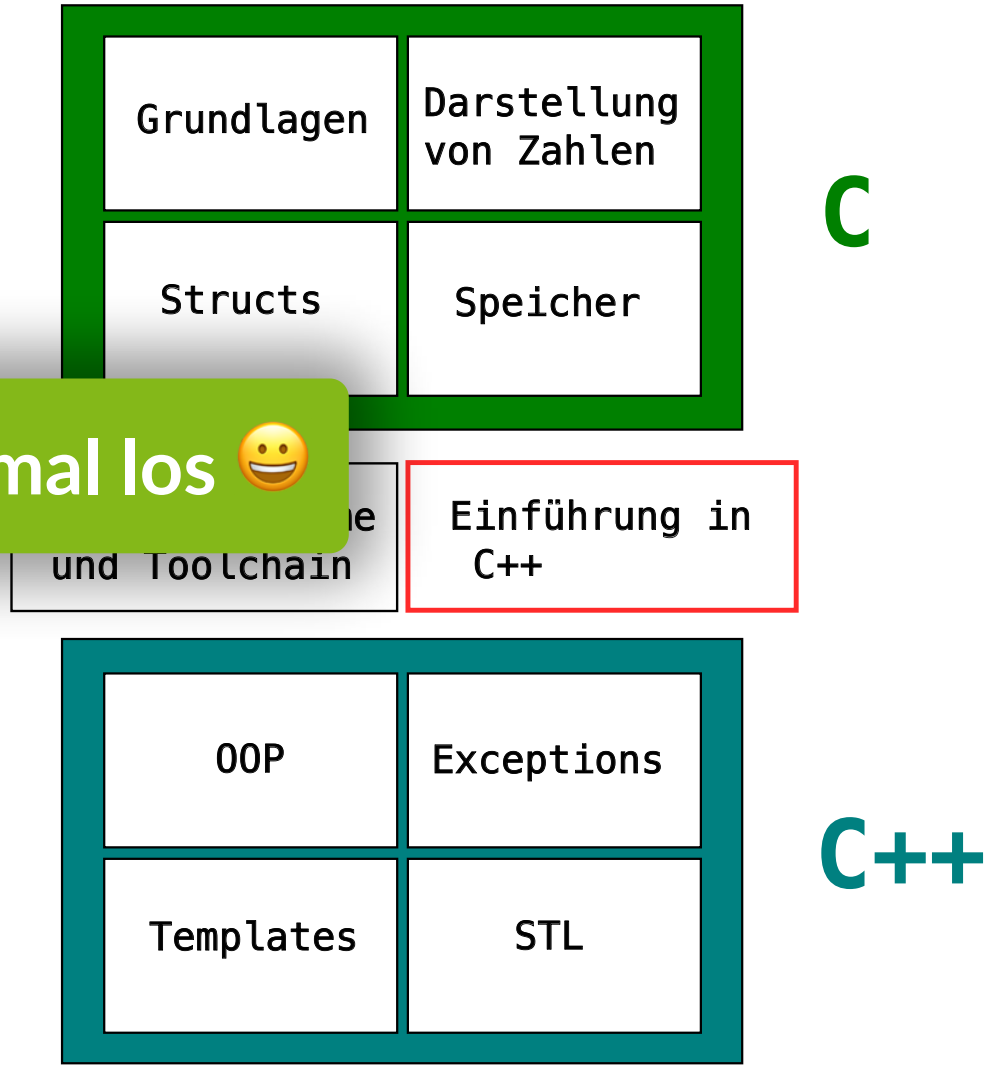
AG Systemsoftware

Veranstaltungswebseite

Rückblick – Aktueller Stand

- Weiterer Ablauf der Vorlesung:
  1. Einführung in C++ (nächstes Kapitel)
     - Wichtigste Unterschiede und Stolpersteine
  2. Objektorientierte Programmierung (OOP)
     - Bündelung von Funktion
       konsistenter Initialisierur
  3. Ausnahmenbehandlung (*Exceptions*)
  4. Generische Programmierung (*Templates*)
  5. Kurzer Ausflug in die *Standard Template Library* (STL)

