

Einführung in die Programmierung

Wintersemester 2017/18

Prof. Dr. Günter Rudolph
 Lehrstuhl für Algorithm Engineering
 Fakultät für Informatik
 TU Dortmund



Letzte Vorlesung

Inhalt

- Ein Blick zurück: Was haben Sie gelernt?
- Gegenwart: Was wurde bzgl. C++ nicht behandelt?
- Ein Blick nach vorne: Wie könnte es weiter gehen?

Ein Blick zurück: Was haben Sie gelernt?

- | | | |
|--------------------------------|-----------------------------------|--------------------------|
| 1. Einleitung | 10. Vererbung | |
| 2. Darstellung von Information | 11. Virtuelle Methoden | |
| 3. Kontrollstrukturen | 12. Ausnahmebehandlung | |
| 4. Zeiger | 13. Datenstrukturen & Algorithmen | } nicht klausur-relevant |
| 5. Funktionen | 14. STL | |
| 6. Gültigkeitsbereiche | 15. GUI Programmierung | |
| 7. Rekursion | 16. (entfällt) | |
| 8. Klassen | | |
| 9. Elementare Datenstrukturen | | |
- Grammatiken / endl. Automaten Schablonen

Gegenwart: Was wurde bzgl. C++ nicht behandelt?

- | | |
|--------------------------|--|
| 1. Komma Operator | } Jetzt: Übersichtartige Vorstellung, um mal davon gehört zu haben!

nicht klausurrelevant → aber gut zu wissen! |
| 2. Bitweise Operatoren | |
| 3. Bitfelder | |
| 4. Union | |
| 5. Lokale Klassen | |
| 6. Geschachtelte Klassen | |
| 7. Mehrfaches Erben | |
| 8. Virtuelle Vererbung | |
| 9. C++ Casts | |
| 10. C++11 Standard | |
| 11. C++14 Standard | |
| 12. C++17 Standard | |

1. Komma Operator

- Erlaubt Reihung von Ausdrücken, die durch Komma getrennt sind
- Ausführung / Auswertung von links nach rechts
- Wert ist der am weitesten rechts stehende

⇒ ermöglicht Platz sparende (und damit beliebig verwirrende) Schreibweise:

```
// int *ia, ix, sz, index;
int ival = (ia != 0)
      ? ix = get_value(), ia[index] = ix
      : ia = new int[sz], ia[index] = 1;
```

... auch beliebt:

```
for (i = 0, j = n; i < n; i++, j--) { /* ... */ }
```

Gefahr:

```
delete x;
delete y; } OK!
delete x, y; ⇔ delete x;
y;           ⇒ Speicherleck!
```

2. Bitweise Operatoren

~	bitweises NOT	
&	bitweises AND	&=
	bitweises OR	=
^	bitweises XOR	^=
>>	Schieben nach rechts (* 2)	>>=
<<	Schieben nach links (/ 2)	<<=

Bsp:

```
unsigned char x, y z;
x = 1; // 00000001
y = 255; // 11111111
z = x & y; // 00000001
z = x << 3; // 00001000
z |= 3; // 00001011
z >>= 1; // 00000101
x = z ^ y; // 11111010
x = ~x; // 00000101
```

- ⇒ kann zu trickreicher (Platz sparender) Schreibweise / Darstellung führen
- ⇒ erschwert i.A. die Verständlichkeit des Programms
- ⇒ **sparsam einsetzen!**

Gefahr:

Verwechslung & und | mit && und ||

3. Bitfelder

```
class File {
// ...
unsigned short modified : 1; // Bitfeld
};
```

Zugriff auch via Bitoperatoren möglich!

„Hallo Compiler: es wird nur 1 Bit zur Datenhaltung benötigt!“

Aufeinander folgende Bitfelder in Klassendefinition werden vom Compiler gepackt!

```
typedef unsigned short Bits;
class File {
public:
Bits mode : 2; // read / write
Bits modified : 1; // no / yes
Bits protection_owner : 3; // read / write / execute
Bits protection_group : 3; // read / write / execute
Bits protection_world : 3; // read / write / execute
};
```

} UNIX / Linux

4. Union

⇒ spezieller **struct** ⇒ bis auf Schlüsselwort gleiche Syntax wie **struct**

```
union Werte {
char cval;
int ival;
char *sval;
double dval;
};
```

Zugriffsrechte per Default: **public** aber auch **protected**, **private** möglich

Sinn und Zweck? ⇒ Platz sparen!

```
Werte x;
int i = x.ival;
char c = x.cval;
```

Illegale Komponenten:

- statische Variable
- Variable, die Referenz ist
- Variable einer Klasse mit Konstruktor und / oder Destruktor

```
union illegal {
static int is;
int &rs;
Screen s;
};
```

Screen *ps; wäre OK!

