

# Algorithmen und Datenstrukturen in C/C++

Prof. Dr. rer. nat. Peter Gerwinski

21. April 2022

# Algorithmen und Datenstrukturen in C/C++

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

## 1 Einführung

## 2 Einführung in C++

...

2.6 Namensräume

2.7 Objekte

2.8 Strings

2.9 Templates

2.10 Container-Templates

2.11 Iteratoren

2.12 Exceptions

2.13 Typ-Konversionen

## 3 Datenorganisation

## 4 Datenkodierung

## 5 Hardwarenahe Algorithmen

## 6 Optimierung

## 7 Numerik

## 2.9 Templates

Anwendung desselben Quelltextes auf verschiedene Datentypen

- `template <typename x> ...`
- `template <class x> ...`

Vorsicht: Fehler werden erst bei Instantiierung erkannt!

- Template-Spezialisierung:  
`template <> foo <int> ...`

## 2 Einführung in C++

### 2.10 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

## 2 Einführung in C++

### 2.11 Iteratoren

Pointer-Arithmetik:

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

Iterator als Verallgemeinerung:

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

Mit **auto**-Datentyp:

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

## 2 Einführung in C++

### 2.11 Iteratoren

Iterator als Verallgemeinerung:

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

Mit **auto**-Datentyp:

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

Mit Doppelpunkt-Syntax:

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

## 2.12 Exceptions

```
try
{
    ...
    throw <value>;
    ...
}
catch (<type> <variable>)
{
    ...
}
catch ...
```

- Nach den `catch()`-Statements wird, soweit nicht anders programmiert, das Programm fortgesetzt.
- `throw;` (ohne Wert):  
an übergeordneten Exception-Handler weiterreichen
- C-Äquivalent:  
`setjmp()`, `longjmp()`
- speziell für `<type>`:  
Nachfahren von `class exception`
- veraltet:  
*dynamic exception specifications*

## 2.13 Typ-Konversionen

- In C:

```
char *hello = "Hello, world!";  
uint64_t address = (uint64_t) hello;  
printf ("%s" PRlu64 "\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<>()
```



## 2.13 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`

## 2.13 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