Teil 7
Objektorientierung

- Klassen
- Const-Methoden
- Operator-Overloading
- Kopierkonstruktor / Zuweisungsoperator
- Member-Initialisierung
- implizite und explizite Konstruktoren
- Vererbung / Polymorphismus
- virtuelle Methoden
- Vorausdeklarationen
- R-Value-Referenzen

Klassen

```
class Scalar
{
public:
  Scalar(double value, string unit);
  string to_string();
private:
  double value;
  string unit;
};

int main()
{
  Scalar distance(42.195, "km");
  distance.unit = "m";
  cout << distance.to_string() << endl;
  return 0;
}
```

nicht erlaubt, weil privat

Struct vs. Class

- `struct` und `class` sind abgesehen von der Default-Sichtbarkeit identisch
- trotzdem werden beide üblicherweise unterschiedlich verwendet

- Struct
  - Standard-Sichtbarkeit: *public*
  - andere Sichtbarkeit eher nicht empfohlen
  - verwendet für Objekte *ohne* Invarianz
  - für „einfache“ Datenobjekte
  - eher keine Vererbung

- Class
  - Standard-Sichtbarkeit: *private*
  - explizite Angabe der Sichtbarkeit üblich
  - verwendet für Objekte *mit* Invarianz
  - für komplexere Objekte mit Funktionalität
  - Vererbung üblich

Const-Methoden

```
class Scalar
{
public:
  Scalar(double value, string unit);
  string to_string() const;
  void setValue(double value);
private:
  double value;
  string unit;
};

int main()
{
  const Scalar distance(42.195, "km");
  distance.setValue(21.0975);
  cout << distance.to_string() << endl;
  return 0;
}
```

nicht erlaubt, weil Methode nicht const

nur erlaubt, weil Methode const

Getter / Setter

```
class Scalar
{
public:
  Scalar(double value, string unit);

  void setValue(double value)
  {
    this->value = value;
  }

  double getValue() const
  {
    return value;
  }

private:
  double value;
  string unit;
};
```

Setter ohne const

Getter mit const

Aufgaben

Aufgabe 19A

- der Typ „Point“ soll zur Klasse werden

Aufgabe 19B

- eine Klasse „Time“ soll einen Zeitpunkt repräsentieren
- es braucht verschiedene Methoden, um sinnvoll auf Stunden, Minuten und Sekunden zugreifen zu können

Operator-Overloading

```
#include <string>

class Scalar
{
public:
  Scalar operator+(const Scalar& other);
};

Scalar Scalar::operator+(const Scalar& other)
{
  Scalar result;
  result.value = value + other.value;
  result.unit = unit;
  return result;
}

Scalar a, b;
Scalar c = a + b;
```

Operator-Overloading

```
int& IntArray::operator[](const int index) const
{
  return values[index];
}

IntArray array;
cout << array[24] << endl;
```

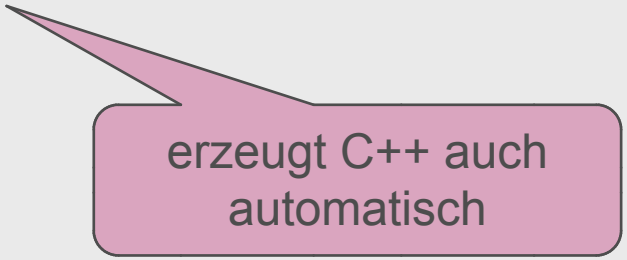
```
IntArray& IntArray::operator<<(const int value)
{
  append(value);
  return *this;
}

IntArray array;
array << 1 << 2 << 3;
```

Kopierkonstruktor

```
class Scalar
{
public:
  Scalar(const Scalar& other);
private:
  double value;
  string unit;
};

Scalar::Scalar(const Scalar& other) :
  value(other.value),
  unit(other.unit)
{
}
```



Zuweisungsoperator

```
class Scalar
{
public:
  Scalar& operator=(const Scalar& other);
private:
  double value;
  string unit;
};

Scalar& Scalar::operator=(const Scalar& other)
{
  if (this != &other)
  {
    value = other.value;
    unit = other.unit;
  }
  return *this;
}
```

erzeugt C++ ebenfalls automatisch

Konstruktor / Zuweisung

```
Scalar a(42.195, "km"); // Konstruktor
Scalar b(a); // Kopierkonstruktor
Scalar c = b; // Kopierkonstruktor
c = a; // Zuweisungsoperator
```