Dann legen wir mal los 😀



Unterschiede – C und C++

- Zur Erinnerung: C++ ist ein *Superset* von C
- Der Funktionsumfang von C ist (größtenteils) in C++ abgebildet

  → *Kein Zwang*, andere Funktionen/Konzepte als bisher zu verwenden
- **Wichtig:** Der C++-
  - ISO-Spezifikation v
  - Viele Funktionalität
  - Alte Komponenten wurden wegen Abwärtskompatibilität behalten
  - Unmöglich, alles davon zu verwenden bzw. in einer Vorlesung zu behandeln

Wir schauen uns jetzt einmal das Notwendigste an

- Pragmatische Lösung: Verwendung der benötigten/bevorzugten Komponenten

  → Rest ignorieren 😀

Back to the Roots – „Hello World“ in C++

cpp Run ▶

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
}
```

- Unmittelbar auffällig: Anderer Header + Ausgabefunktion sieht anders aus
- Header in C++ verzichten auf das Suffix `.h`
- Anstatt `printf()` jetzt `std::cout` (über *Outputstreams*)
- Im Folgenden:
  - Wofür steht das `std::` ?
  - Was zeichnet Outputstreams aus?

Einführung von Namensräumen

- Worin besteht hier das Problem in C?

```
1  void print(double d);
2  void print(int i);
```

- Zur Erinnerung: In C **alle** Funktionen und Variablen auf globaler Ebene angesiedelt
  - Zum einen: Sehr unübersichtlich
  - Zum anderen: Namensvergebung ist schwierig, Doppelungen leicht möglich
- Die Lösung: Namensräume
- Namensräume (*Namespaces*) kapseln Funktionalität

```
1  // Besser: Namespaces in C++
2  namespace Double {
3      void print(double d) {
4          std::cout << d << std::endl;
5      }
6  }
7
8  namespace Integer {
9      int i = 1;
10     void print(int i) {
11         std::cout << i << std::endl;
12     }
13 }
```

Namensräume

- Zugriff auf Komponenten des Namespaces mit dem *Scope Resolution Operator* → `::`
- Daher auch Präfix `std` → Zugriff auf Namensraum `std`

  `std` entspricht C++-Standardbibliothek
- Anmerkung: Es gibt auch den globalen Namespace
  - Leerer Name, Zugriff mit `::` als Präfix (→ `::`x)
  - Verdeutlicht Verwendung globaler Variable → besseres Verständnis
  - Umgehung von lokalem Shadowing einer Variablen

Beispiel - Namensräume

cpp Run ▶

```cpp
#include <iostream>

int k = 4711;

namespace A {
    double d = 3.5;
    void print(double d) {
        std::cout << d << std::endl;
    }
}
namespace B {
    int i = 1;
    void print(int i) {
        std::cout << i << std::endl;
    }
}

int main() {
```

Verschachtelung von Namensräumen

- Namespaces können beliebig geschachtelt werden:

cpp Run ▶

```cpp
#include <iostream>

namespace A {
    int i = 0;
    namespace B {
        int add(int num1, int num2) {
            return num1 + num2;
        }
    }
    namespace C {
        int j = 2;
    }
}

int main () {
```

(Namenräume können über mehrere Dateien verteilt sein → siehe `std`)

Verwendung von Namensräumen

- Problem: Viel Tipparbeit bei Verwendung von Namensräumen
- Lösung: Das Schlüsselwort `using`!
- Erlaubt selektives Einbinden von benötigten Funktionen/Variablen
- Einbinden von ganzen Namespaces ist ebenfalls möglich:

→ Beispiel: `using namespace std`

```cpp
#include <iostream>
using std::cout, std::endl;

int main() {
    cout << "Hello World!" << endl;

}
```

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Hello World" << endl;
}
```

Anmerkungen zum Einsatz von `using namespace std`

- Einerseits: Weniger Tipparbeit als vorher
- Aber: Globaler Namesraum wieder genauso gefüllt wie in C
- Vorteil der Schach
- Außerdem: Häufig
  Implementierunge
  - Bessere Performance für den jeweiligen Anwendungsfall
  - Namensraum hilft bei Unterscheidbarkeit

Nehmt euch die Zeit, das auszuschreiben. Es erspart auf lange Sicht Qualen.