5. Lokale Klassen

= Klassendefinitionen in Funktionen

```
void Funktion(int wert) {
    class Lokal {
    public:
        int lokalerWert;
        // ...
    };
    Lokal wert;
    wert.lokalerWert = wert;
    // ...
}
```

Sichtbarkeit:

Lokale Klasse `Lokal` nur sichtbar im Gültigkeitsbereich der Funktion!

Verwendung der lokalen Klasse außerhalb der Funktion nicht möglich, da dort unbekannt!

Warnung:

Kann die Lesbarkeit / Verständlichkeit des Programms erschweren.

6. Geschachtelte Klassen (nested classes)

= Klassendefinitionen in Klassendefinitionen

```
class Node { /* ... */ };
class Tree {
public:
    class Node { /* ... */ };
    Node tree;
    // ...
};
```

Gültigkeitsbereiche:

Geschachtelte Klasse `Node` ist gültig in class `Tree`.

Sie verdeckt hier die Klasse `Node` im umfassenden Gültigkeitsbereich der Klasse `Tree`.

wg. `public` auch Datendefinition außerhalb der Klasse möglich:

```
Tree::Node node;
```

Typischerweise `private` oder `protected` als Hilfsklasse für „internen Gebrauch“. Falls interne Klasse so wichtig, dass auch andere Klassen sie häufig verwenden möchten, dann als eigenständige Klasse definieren!

7. Mehrfaches Erben (hier nur die Idee)

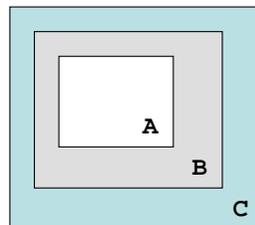
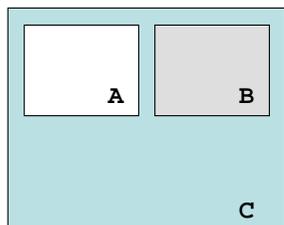
Vererbung = Komposition von Klassen by value



Unterklasse beinhaltet jeweils alle nicht-statischen Attribute (und Methoden) der Oberklasse.

Bsp: Einfaches Erben

Bsp: Mehrfaches Erben



```
class B : public A {};
class C : public B {};
```

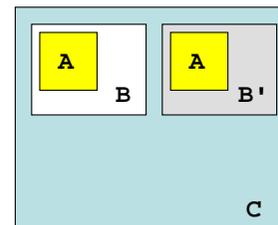
```
class C : public A, public B;
```

Prinzip der Komposition

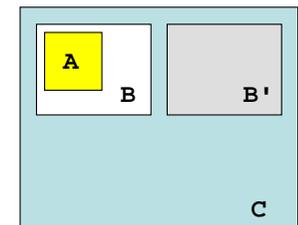
Kommaseparierte Liste von Elternklassen

8. Virtuelle Vererbung (hier nur Idee)

=> beseitigt ein Problem, das (nur) bei mehrfacher Vererbung auftreten kann



In Klasse `C` liegen nun zwei Instanzen von Klasse `A`, da Vererbung = Komposition by value. Es könnte nur eine Instanz von Klasse `A` nötig sein ...



Wie soll das realisiert und in C++ ausgedrückt werden?

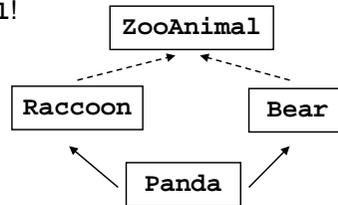
1. Realisiert durch andere Kompositionsart: Komposition by reference d.h. beide Klassen halten nur eine Referenz auf die gleiche Klasse!
2. Ausgedrückt in C++ durch Schlüsselwort `virtual` in Oberklassenliste.

8. Virtuelle Vererbung (hier nur Idee)

Beispiel:

```
class Bear : public ZooAnimal {};
class Raccoon : public ZooAnimal {};
class Panda : public Bear, public Raccoon, public Endangered {}
```

Oops! → Panda hat zwei Instanzen von ZooAnimal!



Lösung: Virtuelle Vererbung!

```
class Bear : virtual public ZooAnimal {};
class Raccoon : virtual public ZooAnimal {};
class Panda : public Bear, public Raccoon, public Endangered {}
```

9. C++ Casts

- Casts sind *explizite* Typumwandlungen
- Explizit? ⇒ Nur wenn wir das wollen (und genau wissen was wir tun)!
- Gibt es in 4 Varianten

static_cast<Typ>(arg)

- Wandelt zwischen verwandten Typen um, z.B. Zahlen
- Ähnlich dem alten C Cast

```
double d = 3.141;

int i1 = (int)d; // Alter C Cast, i1 = 3
int i2 = static_cast<int>(d); // C++ Cast, i2 = 3
int i3 = static_cast<int>(5.973); // C++ Cast, i3 = 5
```

