

Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

17. Dezember 2018

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

2 Wiederholung: Programmieren in C

3 Einführung in C++

...

3.12 Typ-Konversionen

4 Standard-Bibliotheken (STL)

4.1 Container-Templates

4.2 Iteratoren

5 C++11

5.1 Syntax-Erweiterungen

5.2 Streng typisierte Aufzählungstypen

5.3 Intelligente Zeiger

5.4 R-Wert-Referenztypen

5.5 Lambda-Ausdrücke

6 Die Boost-Bibliothek

7 Plug-In-Architekturen



Änderungen
vorbehalten

3.12 Typ-Konversionen

- In C:

```
char *hello = "Hello, world!";  
uint64_t address = (uint64_t) hello;  
printf ("%s" PRIu64 "\n", address);
```

- alternative Syntax in C++:

```
char *hello = "Hello, world!";  
uint64_t address = uint64_t (hello);  
cout << address << endl;
```

- zusätzlich in C++:

implizite und explizite Typumwandlung zwischen Zeigern auf Klassen

```
dynamic_cast<>()  
static_cast<>()  
reinterpret_cast<>()  
const_cast<>()
```

3.12 Typ-Konversionen

<http://www.cplusplus.com/doc/tutorial/typecasting/>

- Zuweisung: Zeiger auf abgeleitete Klasse an Zeiger auf Basisklasse
→ implizite Typumwandlung möglich
- Zuweisung: Zeiger auf Basisklasse an Zeiger auf abgeleitete Klasse
→ nur explizite Typumwandlung möglich:
`dynamic_cast<>()`, `static_cast<>()`
- implizite Typumwandlungen in der Klasse definieren:
 - Initialisierung durch Konstruktor
 - Zuweisungs-Operator
 - Typumwandlungsoperator
- implizite Typumwandlungen ausschalten:
Schlüsselwort `explicit`

3.12 Typ-Konversionen

<http://www.cplusplus.com/doc/tutorial/typecasting/>

- `dynamic_cast<>()`
Zuweisung: Zeiger auf Basisklasse an Zeiger auf abgeleitete Klasse
explizite Typumwandlung mit Prüfung, ggf. Exception
- `static_cast<>()`
Zuweisung: Zeiger auf Basisklasse an Zeiger auf abgeleitete Klasse
explizite Typumwandlung ohne Prüfung
- `reinterpret_cast<>()`
Typumwandlung ohne Prüfung zwischen Zeigern untereinander
und zwischen Zeigern und Integer-Typen
- `const_cast<>()`
„**const**“ ein- bzw. ausschalten

4 Standard-Bibliotheken (STL)

4.1 Container-Templates

array	Array mit fester Größe
bitset	festes Array von Bits (Booleans)
vector	dynamisches Array
vector <bool>	dynamisches Bit-Array
forward_list	einfach-verkettete Liste
list	doppelt-verkettete Liste
set	binärer Baum
multiset	mehrfache Elemente zulässig
unordered_set	Hash-Tabelle
unordered_multiset	mehrfache Elemente zulässig
map	binärer Baum mit separaten Schlüsselwerten
multimap	mehrere Elemente pro Schlüssel
unordered_map	Hash-Tabelle mit separaten Schlüsselwerten
unordered_multimap	mehrere Elemente pro Schlüssel
stack	Stack
queue	FIFO
deque	<i>double-ended queue</i>
priority_queue	geordneter Push-Pop-Container

4 Standard-Bibliotheken (STL)

4.2 Iteratoren

Pointer-Arithmetik:

```
int prime[5] = { 2, 3, 5, 7, 11 };  
for (int *p = prime; p != prime + 5; p++)  
    cout << *p << endl;
```

Iterator als Verallgemeinerung:

```
array<int, 5> prime = { { 2, 3, 5, 7, 11 } };  
for (array<int, 5>::iterator p = prime.begin (); p != prime.end (); p++)  
    cout << *p << endl;
```

5 C++11

5.1 Syntax-Erweiterungen

Allgemeine Syntax-Erweiterung: Datentyp **auto**

Anwendung auf Iterator:

```
array<int, 5> prime = { { 2, 3, 5, 7, 11 } };  
for (auto p = prime.begin (); p != prime.end (); p++)  
    cout << *p << endl;
```

Spezielle Syntax-Erweiterung: **for**-Schleife über Container mit Referenz statt „Zeiger“

```
array<int, 5> prime = { { 2, 3, 5, 7, 11 } };  
for (auto p: prime)  
    cout << p << endl;
```


5 C++11

5.2 Streng typisierte Aufzählungstypen

- `enum class`

5 C++11

5.3 Intelligente Zeiger

Warum?

- bereits freigegebene Zeiger werden u. U. weiterhin verwendet
- Speicherlecks
- uninitialisierte Zeiger
- `shared_ptr`
- `weak_ptr`
- `unique_ptr`
- `move()`

5 C++11

5.4 R-Wert-Referenztypen

- &&
- move()

Literatur:

- http://thbecker.net/articles/rvalue_references/section_01.html
- <http://www.artima.com/cppsource/rvalue.html>