

Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

26. November 2018

Vertiefung Software-Entwicklung in C++

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

1 Einführung

2 Wiederholung: Programmieren in C

...

2.17 Algorithmen

2.18 Bit-Operationen

2.19 Dynamische Speicherverwaltung

2.20 Objektorientierte Programmierung

3 Einführung in C++

3.1 Motivation

3.2 Elementare Neuerungen gegenüber C

3.3 Referenz-Typen

3.4 Überladbare Operatoren und Funktionen

3.5 Namensräume

3.6 Objekte

...

...



Änderungen
vorbehalten

2.17 Algorithmen

2.17.1 Differentialgleichungen

$$\varphi'(t) = \omega(t)$$

$$\omega'(t) = -\frac{g}{l} \cdot \sin \varphi(t)$$

- Von Hand (analytisch): Lösung raten (Ansatz), Parameter berechnen
- Mit Computer (numerisch): Eulersches Polygonzugverfahren

```
phi += dt * omega;  
omega += - dt * g / l * sin (phi);
```

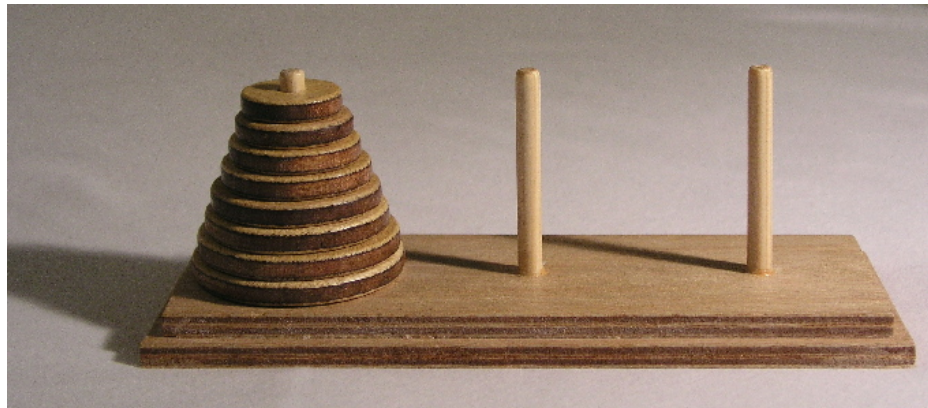
Praktikumsaufgabe im 5. Semester Bachelor: Basketball

2.17 Algorithmen

2.17.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi



2.17 Algorithmen

2.17.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

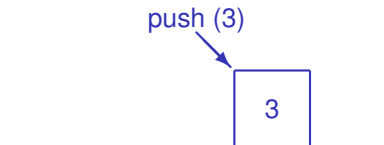
- 64 Scheiben, 3 Plätze,
immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben
auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar:
schiebe $n - 1$ Scheiben auf Hilfsplatz,
verschiebe die darunterliegende,
hole $n - 1$ Scheiben von Hilfsplatz

```
void verschiebe (int n, int start, int ziel)
{
    if (n == 1)
        verschiebe_1_scheibe (start, ziel);
    else
    {
        verschiebe (1, start, hilfsplatz);
        verschiebe (n - 1, start, ziel);
        verschiebe (1, hilfsplatz, ziel);
    }
}
```

2.17 Algorithmen

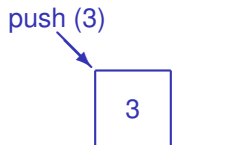
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

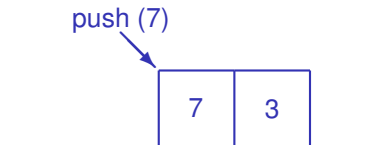


LIFO = Stack = Stapel

2.17 Algorithmen

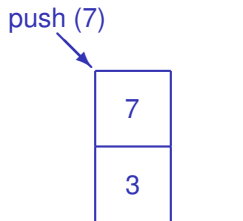
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