- C++ erzeugt impliziten Kopierkonstruktor und Zuweisungsoperator automatisch, wenn möglich, d.h. wenn alle Felder und Basisklassen einen Kopierkonstruktor bzw. Zuweisungsoperator haben

Konstruktor / Zuweisung

C++ 11

```
class Scalar
{
public:
  Scalar(const Scalar& other) = default;
  Scalar(const Scalar& other) = delete;
  Scalar& operator=(const Scalar& other) = default;
  Scalar& operator=(const Scalar& other) = delete;
};
```

- Seit C++11 lassen sich Kopierkonstruktor und Zuweisungsoperator explizit automatisch erzeugen lassen oder löschen
- Vor C++11 konnte man diese lediglich verstecken, indem man sie „private“ deklarierte und leer implementierte.

Member-Initialisierung

```
class Scalar
{
public:
  Scalar(const Scalar& other) = default;
private:
  double value = 0.0;
  string unit;
};
```

- defaultmäßig sind einfache Typen (int, double, bool usw.) uninitialisiert, d.h. zufällig
- komplexe Typen werden defaultmäßig via Konstruktor initialisiert (nicht unbedingt deren Member)
- ab C++11 lassen sich Defaultwerte hinterlegen

implizite Konstruktoren

```
class Scalar
{
public:
  Scalar(double value) : value(value), unit("m") {}
private:
  double value;
  string unit;
};

void print(const Scalar& scalar)
{
  cout << scalar.to_string() << endl;
}

print(10.97);
```

- wenn ein einparametriger impliziter Konstruktor existiert, wird dieser zur impliziten Typkonvertierung verwendet

explizite Konstruktoren

```
class Scalar
{
public:
  explicit Scalar(int value) : value(value), unit("m") {}
private:
  double value;
  string unit;
};

print(Scalar(10.97));
```

- implizite Konvertierung ist schwer nachvollziehbar, insbesondere wegen Verkettung
- Konstruktoren mit Parametern sollten normalerweise als „explicit“ markiert werden

Aufgaben

Aufgabe 20A

- die Konstruktoren der vergangenen Übungen sollen als „explicit“ markiert werden

Aufgabe 20B

- zuvor folgendes ausprobieren (alternativ mit den Typen Point oder Time):

```
void print(const Scalar& scalar)
{
  cout << scalar.to_string()
       << endl;
}

print({10.97, "m"});
```

C++ 11

Vererbung

```
class Time : public Scalar
{
public:
  explicit Time(double h, double min, double sec);
};

Time::Time(double h, double min, double sec) :
  Scalar(h * 3600.0 + min * 60.0 + sec, "s")
{
}
```

- `public` nicht vergessen! (ist leider nicht Default)
- Vererbung mit Sichtbarkeit `private` oder `protected` ist eher unüblich
- heutzutage wird eher die Schnittstelle als die Implementierung vererbt

Vererbung

```
class Time : public Scalar
{
public:
  Time(double h, double min, double sec, string timeZone);
  string getTimeZone() const;
private:
  string timeZone;
};

Time::Time(double h, double min, double sec, string timeZone) :
  Scalar(h * 3600.0 + min * 60.0 + sec, "s"),
  timeZone(timeZone)
{
}
```

- Member-Variablen von Klassen heißen auch „Felder“ oder „Attribute“

Polymorphismus

```
class Operation
{
public:
  virtual ~Operation() {}
  virtual double execute(double a, double b) = 0;
};
```

„Interface“

```
class Multiplication : public Operation
{
public:
  double execute(double a, double b) override;
};

double Multiplication::execute(double a, double b)
{
  return a * b;
}
```

C++ 11

Polymorphismus

```
class Addition : public Operation
{
public:
  double execute(double a, double b) override;
};

double Addition::execute(double a, double b)
{
  return a + b;
}
```

```
int main()
{
  bool multiply = true;
  Operation* op = multiply ? new Multiplication : new Addition;
  double result = op->execute(4, 6);
  delete op;
  return 0;
}
```

virtuelle Methode

virtueller Destruktor

Aufgaben

Aufgabe 21A

- verschiedene Klassen sollen Tierarten repräsentieren (z.B. Hund, Katze, Ente)
- in einer virtuellen Methode soll der typische Laut des Tier ausgegeben werden
- eine Funktion, der ein Tier-Objekt übergeben wird (als Referenz oder Pointer), soll diese Methode aufrufen
- die Funktion soll mit verschiedenen Tier-Objekten getestet werden

