

# Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

3. Dezember 2018

# Vertiefung Software-Entwicklung in C++

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

## 1 Einführung

## 2 Wiederholung: Programmieren in C

...

### 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

### 3.7 Objekte: Zugriffsrechte

### 3.8 Objekte: Konstruktoren und Destruktoren

### 3.9 Strings

### 3.10 Templates

...

...



Änderungen  
vorbehalten

## 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

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

## 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);  
}
```

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;  
}
```

```
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

## 3.9 Strings

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

## 3.10 Templates

Anwendung desselben Quelltextes auf verschiedene Datentypen

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

Beide Schreibweisen sind bedeutungsgleich.

## 3.10 Templates

Anwendung desselben Quelltextes auf verschiedene Datentypen

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

Beide Schreibweisen sind bedeutungsgleich.

Vorsicht: Fehler werden erst bei Instantiierung erkannt!

## 3.10 Templates

Anwendung desselben Quelltextes auf verschiedene Datentypen

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

Beide Schreibweisen sind bedeutungsgleich.

Vorsicht: Fehler werden erst bei Instantiierung erkannt!

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



## 3.11 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*