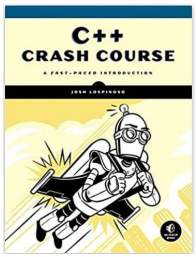
Kurseinheit 1: Von C zu C++

1. Historie C und C++
2. Super C - Features
3. Typsystem von C++

Literatur

Textbook



[Los19] Lospinoso, Josh: *C++ Crash Course: A Fast-Paced Introduction,* No Starch Press, 2019

Zweite Auflage angekündigt (C++20) … coming soon?

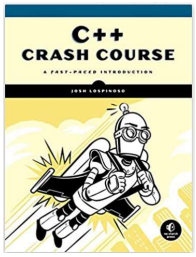
Als E-Book vorhanden!

Sekundärliteratur

[Mey14]	Scott Meyers: *Effective Modern C++: 42 Specific Ways to Improve Use of C++11 and C++14*, O'Reilly Media, 2014 Als E-Book vorhanden	Etwas veraltet, aber dennoch zu empfehlen. Klassiker.
[Shv20]	Alexander Shvets: Dive Into Design Patterns, 2020 Nur als E-Book https://refactoring.guru/design-patterns/book Unbedingt kaufen: 20 Euro	Gehirngerechte Einführung in Design Patterns! Ebenso in zip-Datei enthalten: C++ Code der Beispiele

Literatur

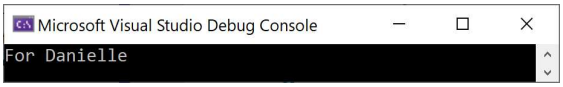
Danksagung im Buch



[Los19] Lospinoso, Josh: *C++ Crash Course: A Fast-Paced Introduction,* No Starch Press, 2019

Code-Formatierung aus dem Buch …

```
int main() {
   auto i{ 0x01B99644 };
   std::string x{ " DFaeeillnor" };
   while (i--)
std::next_permutation(x.begin(),
x.end());
   std::cout << x;
}
```



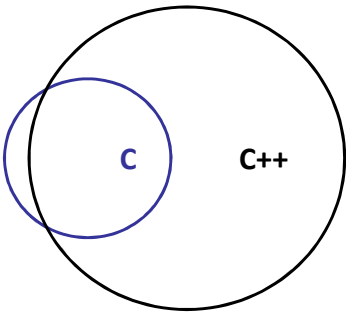
C/C++ Coding Styleguide wird im Kurs verwendet.

1. Historie C und C++

Historie

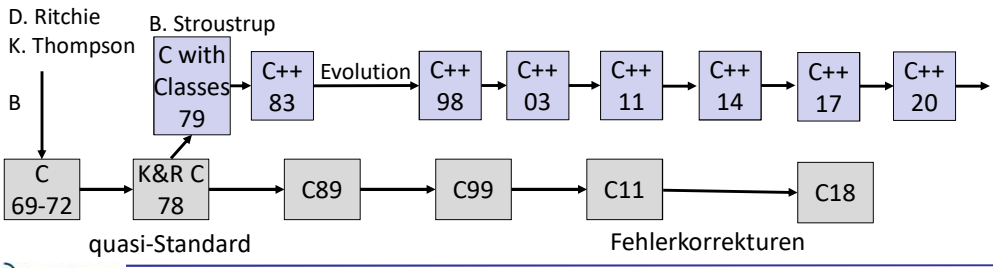
C++ erweitert C um prozedurale und objektorientierte Sprachfeatures.
++ ist dabei der Post-inkrementoperator.

C++ ist eine **Multi Purpose Language.**



C enthält allerdings auch einige Sprachfeatures, die nicht in C++ enthalten sind. Dies ist zu berücksichtigen, wenn Code in C und C++ verwendet werden soll.

C noch führend bei **Deeply Embedded Systems**.



Umstieg

Der Umstieg von C zu(m) (OO-)C++ fällt langjährigen C-Entwicklern schwer!

C ist eine **rein prozedurale** Sprache!

Funktionsweise einer CPU (Prozessor) spiegelt sich in der Sprache wieder.

C++ erweitert C um wenige prozedurale und viele **objektorientierte** (OO) Features.

In den OO-Features spiegelt sich die menschliche Denkweise wieder.

Objekte von Hunden

Klasse

Dog

Aber: Die ersten C++ Compiler übersetzen erst in C, ein C-Compiler übersetzte dann in Maschinencode. OO-Features können mit mehr oder minderem Aufwand in C transformiert werden. Viele C-Programmierer nutzen dies auf Embedded Systems!