2.17 Algorithmen

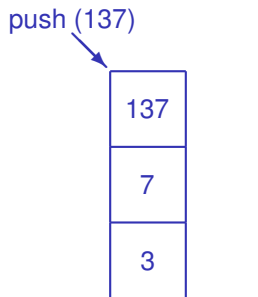
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

2.17 Algorithmen

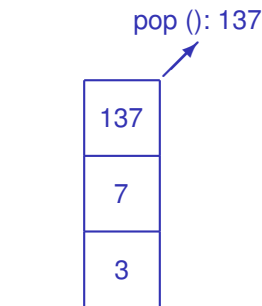
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

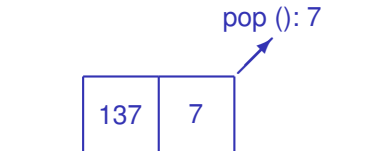


LIFO = Stack = Stapel

2.17 Algorithmen

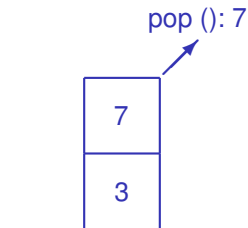
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

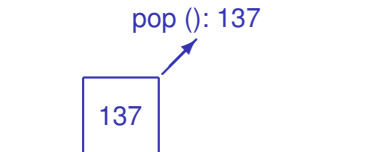


LIFO = Stack = Stapel

2.17 Algorithmen

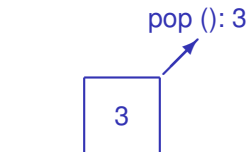
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



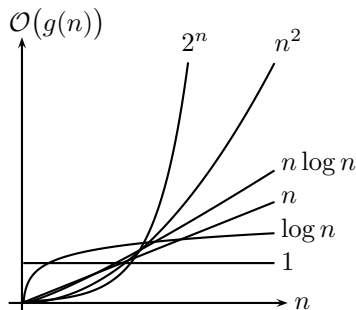
LIFO = Stack = Stapel

2.17 Algorithmen

2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Rekursiv sortieren
→ Quicksort: $\mathcal{O}(n \log n)$ bis $\mathcal{O}(n^2)$



n : Eingabedaten

$g(n)$: Rechenzeit

2.18 Bit-Operationen

2.18.1 Zahlensysteme

Basis	Name	Beispiel	Anwendung
2	Binärsystem	1 0000 0011	Bit-Operationen
8	Oktalsystem	0403	Dateizugriffsrechte (Unix)
10	Dezimalsystem	259	Alltag
16	Hexadezimalsystem	0x103	Bit-Operationen
256	(keiner gebräuchlich)	0.0.1.3	IP-Adressen (IPv4)

- Computer rechnen im Binärsystem.
- Für viele Anwendungen (z. B. I/O-Ports, Grafik, ...) ist es notwendig, Bits in Zahlen einzeln ansprechen zu können.

2.18 Bit-Operationen

2.18.1 Zahlensysteme

000	0	0000	0	1000	8
001	1	0001	1	1001	9
010	2	0010	2	1010	A
011	3	0011	3	1011	B
100	4	0100	4	1100	C
101	5	0101	5	1101	D
110	6	0110	6	1110	E
111	7	0111	7	1111	F

- Oktal- und Hexadezimalzahlen lassen sich ziffernweise in Binär-Zahlen umrechnen.
- Hexadezimalzahlen sind eine Kurzschreibweise für Binärzahlen, gruppiert zu jeweils 4 Bits.
- Oktalzahlen sind eine Kurzschreibweise für Binärzahlen, gruppiert zu jeweils 3 Bits.
- Trotz Taschenrechner u. ä. lohnt es sich, die o. a. Umrechnungstabelle **auswendig** zu kennen.