Aufgabe 21B

- verschiedene Klassen sollen geometrische Formen repräsentieren (z.B. Quadrat, Kreis, Rechteck)
- in einer virtuellen Methode soll der Flächeninhalt oder Umfang berechnet werden
- eine Funktion, der solch ein Objekt übergeben wird (als Referenz oder Pointer), soll die Methode aufrufen und das Ergebnis ausgeben
- die Funktion soll mit verschiedenen Objekten getestet werden

Virtuelle Methoden

```
class Operation
{
public:
  virtual ~Operation() {}
  virtual double execute(double a, double b) = 0;
};
```

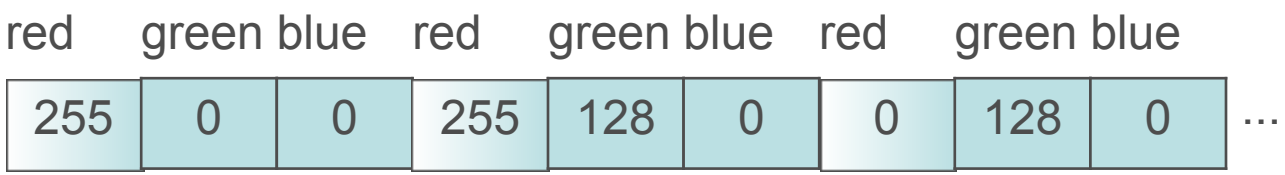
- Klassen, von denen abgeleitet wird, sollten einen *virtuellen Destruktor* haben
- Objekte von Klassen mit mindestens einer virtuellen Methode haben zusätzlich einen Pointer zur „VMT“ (auch „vtable“) und sind damit normalerweise 4 oder 8 Byte größer

Virtuelle Methoden

```
struct Color
{
  unsigned char red;
  unsigned char green;
  unsigned char blue;
  double getBrightness() const;
};

double Color::getBrightness() const
{
  return double(red + green + blue) / 3.0;
}

Color pixels[1000];
```

