

# Algorithmen und Datenstrukturen in C/C++

Prof. Dr. rer. nat. Peter Gerwinski

11. Juli 2024

# Algorithmen und Datenstrukturen in C/C++

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

- 1 Einführung
- 2 Arrays und Zeiger für Fortgeschrittene
- 3 Langzahl-Arithmetik
- 4 Kryptographie
- 5 Datenkompression
- 6 Einführung in C++, Datenorganisation
- 7 Hardwarenahe Algorithmen



Änderungen  
vorbehalten

## 6.6 Objekte

```
struct TBase  
{  
};
```

```
struct TInteger: public TBase  
{  
    int content;  
};
```

```
struct TString: public TBase  
{  
    char *content;  
};
```

## 6.6 Objekte: Zugriffsrechte

- `public`, `private`, `protected`  
nicht nur Bürokratie, sondern auch Kapselung  
(Maßnahme gegen „Namensraumverschmutzung“)
- **`struct`**: standardmäßig `public`  
`class`: standardmäßig `private`
- `friend`-Funktionen und -Klassen
- Klasse als Namensraum:  
**`static`**-„Member“-Variable  
**`static`**-„Methoden“  
Deklarationen von z. B. Konstanten und Typen

## 6.6 Objekte: Konstruktoren und Destruktoren

- leerer Standard-Konstrutor
- *Copy-Konstruktor*
- Konstruktor-Aufruf als „Initialisierung“
- Konstruktor-Aufruf mit `new`  
Destruktor-Aufruf mit `delete`
- automatischer Destruktor-Aufruf  
beim Verlassen des Gültigkeitsbereichs

## 6.7 Strings

- **#include** <string>
- String-Klasse
- String-Konstante sind **const char \***
- C-kompatiblen String extrahieren: `c_str ()`
- In String schreiben: **#include** <sstream>, `ostringstream`

## 6.8 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> ...`

## 6.9 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



## 6.10 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;
```

## 6.11 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*

## 6.12 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<>()`

## 6.12 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`

## 6.12 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

## 6.13 Intelligente Zeiger

### Warum?

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

## 6.13 Intelligente Zeiger

### Wie?

- R-Wert-Referenztypen: `&&`
- `move()`-Funktion

### Literatur:

- [http://thbecker.net/articles/rvalue\\_references/section\\_01.html](http://thbecker.net/articles/rvalue_references/section_01.html)
- <http://www.artima.com/cppsource/rvalue.html>