

Programmieren in C++

Christian Lang (Lac)

13. September 2019

C++ Grundlagen

- Öffentliche Klassen
- Qualifier
- `auto`
- Global vs. Lokal und `static`
- Parameter-Übergabe
- Casting
- Strings
- Konsolen-Output
- Range-based Loop
- Initialisierungs-Listen
- Präprozessor

Öffentliche Klassen: Deklaration & Definition

- Alle Member (Variablen und Methoden) sind per default **public**.
- Konstruktoren haben **kein** Return-Typ.
- **Default**-Konstruktor hat keine Parameter.

```
1  struct PublicClass {  
2      PublicClass();  
3      PublicClass(int initial);  
4  
5      void Clear();  
6  
7      int value;  
8  };
```

```
1  PublicClass::PublicClass() {  
2      Clear();  
3  }  
4  
5  PublicClass::PublicClass(int v)  
6      : value(v)  
7  {}  
8  
9  PublicClass::Clear() {  
10     value = 0;  
11 }
```

Öffentliche Klassen: Verwendung

- Zugriff auf Member per `.`-Operator
- Alles ist zugreifbar.

```
1  #include "public_class.h"
2
3  int main() {
4      PublicClass p;
5      PublicClass i(2);
6
7      p.value = 3;
8      i.Clear();
9  }
```

- Automatische Typ-Evaluierung
- Kann auch durch weitere Qualifier (`const`, `*`, `&`) spezialisiert werden.

```
1  auto i = 2;      // int
2  auto d = 2.23    // double
3
4  PublicClass container;
5  const auto p = container;
```

- Unveränderbarkeit: `const`
- Bekannt zur Compiletime: `constexpr`
- Speicher mit Nebeneffekten: `volatile`

```
1  constexpr size_t kNumberOfBytes = 8;  
2  
3  volatile uint8_t register[kNumberOfBytes];  
4  
5  const uint8_t value = register[0];
```

Global vs. Lokal und static

```
1  // öffentlich globale Variable
2  int pub_global = 1;
3
4  // Variable nur in dieser Compilation-Unit sichtbar
5  static private = 1;
6
7  struct Members {
8      static int per_class;
9      int per_instance;
10 };
11
12 int main() {
13     int local = 1;
14 }
```


- Parameter-Übergabe auf **viele** Arten möglich.
- Immer kombinierbar mit **Qualifier**.

```
1 void Func(int by_value,  
2         const int& by_const_reference,  
3         int* by_pointer  
4     );
```

- Für **In**-Parameter: By Value oder By Const-Reference
- Für **In/Out**-Parameter: By Pointer
 - Dadurch ist ein Out-Parameter beim Aufruf durch das **&** erkennbar

- In C gab es nur (int)variable.
- Alle Varianten von Casts wurden in C++ separiert:

Cast	Anwendung
<code>static_cast<T>()</code>	normale Typ-Casts
<code>const_cast<T>()</code>	<code>const</code> hinzufügen / entfernen
<code>dynamic_cast<T>()</code>	Down-Cast in Klassenhierarchie
<code>reinterpret_cast<T>()</code>	Keine Compiler-Checks

- Immer den möglichst **spezifischen** Cast bevorzugen.

- In C gab es C-Strings:

```
1 char name[];  
2 // oder  
3 const char* name
```

- Container `std::string` in Standard-Library.

```
1 #include <string>  
2  
3 std::string name;  
4 const std::string description = "test";  
5  
6 name = description;  
7 auto sub = name.substr(0, 2);
```

- IO-Stream Library in Standard-Library.
- Typen mit <<-Operator direkt **print**-bar.

```
1  #include <iostream>
2
3  std::cout << "text on stdout" << std::endl;
4  std::cerr << "text on stderr" << std::endl;
5
6  const int v = 2;
7  std::cout << "v=" << v << std::endl;
```

Range-based Loop

- Ohne Index oder Iteratoren
- `auto` verwenden
- Funktioniert auf `rohen Arrays` und Container

```
1  int container[] = { 1, 2, 3 };
2  for (const auto& value : container) {
3      ...
4  }
```

- Kombinieren mit `Structured bindings`

```
1  std::map<int, double> map;
2  for (const auto& [key, value] : map) {
3      ...
4  }
```

- Ermöglicht Ausdrücke wie: `MyClass c = { 1, 2, 3 };` für **eigene Typen** zu implementieren.
- Setzt **Vereinheitlichte initialisierungs-Syntax** voraus.

```
1  #include <initializer_list>
2  struct MyClass {
3      MyClass(int init);
4      MyClass(const std::initializer_list<int>& init);
5  };
6
7  MyClass c(1);           // ctor A
8  MyClass c{1};          // ctor A
9  MyClass c = { 1, 2, 3 }; // ctor B
```

- Eingeleitet durch `#`
- Text-Ersetzung
- Abhängigkeiten: `#include`
- Makros: `#define`
- Präprozessor-Befehle: `#pragma ...`

```
1  #pragma once
2
3  #include <vector>
4
5  #ifdef HW_ENABLE
6  #define REGISTER_ADDRESS 0x03A2
7  #endif
```