C++ Schnellüberblick

C++ hat:

Wenige Non-OO Erweiterungen – **Super C und erweitertes Typsystem**

Hybride Sprache

Umfangreiche OO-Erweiterungen

- Klassen und Objekte
  - Information Hiding
  - Vererbung (Einfach- und Mehrfachvererbung)
  - Kapselung
  - Polymorphismus
  - Interfaces (indirekt)
  - Konstruktor und Destruktoren
  - Unterschiedliche Beziehungstypen (Assoziation, Aggregation, Komposition)
- Exception Handling
- Templates
- Virtuelle Funktionen
- Umfangreiche Bibliothek (STL – Standard Template Library -> Container und Algorithmen)
- …

Streams

In C++ können z.B. zur einfachen Ein- und Ausgabe auf der Konsole Streams verwendet werden. Für einfache Ausgaben erscheint dies einfacher, für formatierte Ausgaben sei auf die Dokumentation verwiesen.

```
#include <iostream>

int32_t s32Input;

std::cout << "Hello World" << std::endl;
std::cout << "Please enter value for s32Input: ";
std::cin >> s32Input;
std::cout << std::endl << "Value auf s32Input: " << s32Input << std::endl;
```

Die bitorientierten Operatoren << und >> dienen hier als Pipe-Zeichen. Was fließt wohin?
cout ist der Bildschirm, **cin** die Tastatur, std:: siehe später!
cerr ist Standardfehlerausgabe (i.d.R. Bildschirm), **clog** ist Standardprotokoll (i.d.R. Bildschirm) – können natürlich mit der Funktion rdbuf umgeleitet werden.

Es kann natürlich auch weiterhin mit printf gearbeitet werden. Die alten C-Zöpfe (printf, casts, etc.) sollten Sie aber bald abschneiden.

Name Spaces - Namensräume

Funktionen, Klassen oder Variablen mit dem **gleichen Namen** können in unterschiedlichen Namensräumen vorkommen.

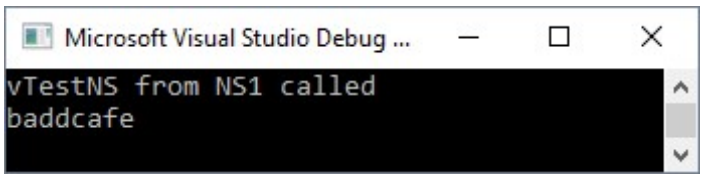
```
namespace NS1
{
   uint32_t u32Var = 0xDEADC0DE;
   void vTestNS(void)
   {
      std::cout << "vTestNS from NS1 called" << std::endl;
   }
}

namespace NS2
{
   uint32_t u32Var = 0xBADDCAFE;
   void vTestNS(void)
   {
      std::cout << "vTestNS from NS2 called" << std::endl;
   }
}
```

Name Spaces - Namensräume

Mit **Zugriffsoperator ::** kann auf die Funktionen oder Variablen des Name Spaces zugegriffen werden.

```
NS1::vTestNS();
std::cout << std::hex << NS2::u32Var << std::endl;
std::cout << std::dec;
```

hex schaltet auf hexadezimale Ausgabe um, dec wieder auf dezimal.



Die Verwendung von Name Spaces gehört auch zum "C++ Knigge" (Benimmbuch von Adolph Knigge, 1788).

Beispiel: Firma verwendet in ihrer Anwendung die Struktur (oder Klasse) Customer. Für die Abwicklung der Werbepost soll eine Lib X der Firma Y integriert werden. Diese hat auch eine Struktur (oder Klasse) Customer. Jetzt gäbe es ohne Name Spaces Konflikte!

Nutzung von Namensräumen

cin, cout, cerr, clog, endl und hex sind im Name Space **std** enthalten. Es gibt drei Möglichkeiten der Nutzung:

- **Pauschale Nutzung**

```
using namespace std;
// … more code
cout << "hello" << endl;
```

- **Nutzung mit qualifiziertem Namen**

```
std::cout << "hello" << std::endl;
```

- **Deklaration ausgewählter Teilbereiche**

```
using namespace std::out;
using namespace std::endl
// … more code
cout << "hello" << endl;
```

Die Nutzung mit qualifiziertem Namen ist vorzuziehen. Für den Umsteiger/Anfänger wird der Code dadurch nicht unbedingt lesbarer.

Headerdateien

Korrespondierende Cpp-Headerdateien →

```
#include <stdexcept>
#include <iostream>
#include <cstring>
#include "SimpleString.h"
```

} Neue Cpp-Headerdateien

Es **könnten** weiterhin die C-Headerdateien verwendet werden. Es bietet sich allerdings an, die entsprechenden Cpp-Headerdateien <c…> zu verwenden, da dort alle Funktionen im **Namensraum std** enthalten sind!