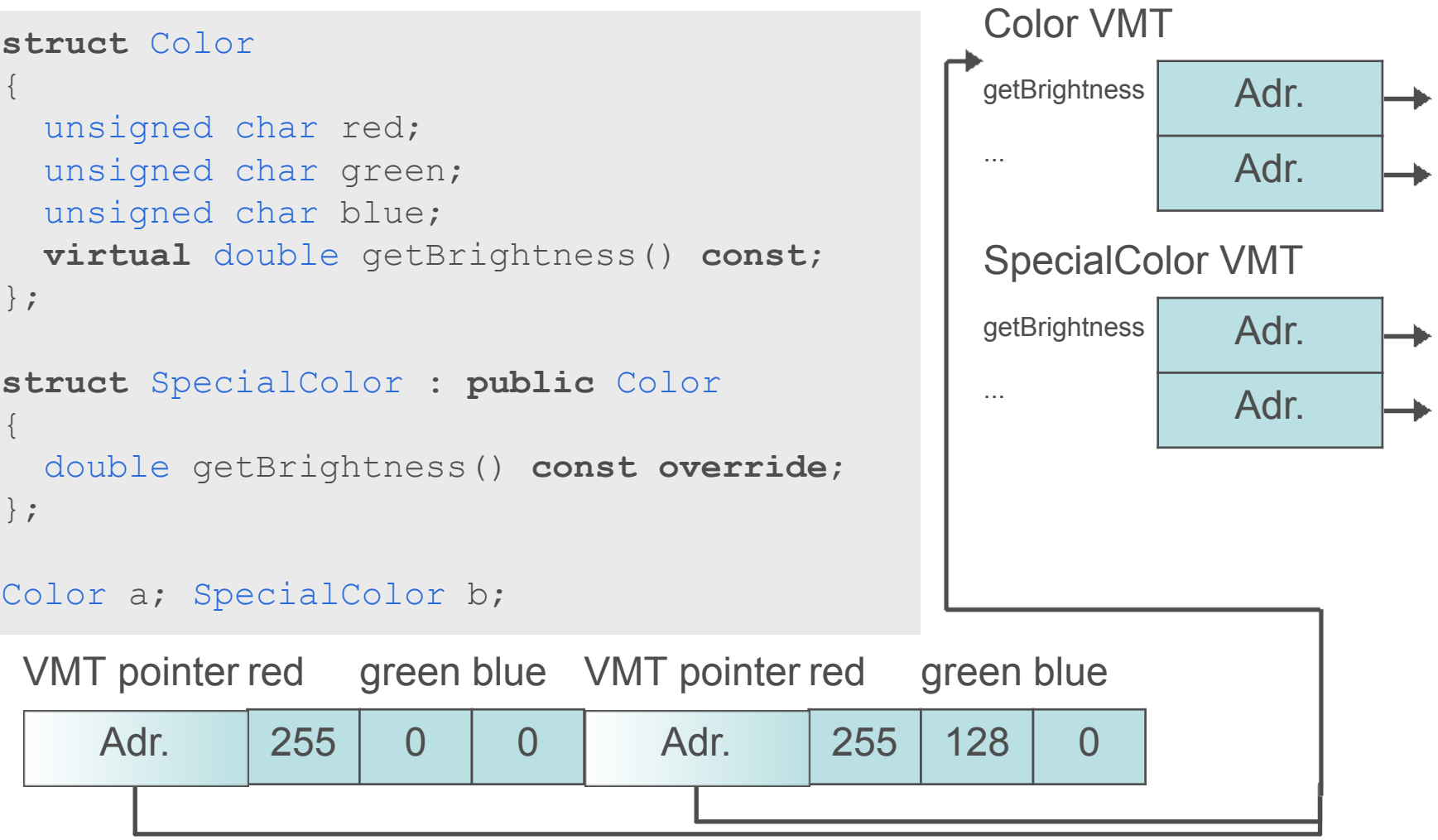


Virtuelle Methoden

```
struct Color
{
  unsigned char red;
  unsigned char green;
  unsigned char blue;
  virtual double getBrightness() const;
};

struct SpecialColor : public Color
{
  double getBrightness() const override;
};

Color a; SpecialColor b;
```



Vorausdeklarationen

```
#include "scalar.h"

class Printer
{
public:
  void print(const Scalar& scalar);
};
```

unnötig für Header

```
#include "printer.h"


void Printer::print(const Scalar& scalar)
{
  std::cout << scalar.to_string() << std::endl;
}
```

Vorausdeklarationen

```
class Scalar;

class Printer
{
public:
  void print(const Scalar& scalar);
};
```

ausreichend für Referenz oder Pointer

```
#include "printer.h"
#include "scalar.h"

void Printer::print(const Scalar& scalar)
{
  std::cout << scalar.to_string() << std::endl;
}
```

nötig für Verwendung

R-Values

```
x = 7;
y = x + 2;
z = x;
pi = calcPi();
```

```
7 = x;
x + 2 = y;
x = z;
calcPi() = pi;
```

?

- R-Values können nur rechts vom Zuweisungsoperator stehen
- L-Values können links und rechts stehen
- R-Values haben keinen Namen und sind temporär

R-Value-Referenzen

C++ 11

```
string text = getErrorMessage(404);
const string& text = getErrorMessage(404);
string& text = getErrorMessage(404);
string&& text = getErrorMessage(404);
```

nicht erlaubt

- Referenzen auf temporäre Objekte müssen zwingend „const“ sein
- R-Value-Referenzen müssen nicht „const“ sein

R-Value-Referenzen

C++ 11

```
class Scalar
{
public:
  Scalar(const double value, const string& unit);
  Scalar(const Scalar& other);
  Scalar(Scalar&& other);
private:
  double value;
  string unit;
};

Scalar first(42.195, "km");
Scalar second(first);
Scalar third(getScalar());
```

erlaubt auch R-Values

bevorzugt R-Values

Übergabe als R-Value

- Übergabe als R-Value-Referenz erlaubt besondere Optimierungen (da man ja weiß, dass danach nicht mehr auf das Objekt zugreifen werden kann)