

Algorithmen und Datenstrukturen in C/C++

Prof. Dr. rer. nat. Peter Gerwinski

19. April 2018

Algorithmen und Datenstrukturen in C/C++

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

1 Einführung

2 ...

Rasterung

$O(n \log n)$

array

struct

CORDIC

Wegfindung

FFT

B-Baum

Verschlüsseln

pointer

verkettete Liste

Datenbanken

Datenkompression

digitale Signatur

Hash-Tabelle

Prüfsumme

kryptographische Hash-Funktion

Algorithmen und Datenstrukturen in C/C++

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

1 Einführung

2 Einführung in C++

3 Datenorganisation

- Listen, Bäume, Hash-Tabellen, ...

4 Datenkodierung

- Fehlererkennung und -korrektur
- Kompression
- Kryptographie

5 Hardwarenahe Algorithmen

- FFT, CORDIC, ...

6 Optimierung

- Wegfindung, ...

7 Numerik



Änderungen
vorbehalten

2 Einführung in C++

2.0 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.

2 Einführung in C++

2.0 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“

- kein „Fallschirm“
- kompakte Schreibweise

C makes it easy to shoot yourself in the foot.

Bjarne Stroustrup, ca. 1986

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

Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln
- kein „Fallschirm“

2 Einführung in C++

2.1 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
[#really-say-that](#)

Motivation:

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

2.2 Elementare Neuerungen gegenüber C

- Kommentare mit //
- Konstante:
const int n = 5;
int prime[n] = { 2, 3, 5, 7, 11 };
- Ab C++11: **constexpr**-Funktionen
darf nur aus einem einzigen **return**-Statement bestehen
→ keine Schleife, aber Rekursion 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()**

2.2 Elementare Neuerungen gegenüber C

- Kommentare mit //
- Konstante:
const int n = 5;
int prime[n] = { 2, 3, 5, 7, 11 };
- Ab C++11: **constexpr**-Funktionen
... **anscheinend auch ohne „constexpr“** ...
darf nur aus einem einzigen **return**-Statement bestehen
→ keine Schleife – **oder vielleicht doch außer ab C++14** –, aber
Rekursion 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()**

2.2 Elementare Neuerungen gegenüber C

- Kommentare mit //
- Konstante:
const int n = 5;
int prime[n] = { 2, 3, 5, 7, 11 };
- Ab C++11: **constexpr**-Funktionen
... **anscheinend auch ohne „constexpr“** ...
darf nur aus einem einzigen **return**-Statement bestehen
→ **C++-11**: keine Schleife, aber Rekursion erlaubt
→ **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()**

2.3 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

2.4 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.

2.4 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.

2.4 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

2.5 Namensräume

```
#include <iostream>
```

```
using namespace std;
```

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

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

```
using namespace my_output;
```

2.6 Objekte

```
struct TBase  
{  
};
```

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

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

2.6 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

2.6 Objekte: Konstruktoren und Destruktoren

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

Algorithmen und Datenstrukturen in C/C++

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

1 Einführung

2 Einführung in C++

...

2.3 Referenz-Typen

2.4 Überladbare Operatoren und Funktionen

2.5 Namensräume

2.6 Objekte

2.7 Strings

2.8 Templates

2.9 Exceptions

3 Datenorganisation

4 Datenkodierung

5 Hardwarenahe Algorithmen

6 Optimierung

7 Numerik



Änderungen
vorbehalten