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Hello" << endl;
```

```cpp
<cmath>

 FooMath {
 sin(float num) {
return num;
    }
}

int main() {
    // Jetzt gut unterscheidbar
    std::sin(3.5f); // -0.3507
    FooMath::sin(3.5f); // 3.5
}
```

Outputstreams

- Parallel zu `printf()` gibt es in C++ einen neuen Mechanismus: Outputstreams
- Name ist Programm: Sind als beliebig langer Zeichenstrom konzipiert
- Konsolenausgabe über `std::cout`
- Verkettung der Zeichen durch Operator `<<`
- Neue Zeile mit `std::endl`
- Besonderheit: Setzen von Manipulator-Flags
  - „Einfärbung“ des Stroms, bis Flag wieder geändert wird
  - Beispiel: `std::hex` , um alle folgenden Zahlen hexadezimal darzustellen

```cpp
#include <iostream>
using std::cout, std::endl;

int main() {
    cout << "Hello" << endl;
}
```

Outputstreams

cpp Run ▶

```cpp
#include <iostream>

int main() {
    bool b = true;
    int i = 42;

    std::cout << "1: \t" << b << std::endl;
    std::cout << std::boolalpha;                    // Uminterpretierung von bool
    std::cout << "true: \t" << b << std::endl;
    std::cout << std::noboolalpha;                  // Revertierung
    std::cout << "1: \t" << b << std::endl;

    std::cout << "Dec: \t" << i << std::endl;       // Default ist std::dec
```

(Eine Liste aller Manipulator-Flags ist hier verlinkt)

Inputstreams

- Analog zu Outputstreams gibt es Inputstreams
- Funktionsweise wie bei Outputstreams, aber in umgekehrter Richtung
- Einlesen von der Konsole: `std::cin`
- **Einlesen** eines Zeichenstroms und Abspeichern in Variable
- **Achtung:** Der Operator zum Verketten ist umgedreht → `>>`

```cpp
#include <iostream>
using std::cout, std::endl;
using std::cin;

int main() {
    char buffer[24];
    cout << "Name: ";
    cin >> buffer;
    cout << buffer << endl;
    return 0;
}
```

Alternative: `std::print`

- Outputstreams sind mächtiges Werkzeug
- Aber: Für einfache Ausgabe oft zu mächtig und zu viel Schreibaufwand
- Seit C++23: `std::print()`
- Verbindet Einfachheit von `printf()` mit Nützlichkeit von Outputstreams
- Erkennt Parameter über Position in Parameterliste und konvertiert diese

```cpp
#include <print>

int main() {
    // Achtung: Nullbasiertes Indizes
    std::print("B: {1}, I: {0}, S: {2}", 1, true, "Hello");
    return 0;
}
```

Header

- C-Header sind in C++ jeweils in zwei Ausführungen vorhanden
  1. Bereits bekannte Form: `stdio.h`
  2. Version für C++: `cstdio`

     Mit Präfix `c`, ohne Suffix `.h`
- Unterschied: C++-Version kapselt im Namensraum `std` → `std::printf;`
- Mit anderen Worten: C-Variante „verschmutzt“ den globalen Namesraum

Zeiger++

- In C: Zeiger
- In C++: Zeiger + Referenzen
- Funktional: Beide verweisen auf einen Speicherbereich
- Notation: Referenzoperator & ersetzt Zeigerstern *
- Hauptunterschied
  - Zeiger sind eigenständige Typen (mit eigenem Speicher, z.B. auf dem Stack)
  - Wert des Zeigers ist der verwiesene Speicherbereich
  - Wert kann verändert werden (→ Tor zur Hölle 😈)
- Referenzen sind **nur ein Alias** der Variable, **keine** Dereferenzierung möglich!