2.18 Bit-Operationen

2.18.2 Bit-Operationen in C

C-Operator	Verknüpfung	Anwendung
<code>&</code>	Und	Bits gezielt löschen
<code> </code>	Oder	Bits gezielt setzen
<code>^</code>	Exklusiv-Oder	Bits gezielt invertieren
<code>~</code>	Nicht	Alle Bits invertieren
<code><<</code>	Verschiebung nach links	Maske generieren
<code>>></code>	Verschiebung nach rechts	Bits isolieren

Numerierung der Bits: von rechts ab 0

Bit Nr. 3 auf 1 setzen: `a |= 1 << 3;`

Bit Nr. 4 auf 0 setzen: `a &= ~(1 << 4);`

Bit Nr. 0 invertieren: `a ^= 1 << 0;`

Abfrage, ob Bit Nr. 1 gesetzt ist: `if (a & (1 << 1))`

2.19 Dynamische Speicherverwaltung

```
char *name[] = { "Anna", "Berthold", "Caesar" };
```

```
...
```

```
name[3] = "Dieter";
```


2.19 Dynamische Speicherverwaltung

```
#include <stdlib.h>
```

```
...
```

```
char **name = malloc (3 * sizeof (char *));  
    /* Speicherplatz für 3 Zeiger anfordern */
```

```
...
```

```
free (name)  
    /* Speicherplatz freigeben */
```

Aufgabe: Schreiben Sie eine C-Bibliothek, die ein dynamisches „Array von Bits“ realisiert, z. B. mittels der folgenden Funktionen:

<code>bit_array *create_bit_array (int size);</code>	Bit-Array dynamisch erzeugen
<code>void free_bit_array (bit_array *b);</code>	Bit-Array wieder freigeben
<code>void set_bit (bit_array *b, int i);</code>	Bei Index i auf 1 setzen
<code>void clear_bit (bit_array *b, int i);</code>	Bei Index i auf 0 setzen
<code>int get_bit (bit_array *b, int i);</code>	Bei Index i lesen

Hinweise:

- Der Datentyp `bit_array` ist – sinnvollerweise in der `.h`-Datei – selbst zu definieren, z. B. als ein `struct`, das einen Zeiger auf die eigentlichen Daten, eine Größenangabe und evtl. noch zusätzliche Daten enthält.
- Die Benutzung des Bit-Arrays soll vollständig durch die o. a. Funktionen („Methoden“) erfolgen. Der Benutzer braucht sich insbesondere nicht damit zu beschäftigen, wie das Bit-Array intern aufgebaut ist.
- Sie benötigen ein Array, z. B. von `char`- oder `int`-Variablen, am besten `uint8_t` oder einen größeren Typ (aus `stdint.h`).
- Sie benötigen eine Division (/) sowie den Divisionsrest (Modulo: %).
- Die Größe des Bit-„Arrays“ wird über den Aufruf der Funktion `create_bit_array()` dynamisch festgelegt.

2.20 Objektorientierte Programmierung

2.20.1 Konzepte und Ziele

- Array: feste Anzahl von Elementen desselben Typs (z. B.: 3 Zeiger)
- Dynamisches Array: variable Anzahl von Elementen desselben Typs
- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern

2.20 Objektorientierte Programmierung

2.20.1 Konzepte und Ziele

- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern
- Zeiger auf verschiedene Strukturen mit einem gemeinsamen Anteil von Datenfeldern
→ „verwandte“ *Objekte*, *Klassen* von Objekten
- Struktur, die *nur* den gemeinsamen Anteil enthält
→ „Vorfahr“, *Basisklasse*, *Vererbung*
- Explizite Typumwandlung eines Zeigers auf die Basisklasse in einen Zeiger auf die *abgeleitete Klasse*
→ Man kann ein Array unterschiedlicher Objekte in einer Schleife abarbeiten.
→ *Polymorphie*

2.20 Objektorientierte Programmierung

2.20.1 Konzepte und Ziele

- *Objekte, Klassen, Basisklassen, abgeleitete Klassen*
- *Vererbung, Polymorphie*
- Funktionen, die mit dem Objekt arbeiten: *Methoden*
- Aufgerufene Funktion hängt vom Typ des Objekts ab: *virtuelle Methode*
- Realisierung über Zeiger, die im Objekt gespeichert sind
(Genaugenommen: Tabelle von Zeigern)