C-Headerdateien

```
#include <string.h>
#include <stdio.h>
…
```

→

Korrespondierende Cpp-Headerdateien

```
#include <cstring>
#include <cstdio>
…
```

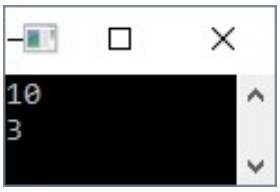
Eine wichtige Cpp-Headerdatei ist <iostream>. Diese wird im Skript hauptsächlich für die Konsoleneingabe und –ausgabe verwendet.

Default-Werte

Funktionen können auch Default-Werte haben. Es können allerdings nur die letzten Parameter mit Default-Werten versehen werden.

```
int32_t GetSum(int32_t s32V1, int32_t s32V2,
               int32_t s32V3 = 0, int32_t s32V4 = 0);
```

```
std::cout << GetSum(1, 2, 3, 4) << std::endl;
std::cout << GetSum(1, 2) << std::endl;
```



Default-Werte nur in die Deklaration, nicht in die Definition

Nutzen ist sicherlich in Einzelfällen vorhanden.

Referenzen

Neben Zeigern können – und sollten – Referenzen verwendet werden. Referenzen bieten folgende **Vorteile**:

- Können nie Null sein. Sanity-Checks sind daher nicht notwendig.
- Müssen daher bei der Definition initialisiert werden.
- Code erscheint lesbarer, da Dereferenzierung nicht notwendig ist.

```
void vSetValue1(uint32_t* pu32Val, const uint32_t cu32Con)
{
   *pu32Val = cu32Con;
}
```

```
void vSetValue2(uint32_t& ru32Val, const uint32_t cu32Con)
{
   ru32Val = cu32Con;
}
```

Der Compiler erzeugt für beide Varianten den **gleichen** Maschinencode!

Referenzen

Deklaration

```
void vSetValue2(uint32_t& ru32Val, const uint32_t cu32Con2);
```

Funktionsaufruf

```
uint32_t u32X2 = 0U;

vSetValue2(u32X2, 73);
```

→ Hier ist nicht unbedingt ersichtlich, dass u32X2 per Reference übergeben wird.

Definition

```
void vSetValue2(uint32_t& ru32Val, const uint32_t cu32Con)
{
   ru32Val = cu32Con;
}
```

Insbesondere an der Nutzung von Referenzen erkennt man, wie C++ schon verinnerlicht wurde. Wer noch mental in der C-Welt steckt, verwendet eher weiterhin noch Zeiger!

Überladen von Funktionen

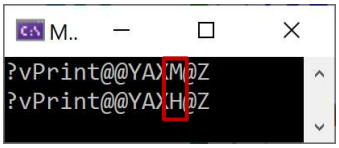
Funktionen können in C++ überladen werden! Gleicher Funktionsname – unterschiedliche Parameter! Rückgabewert bleibt dabei unberücksichtigt! Nicht möglich in C!

```
void vPrint(f32_t f32Val);
void vPrint(int32_t s32Val);
```

```
typedef float  f32_t;
```

```
void vPrint(f32_t f32Val)
{
   wprintf(L"%s
}
```

```
void vPrint(int32_t s32Val)
{
   wprintf(L"%s\n", _CRT_WIDE(__FUNCDNAME__));
}
```



In C++ werden die Datentypen der Übergabeparameter in den Funktionsnamen codiert (Name Mangling). Dies macht später Probleme, wenn C und C++ Module gemischt werden.

Funktionstemplates

```
template <class T>
void vSwap(T& tra, T& trb)
{
   T ttemp = tra;
   tra = trb;
   trb = ttemp;
}
```

Compiler generiert für jeden verwendeten Typparameter T eine eigene Funktion (Überladen von Funktionen).
Darin wird der generische Typ T durch den konkreten Typ ersetzt.
Die verwendeten Operatoren für diesen Typ T müssen aber existieren.
Hier im Beispiel muss der Zuweisungsoperator für den Typ T vorhanden sein.

```
vSwap(u32Val1, u32Val2);
```

```
void vSwap(uint32_t& tra, uint32_t& trb)
{
   uint32_t ttemp = tra;
   tra = trb;
   trb = ttemp;
}
```

```
vSwap(f32Val1, f32Val2);
```

```
void vSwap(f32_t& tra, f32_t& trb)
{
   f32_t ttemp = tra;
   tra = trb;
   trb = ttemp;
}
```

Mehrere Typparameter sind auch möglich.