9. C++ Casts

const_cast<Typ>(arg)

- Entfernt die Konstanz von Variablen
- Verwendung kann gefährlich sein!

```
void division3(double& d){
    d = d / 3.0;
}

const double zahl = 6.0;
division3(zahl); // Fehler, zahl ist
                // konstant!
division3(const_cast<double&>(zahl)); // Funktioniert, aber
                // fragwürdig!
```

9. C++ Casts

dynamic_cast<Typ>(arg)

- Castet sicher in Vererbungshierarchien nach *unten*
- Liefert Nullpointer (Zeiger) oder wirft Exception (Referenz), wenn Cast fehlschlägt

```
class Mitarbeiter {
public:
    virtual double gehalt() = 0;
};

class WiMi : public Mitarbeiter{
public:
    double gehalt() { return 20.0; }
};

class SHK : public Mitarbeiter{
public:
    double gehalt() { return 8.0; }
    void gibVielArbeit(){
        cout << "Armer Studi" << endl;
    }
};
```

9. C++ Casts

dynamic_cast<Typ>(arg)

```
Mitarbeiter* m1 = new WiMi();
Mitarbeiter* m2 = new SHK();

SHK* shk = dynamic_cast<SHK*>(m1);      // wird nicht klappen!
if(shk != nullptr) shk->gibVielArbeit();
else cout << "Cast fehlgeschlagen" << endl;

shk = dynamic_cast<SHK*>(m2);           // funktioniert
if(shk != nullptr) shk->gibVielArbeit();
else cout << "Cast fehlgeschlagen" << endl;
```

9. C++ Casts

reinterpret_cast<Typ>(arg)

- kopiert das Bitmuster und ändert nur den Typ
- z.B. zum Umwandeln von (Funktions-) Zeigern (C Relikt)

```
int doSomething(int d){
    cout << "Do something!" << endl;
    return d-42;
}

void (*fptr)(int);      // Pointer auf Funktion
fptr = &doSomething;   // Geht nicht
fptr = reinterpret_cast<void (*)(int)>(&doSomething); // Geht!
fptr(39);              // Aufruf über Function Pointer
```

10. C++11 Standard

bisher: C++ Standard von 2003 (Erweiterung von 1998)

in dieser Vorlesung eingeführt:

- `nullptr`
- Delegation von Konstruktoren
- *deleted* & *defaulted* Funktionen `=delete;` oder `=default;`
- **struct**-artige Initialisierung von Klassen

nicht eingeführt:

- Erweiterungen der STL (**smart pointers!**)
- lambda expressions
- Automatische Typerkennung: `auto`
- **u.v.a.m.**

11. C++14 Standard

... im Wesentlichen *bug fixes* von C++11

und einige Erweiterungen/Verallgemeinerungen:

z.B.

- **return Typ Deduktion:** `auto`

12. C++17 Standard

... viele neue Features: Wichtigkeit / Notwendigkeit nicht eindeutig klar

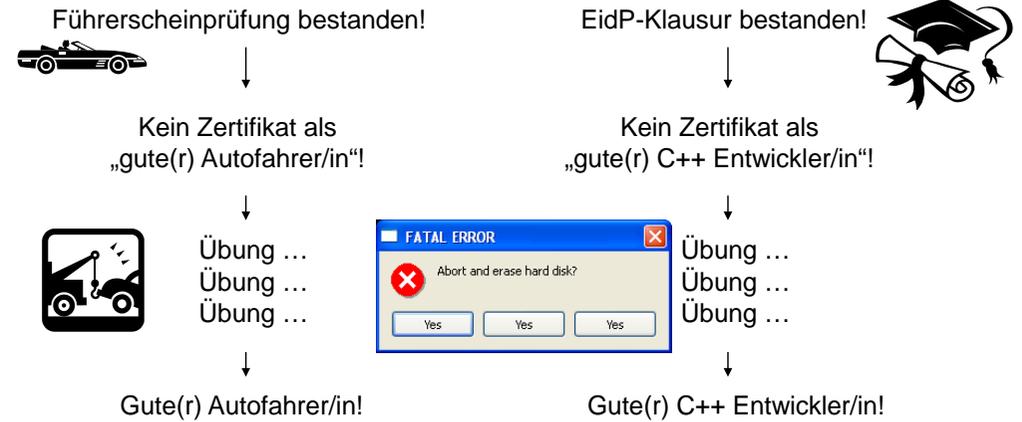
- Structured binding declarations, allowing `auto [a, b] = getTwoReturnValues();`
- neue File Library (Dateibehandlung)
- u.v.a.m.

} "nice to have"

Nächste Version des Standards: **C++20**

Wie könnte es weiter gehen?

Analogie: Führerschein



Wie könnte es weiter gehen?