```
1  void swap (int *a, int *b); // Alt
2  void swap (int &a, int &b); // Neu
3
4  int a = 1;
5  int &a_ref = a;      // Alias von a
```

Beispiel - Referenzen

cpp Run ▶

```cpp
#include <iostream>

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int a = 1, b = 2;
    std::cout << "A: " << a << ", B: " << b << std::endl;
    swap(a, b);
    std::cout << "A: " << a << ", B: " << b << std::endl;

}
```

Funktionen++

- C++ ermöglicht das sogenannte „Überladen“ von Funktionen (*Overloading*)
- Gleichen Funktionsnamen mit verschiedenen Parametern zu überladen
- Geht also auch ohne Namensräume

```cpp
#include <iostream>

using std::cout, std::endl;

void print_num(double d);
void print_num(int i);

int main() {
    cout << print(3.5) << endl;
    cout << print(1) << endl;
}
```

- Achtung: Gilt nur für die Parameter
- Rückgabewert **wird nicht berücksichtigt**

```cpp
void print_num(int i);
int  print_num(int i); // Error
```

Default-Werte für Funktionsparameter

- Weitere Neuerung: Parameter können Default-Werte haben
- Beginnend bei letztem Parameter können für Parameter Rückfallwerte verwendet werden
- Derartige Parameter müssen nicht explizit im Aufruf genannt werden

```
1  int add1(int i, int j, int k = 2);           // Ok
2  int add2(int i, int j = 1, int k = 2);       // Ok
3  int add3(int i = 0, int j = 1, int k = 2);   // Ok
4  int add4(int i = 0, int j, int k = 2);       // Error, falsche Reihenfolge
5
6  add3();          // Rückgabe: 3
7  add3(1);         // Rückgabe: 4
8  add3(1,4);       // Rückgabe: 7
9  add3(1,2,3);     // Rückgabe: 6
```

Überladung von Funktionen hinter den Kulissen

```c
// Dieser C-Quellcode...
int add(int i, int j) {
    return i + j;
}

int sub(int i, int j) {
    return i - j;
}

int main() {
    add(1,2);
    sub(2,1);
}
```

```
// ... erzeugt folgende Symbole
add
sub
main
```

```c
// Dieser C++-Quellcode...
int add(int i, int j) {
    return i + j;
}

int sub(int i, int j) {

    sub(2,1);
}
```

```
// ... erzeugt folgende Symbole
_Z3addii
_Z3subii
main
```

In C++ codiert der Symbolname u. a. Parameter 💡

Name Mangling

- Codiert u. a. Namensraum, Rückgabewert, Funktionsnamen, Parameter und deren Typen in Symbolname

  → *C++ Name Mangling*

- Achtung: Die Kodierung ist nicht standardisiert
  - Unter anderem abhängig von Compiler und Plattform
  - GCC und Clang verwenden unter Linux/Unix die Itanium C++ ABI für Name Mangling (Link)
  - Microsoft hingegen definiert seine eigene ABI für Windows

```
1  // Dieser C++-Quellcode...
2  int add(int i, int j) {
3      return i + j;
4  }
5
6  int sub(int i, int j) {
7      return i - j;
8  }
9
10 int main() {
11     add(1,2);
12     sub(2,1);
13 }
```

```
1  // ... erzeugt folgende Symbole
2  _Z3addii
3  _Z3subii
4  main
```

Name Mangling meets Reality

- Situation: Existierendes C-Projekt mit C++-Code kombinieren
- Problem
  - C++ nutzt *Name Mangling*, C hingegen nicht
  - Linker kann hier die Namen der Symbole nicht richtig auflösen
- Lösung: Rückfall auf C-Namensschema mithilfe von `extern "C"`
- Definierte Bereiche verwenden *C Linkage*