2. Super C

Ranged Based Loop

```
uint32_t au32Array[] = { 400U, 300U, 200U, 100U };
uint32_t u32C;

for (u32C : au32Array)
{
   std::cout << u32C << std::endl;
}
```

Durchläuft alle Elemente des Arrays. Später wird dies dann noch auf sogenannte Container (Liste, Vectoren, Sets, usw.) angewendet.

Code wird dadurch lesbarer.

3. Typsystem von C++

Variablendefinition innerhalb eines Blocks

C++ **und** C99 bietet bezüglich Sichtbarkeit/Gültigkeit von Variablen noch eine weitere Ebene an: Sichtbarkeit/Gültigkeit innerhalb eines Code-Blocks ({ }).

```
// Like in C 99 (which is often used) definition of Variables
// inside a block
{
   uint32_t u32V = 0xA5A5A5A5;
}
// not visible outside
// u32V = 42U;
```

```
//Most common is the usage inside a loop
for (uint32_t u32C = 0U; u32C < 10U; u32C++)
{
   ; //do something
}
```

Dies compiliert in C89 nicht!

Das Konzept zuerst alle Variablen zu definieren wird hier aufgegeben.

Strengere Typisierung

Der Compiler überwacht die Datentypen bei Zuweisung deutlich strenger.
Beispiel: Pointer

```
void* pvVal = nullptr;
f32_t* pf32Val = nullptr;
uint32_t u32Val = 0xDEADC0DE;
uint32_t* pu32Val = &u32Val;

pvVal = pu32Val; // okay in C++
// pu32Val = pvVal; // not okay in C++ ... okay in C
// pf32Val = pu32Val; // not okay in C++ ... okay in C (warning)
```

Einführung eines Nullzeigers!
nullptr statt NULL

```
uint32_t* pu32Val = (uint32_t*) malloc(100 * sizeof(uint32_t));
```

Returnwert von malloc (void*) kann in C++ nicht einem non-void Pointer zugewiesen werden. Daher ist in C++ der Cast notwendig.

C-Programme könnten später mit einem Cpp-Compiler übersetzt werden. Daher sollte schon vorbereitend dies berücksichtigt werden.

Einfache Datentypen

Neuer Datentyp bool: true und false als Werte

```
bool bRet = true;
//bRet = 23; warning in C++
std::cout << "Sizeof bool is " << sizeof(bool) << std::endl;
//sizeof bool is 1
```

Seit C99 gibt es auch einen bool-Datentyp in C: **_BOOL** .stdbool.h behebt die Inkompatibilität.

Typedefs für Ganzzahltypen:
Seit C++11 gibt es den <cstdint> Standardheader mit typedef für Ganzzahlendatentypen und deren Limits. Scheint in MSVS 2022 automatisch eingebunden zu werden.

https://en.cppreference.com/w/cpp/header/cstdint

Ein großes Anwendungsgebiet für C++ sind die Embedded Systems. Dort werden diese typedefs nahezu durchgängig eingesetzt.

```
int32_t s32X1 =  42;
uint32_t u32X1 = 42U;
```

Fließkommazahlen selbst “typedefen”

```
typedef float  f32_t;
typedef double f64_t;
```

Structs, Unions, Enums

Sind neue Datentypen (typedefs wie in C nicht notwendig)! **s, u, e jetzt voranstellen!**

```
struct sConfig
{
   uint32_t u32NumberOfIniFiles;
   uint32_t u32MaxEntriesInFile = 200U;
   uint32_t u32MinEntriesInFile;
};

union uConvert
{
   uint32_t u32Number;
   char acNumberAsText[10];
};

enum eTowns { Offenburg, Strasbourg,
              New_York };
```

Ggf. typedefs für Pointer anlegen:

```
typedef sConfig* psConfig_t;
```

Sowohl neuer Datentyp als auch Variable sollte den Präfix erhalten (C/C++ Coding Styleguide)

```
sConfig sConfig1 = { 0 };
uConvert uConvert1 = { 0 };
eTowns eTowns1 = Strasbourg;
```

Structs haben in C++ eine geringere Bedeutung, da dort eher Klassen verwendet werden (siehe später).

