

# Algorithmen und Datenstrukturen in C/C++

Prof. Dr. rer. nat. Peter Gerwinski

20. Juni 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++
- 7 Datenorganisation

...



Änderungen  
vorbehalten

# 5 Datenkompression

## 5.2 Algorithmen

### Verlustfreie Kompression

- Run-Length Encoding (RLE)
- Entropie-Kodierung
  - Morsecode
  - UTF-8
  - Huffman-Kodierung
  - Lempel-Ziv-Welch-Algorithmus
- Differenzbildung

### Verlustbehaftete Kompression

- JPEG

### Weitere Aspekte

- Pixel- vs. Vektorgrafik
- E-Mail-Attachment vs. HTTP

Hochschule Bochum  
Bochum University  
of Applied Sciences



Hochschule Bochum  
Bochum University  
of Applied Sciences



Hochschule Bochum  
Bochum University  
of Applied Sciences



# 6 Einführung in C++

## 6.0 Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen



Hardware und/oder Betriebssystem

- Hardware direkt ansprechen und effizient einsetzen
- ... bis hin zu komplexen Software-Projekten

→ Man kann Computer vollständig beherrschen.

# 6 Einführung in C++

## 6.0 Was ist C?

### Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen
- Hardware direkt ansprechen und effizient einsetzen
- ... bis hin zu komplexen Software-Projekten
- leistungsfähig, aber gefährlich

### „High-Level-Assembler“

*C makes it easy to shoot yourself in the foot.*

- kein „Fallschirm“
- kompakte Schreibweise

Bjarne Stroustrup, ca. 1986

[http://www.stroustrup.com/bs\\_faq.html](http://www.stroustrup.com/bs_faq.html)  
[#really-say-that](#)

### Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln
- kein „Fallschirm“

# 6 Einführung in C++

## 6.1 Was ist C++?

Etabliertes Profi-Werkzeug

- kompatibel zu C

*C++ is a better C.*

C++ unterstützt

- *objektorientierte Programmierung*
- *Datenabstraktion*
- *generische Programmierung*

Bjarne Stroustrup, Autor von C++  
<http://www.stroustrup.com/C++.html>

*C makes it easy to shoot yourself in the foot;  
C++ makes it harder, but when you do  
it blows your whole leg off.*

Bjarne Stroustrup, Autor von C++, ca. 1986  
[http://www.stroustrup.com/bs\\_faq.html](http://www.stroustrup.com/bs_faq.html)  
[#really-say-that](#)

### Motivation:

Vermeidung unsicherer Techniken,  
insbesondere von Präprozessor-Konstruktionen und Zeigern,  
unter Beibehaltung der Effizienz

## 6.2 Elementare Neuerungen in C++ gegenüber C

## 6.2 Elementare Neuerungen in C++ gegenüber C

- Kommentare mit //



## 6.2 Elementare Neuerungen in C++ gegenüber C

- Kommentare mit //
- Konstante:  
**const int** n = 5;  
**int** prime[n] = { 2, 3, 5, 7, 11 };

## 6.2 Elementare Neuerungen in C++ gegenüber C

- Kommentare mit //
- Konstante:  
`const int n = 5;`  
`int prime[n] = { 2, 3, 5, 7, 11 };`
- Ab C++11: `constexpr`-Funktionen  
C++11: darf nur aus einem einzigen `return`-Statement bestehen  
→ `?:` statt `if`, Rekursion statt Schleife  
C++14: auch Verzweigungen und Schleifen erlaubt

## 6.2 Elementare Neuerungen in C++ gegenüber C

- Kommentare mit //
- Konstante:  
`const int n = 5;`  
`int prime[n] = { 2, 3, 5, 7, 11 };`
- Ab C++11: `constexpr`-Funktionen  
C++11: darf nur aus einem einzigen `return`-Statement bestehen  
→ `?:` statt `if`, Rekursion statt Schleife  
C++14: auch Verzweigungen und Schleifen erlaubt
- leere Parameterliste: `void` optional  
in C: ohne `void` = Parameterliste wird nicht geprüft