2.20 Objektorientierte Programmierung

2.20.2 Beispiel: Zahlen und Buchstaben

```
typedef struct  
{  
    int type;  
} t_base;
```

```
typedef struct  
{  
    int type;  
    int content;  
} t_integer;
```

```
typedef struct  
{  
    int type;  
    char *content;  
} t_string;
```

```
typedef struct
```

```
{  
    int type;  
} t_base;
```

```
typedef struct
```

```
{  
    int type;  
    int content;  
} t_integer;
```

```
typedef struct
```

```
{  
    int type;  
    char *content;  
} t_string;
```

```
t_integer i = { 1, 42 };
```

```
t_string s = { 2, "Hello,_world!" };
```

```
t_base *object[] = { (t_base *) &i, (t_base *) &s };
```



explizite

Typumwandlung

2.20 Objektorientierte Programmierung

2.20.2 Beispiel: Zahlen und Buchstaben

Weitere Beispiele:

- Editor für graphische Objekte
- Datenbank-Software
- graphische Benutzeroberfläche (GUI)

2.20 Objektorientierte Programmierung

2.20.3 Objektorientierte Programmierung in C

```
typedef struct  
{  
    int type;  
} t_base;
```

```
typedef struct  
{  
    int type;  
    int content;  
} t_integer;
```

```
typedef struct  
{  
    int type;  
    char *content;  
} t_string;
```

2.20 Objektorientierte Programmierung

2.20.3 Objektorientierte Programmierung in C++

```
struct TBase  
{  
};
```

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

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

3 Einführung in C++

3.1 Motivation

- Vermeidung unsicherer Techniken, insbesondere von Präprozessor-Konstruktionen und Zeigern, unter Beibehaltung der Effizienz
- Compiler-Unterstützung für objektorientierte Programmierung

3.2 Elementare Neuerungen gegenüber C

3.2 Elementare Neuerungen gegenüber C

- Kommentare mit //

3.2 Elementare Neuerungen gegenüber C

- Kommentare mit //
- Konstante:
const int answer = 42;

3.2 Elementare Neuerungen gegenüber C

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

3.2 Elementare Neuerungen gegenüber C

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

3.2 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

3.2 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
- Ab C++14: auch „normale“ Funktionen erlaubt

3.2 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
- Ab C++14: auch „normale“ Funktionen erlaubt
- leere Parameterliste: **void** optional

3.2 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
- Ab C++14: auch „normale“ Funktionen erlaubt
- leere Parameterliste: **void** optional
in C: ohne **void** = Parameterliste wird nicht geprüft

3.2 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
- Ab C++14: auch „normale“ Funktionen 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()**

3.3 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

```
#include <iostream>
```

```
int main ()
```

```
{
```

```
    std::cout << "Hello, world!" << std::endl;
```

```
    return 0;
```

```
}
```

3.4 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.

3.4 Überladbare Operatoren und Funktionen

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

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

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

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

3.4 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);  
}
```

3.4 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



3.4 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

3.5 Namensräume

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    cout << "Hello,_world!" << endl;
```

```
    return 0;
```

```
}
```

3.5 Namensräume

```
#include <iostream>
```

```
using namespace std;
```

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

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

```
using namespace my_output;
```

3.6 Objekte

```
struct TBase  
{  
};
```

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

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

Aufgabe

Schreiben Sie eine Klasse, die eine (z. B. einfach verkettete) Liste von Objekten (z. B. Strings) verwaltet.

Anregungen:

- Einfügen eines Strings in die Klasse mit einem Operator, z. B. `+=`
- Geht es mit Referenzen anstelle von Zeigern?
- Operator `[]`, um die Liste wie ein Array ansprechen zu können (wenn auch mit $\mathcal{O}(n)$ statt $\mathcal{O}(1)$)
- Methode (z. B. `foreach()`), um eine Callback-Funktion für alle Listenelemente aufzurufen

3.7 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

3.8 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