Designed Initialization von Structs/Unions

Erst ab C++20: Compiler schon umgestellt?

```
struct sConfig
{
   uint32_t u32NumberOfIniFiles;
   uint32_t u32MaxEntriesInFile = 200U;
   uint32_t u32MinEntriesInFile;
};
```

```
union uConvert
{
   uint32_t u32Number;
   char acNumberAsText[10];
};
```

```
//Designed Initialization, Order is important, Default Values can be skipped
//Properties, C/C++, Language, C++ Language ISO C++20 Standard (/std:c++20)
//C++20 Standard
sConfig sConfig2 = { .u32NumberOfIniFiles = 50U, .u32MinEntriesInFile = 0U};
```

Designed Initialization gibt es auch in **C99**. Dort muss allerdings **nicht** die Reihenfolge eingehalten werden. **Kompatibilitätsproblem!**

Designed Initialization bei Unions: Nur ein Element!

```
uConvert uConvert3 = { .u32Number = 42U };
```

Scoped Enums

Unscoped Enums in C:

```
enum eTowns { Offenburg, Strasbourg, New_York };
enum eFamousHAW { Offenburg, Hagenau}; //Conflict in C++, redefinition
```

Namenskonflikt in C++

Dieser Namenskonflikt lässt sich durch **Scoped Enums** in C++ (seit C++11) lösen.

```
enum class eColors {Red, Orange, Blue, White};
enum class eFruits {Kiwi, Orange, Banana};
```

Enums sollten grundsätzlich in C++ – auch wenn kein Konflikt vorhanden ist - als Scoped Enums deklariert werden. Unscoped Enums führen in C++ zu Warnungen.

```
eColors eColor1 = eColors::Orange;
```


Initialisierung von Variablen, komplexen Datentypen und Objekten

In C++ gibt es **vier** Möglichkeiten Variablen, komplexe Datentypen und Objekte zu initialisieren.

```
//There are four ways to initialize
int32_t s32X1 = 0;
//Braced initialization is always appropriate
//This was added, because it avoids the "most vexing parse" later
//with objects
int32_t s32X2{};
int32_t s32X3 = {};
//Parentheses initialization
int32_t s32X4(0);
```

Allgemeine C++ Regel:
Verwenden Sie zum Initialisieren nur noch Braced Initialization – Brace yourself!

Most Vexing Parse:
Wird später im Zusammenhang mit dem Klassenkonzept gezeigt.

auto

In C gibt es das Default-Keyword auto für lokale Variablen. Ab C++11 gibt es das Keyword auto für einen automatischen Datentyp der sich aus einer Initialisierung ergibt.

```
//C++ is strongly typed language. C++ affords its compiler a lot of
//information. auto keyword tells the compiler to perform such divination
//(Voraussage)
auto aVar3{ 42 };      // int32_t
auto aVar4{ 12L };     // long -> 4 Bytes at 32 and 64 Plattform
auto aVar5{ 3.0F };    // f32_t
auto aVar6{ 10.0 };    // f64_t
auto aVar7{ false };   // bool
auto aVar8{ "Hello" }; // pointer to char
```

a als Präfix im C/C++ Coding Styleguide (Bei einem a für Array folgt ein Datentyp (z.B au32Array) – daher noch gut unterscheidbar.

Autotypen vereinfachen das Code Refactoring: Namensänderungen von Structs und Klassen usw. müssen nicht im Code “nachgezogen” werden, da auto die neuen Typen übernimmt.

auto ranged loop

Das Keyword auto kann auch in ranged loops eingesetzt werden. Dies nennt sich dann auto ranged loop!

```
uint32_t au32Array[] = { 0U, 42U, 73U, 97U };
for (auto aVar : au32Array)
{
   std::cout << aVar << std::endl;
}
```

Im späteren Kurseinheiten wird gezeigt, wie mächtig die auto ranged loop gerade beim “Durch-Iterieren” von Collections/Containern ist.

Merken Sie sich dieses mächtige C++ Feature!

Strings

Zeichenketten wurden bisher in C mittels char-Arrays abgebildet. Dabei markierte das EOS-Zeichen (0x00) das Ende einer Zeichenkette. In string.h finden sich umfangreiche Funktionen, um auf diese char-Arrays zuzugreifen. Problematisch waren immer die Pufferüberläufe, was durch spezielle Varianten der Funktionen (_s) vermieden werden sollte.

C++ enthält in der sogenannten STL (Standard Template Library) eine eigene Klasse, um Zeichenketten (Strings) zu verarbeiten.
https://www.cplusplus.com/reference/string/string/

```
//C++ has it own String class with useful concatenation features
std::string strSomething = "Hello ";
strSomething += "World";
//Performance will be discussed later in the course
std::cout << "strSomething" << std::endl;
```

Details zu Strings in späteren Kurseinheiten!

C++ ist
- eine Multi Purpose Language (von Embedded Systems bis Facebook)
- eine sehr anspruchsvolle Programmiersprache
- am Arbeitsmarkt gesucht

```
//C++
for(;"ever";);
```