## 6.2 Elementare Neuerungen in C++ gegenüber C

- Kommentare mit //
- Konstante:  
**const int** n = 5;  
**int** prime[n] = { 2, 3, 5, 7, 11 };
- Ab C++11: **constexpr**-Funktionen  
C++11: darf nur aus einem einzigen **return**-Statement bestehen  
→ **?:** statt **if**, Rekursion statt Schleife  
C++14: auch Verzweigungen und Schleifen erlaubt
- leere Parameterliste: **void** optional  
in C: ohne **void** = Parameterliste wird nicht geprüft
- Operatoren **new** und **delete**  
als Alternative zu den Funktionen **malloc()** und **free()**

## 6.3 Referenz-Typen

```
void calc_answer (int &answer)
{
    answer = 42;
}
```

... als Alternative zu ...

```
void calc_answer (int *answer)
{
    *answer = 42;
}
```

- Zeiger „verborgen“, übersichtlicher und sicherer
- Es gibt keinen **NULL**-Wert.  
→ Für verkettete Listen u. ä.: Tricks erforderlich

## 6.4 Überladbare Operatoren und Funktionen

```
#include <iostream>
```

```
int main ()  
{  
    std::cout << "Hello, world!" << std::endl;  
    return 0;  
}
```

## 6.4 Überladbare Operatoren und Funktionen

```
#include <iostream>
```

```
int main ()  
{  
    std::cout << "Hello, world!" << std::endl;  
    return 0;  
}
```

Bemerkungen:

- Compilieren mit `g++` statt `gcc`:  
C++-Bibliotheken mit einbinden
- Der Operator `<<` hat normalerweise keinen Seiteneffekt, hier schon.

## 6.4 Überladbare Operatoren und Funktionen

```
#include <iostream>
```

```
struct vector  
{  
    double x, y, z;  
};
```

```
vector operator + (vector u, vector v)  
{  
    vector w = { u.x + v.x, u.y + v.y, u.z + v.z };  
    return w;  
}
```

- ++ wird zum Präfix-Operator.
- ++ mit zusätzlichem (ungenutzten) **int**-Parameter wird zum Postfix-Operator.



## 6.4 Überladbare Operatoren und Funktionen

```
void print (const char *s)
```

```
{  
    printf ("%s", s);  
}
```

```
void print (int i)
```

```
{  
    printf ("%d", i);  
}
```

## 6.4 Überladbare Operatoren und Funktionen

```
void print (const char *s)
```

```
{  
    printf ("%s", s);  
}
```

```
void print (int i)
```

```
{  
    printf ("%d", i);  
}
```

Optionale Parameter:

```
void print (const char *s = "\n")
```

```
{  
    printf ("%s", s);  
}
```

## 6.4 Überladbare Operatoren und Funktionen

```
void print (const char *s)
```

```
{  
    printf ("%s", s);  
}
```

```
void print (int i)
```

```
{  
    printf ("%d", i);  
}
```

Für den Linker:

veränderte, eindeutige Namen

Optionale Parameter:

```
void print (const char *s = "\n")
```

```
{  
    printf ("%s", s);  
}
```

wird vom Compiler erledigt

## 6.4 Überladbare Operatoren und Funktionen

```
void print (const char *s)
```

```
{  
    printf ("%s", s);  
}
```

```
void print (int i)
```

```
{  
    printf ("%d", i);  
}
```

Für den Linker:  
veränderte, eindeutige Namen

Wenn man das nicht will:  
extern "C" { ... }

Optionale Parameter:

```
void print (const char *s = "\n")
```

```
{  
    printf ("%s", s);  
}
```

wird vom Compiler erledigt

## 6.5 Namensräume

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()  
{  
    cout << "Hello,_world!" << endl;  
    return 0;  
}
```

## 6.5 Namensräume

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()  
{  
    cout << "Hello,_world!" << endl;  
    return 0;  
}
```

```
namespace my_output  
{  
    ...  
}
```

```
using namespace my_output;
```

## 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