```
1  // Ohne Name Mangling...
2  int add(int i, int j);
3  int sub(int i, int j);
4  int main();
```

```
1  // ... werden C-Symbole erzeugt
2  add
3  sub
4  main
```

(Das C-Namensschema ist allgemein sehr einfach und wird deswegen für die Interoperabilität zwischen verschiedenen Programmiersprachen sehr oft verwendet.)

Abschalten des *Name Manglings*

Markieren einzelner Funktionen/Blöcke

```
1  // Markierung einer Funktion
2  extern "C" void print(int i);
3
4  // Alternativ: Blockweise Markierung
5  extern "C" {
6      void foo1(double d);
7      void foo2(char c);
8  }
```

Entfernen des Name Manglings für Header

```
1  // Kein #pragma once -> Komplett kompatibel
2  // Gut für Projekte mit C und C++
3  #ifndef HEADERNAME_H
4  #define HEADERNAME_H
5
6  #ifdef __cplusplus  // Nur in C++ definiert
7  extern "C" {        // Block-Beginn
8  #endif
9
10 /* Header-Code */
11
12 #ifdef __cplusplus
13 }                   // Block-Ende
14 #endif
15 #endif // HEADERNAME_H
16 // Dateiende
```

Strings

- In C++ eine Abstraktion für Strings → `std::string`
- Kapselt C-String `char *` bzw. `char []` in Klasse
- Zugriff auf C-String mit `c_str()`
- Neuer Komfort: `size()` → Gibt Größe aus
- `std::string` bietet einige Vorteile
  - Einfache String-Konkatenation: `str += " World";`
- Für vollständige Liste der Funktionen in Referenz nachschauen

```
1  std::string str = "Hello";
2  std::cout << str;
```

Range-based for-loops

- In C++ gibt es eine Abstraktion der bereits bekannten `for` -Schleifen:

  Range-based for-loops

- Gedacht als Tipperleichterung für komplexere Daten-Container wie `std::vector` (*kommt gleich*)
- Vorteil: Es muss nicht mehr auf die Indizes geachtet werden
- Erreicht dies unter der Haube über sogenannte Iteratoren (hier **nicht** weiter Thema)

cpp Run ▶

```cpp
#include <string>
#include <iostream>
using std::cout, std::endl;

int main() {
    std::string str = "Hello World";
    for (char c: str) {
        // Print every character
        cout << c;
    }
    cout << endl;
}
```

Range-based for-loops

- Achtung: Range-based for-loops haben eine kleine Tücke
- Die Schleifenvariable wird standardmäßig bei *jedem* Durchlauf neu initialisiert (**By Value**!)
  - Schlecht bei großen Datenstrukturen (Structs/Klassen)
- Besser: Variable als Referenz deklarieren
- Und falls Daten nicht verändert werden sollen: Zusätzliches `const`

```
1  // Okay für primitive Datentypen:
2  // By-Value
3  for (char c: str) {
4      cout << c;
5  }
6  cout << endl;
```

```
1  // Besser für komplexe Datentypen:
2  // Read-Only und By-Reference
3  for (const char & c: str) {
4      cout << c;
5  }
6  cout << endl;
```

Moderne Felder

- Klasse `std::vector` kapselt Eigenschaften eines Feldes
- Zugriff mittels `[]`, kann beliebige Datentypen aufnehmen und dynamische Größe 🤯
  - Alternativ: Zugriff mit Funktion `at()`, da `[]` bei falschem Index undefiniert ist

cpp Run ▶

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> nums;

    nums.push_back(3);
    nums.push_back(42);
    nums.push_back(-1);

    nums[2] += 10;

    for (const int & i : nums) {
```

Zusammenfassung

- Übergang von zu C zu C++ 🥳
- „Hello World“-Beispiel in C++
- Namensräume
- Überladen von Funktionen
- Default-Parameter
- Range-based for loops
- `std::string` und `std::vector`

Kapitel 12 - Einführung in C++