Teil 6
Objekte

- Structs
- Methoden
- Inlining
- Operator-Overloading
- Objektzeiger
- Konstruktoren & Destruktoren
- Unions
- Namensräume
- Globale Variablen
- Defines & Makros

Benutzerdefinierte Typen

```
struct Scalar
{
  double value;
  string unit;
};

int main()
{
  Scalar distance;
  distance.value = 42.195;
  distance.unit = "km";
  return 0;
}
```

abschließendes Semikolon !!!

- Zusammenfassung von Werten, die zusammengehören
- übernommen aus C

Benutzerdefinierte Typen

```
struct Scalar
{
  double value;
  string unit;
};

Scalar distance = {42.195, "km"};
```

```
struct Scalar
{
  double value;
  string unit;
} distance = {42.195, "km"};
```

Definition + Deklaration
eher nicht empfehlenswert

```
Scalar distance = {};
distance.unit = "km";
```

alle Felder mit Nullwerten
initialisieren

Methoden

```
#include <string>

struct Scalar
{
  double value;
  string unit;
  string to_string();
};

string Scalar::to_string()
{
  return std::to_string(value) + " " + unit;
}

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

Typname bildet Namensraum

Zugriff auf „Member-Variablen“ (Fields)

Inlining

```
#include <string>

struct Scalar
{
  double value;
  string unit;
  string to_string()
  {
    return std::to_string(value) + " " + unit;
  }
};
```

- Beim Inlining findet kein Funktionsaufruf statt, sondern der Funktionscode wird jeweils eingebaut
- sinnvoll bei kurzen Funktionen
- der Compiler ist nicht darauf festgelegt

Inlining

```
#include <string>

struct Scalar
{
  double value;
  string unit;
  string to_string();
};

inline string Scalar::to_string()
{
  return std::to_string(value) + " " + unit;
}
```

- Definition muss ebenfalls im Header sein
- Syntax ist auch für normale Funktionen verfügbar

Operator-Overloading

```
#include <string>

struct Scalar
{
  double value;
  string unit;
};

Scalar operator+(const Scalar& first, const Scalar& second)
{
  Scalar result;
  result.value = first.value + second.value;
  result.unit = first.unit;
  return result;
}

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

Aufgaben

Aufgabe 17A

- ein Typ „Point“ soll die Koordinaten x, y und z aufnehmen
- zwei Punkte sollen addiert werden können, wie man es bei Vektoren gewöhnt ist (Implementierung als Operator, Test in Main-Funktion)

Aufgabe 17B

- ein Typ „Point“ soll die Koordinaten x, y und z aufnehmen
- der Typ „Point“ soll einen Operator für die Multiplikation mit einem Faktor (double) bekommen
- außerdem soll es einen Operator für das Skalarprodukt ($a_x b_x + a_y b_y + a_z b_z$) geben
- Kann man das Kreuzprodukt ($a_y b_z - a_z b_y$, $a_z b_x - a_x b_z$, $a_x b_y - a_y b_x$) auf dieselbe Weise hinzufügen?

Objektzeiger

```
Scalar distance;
distance.value = 42.195;
distance.unit = "km";
cout << distance.to_string() << endl;
```

```
Scalar* distance = new Scalar;
(*distance).value = 42.195;
(*distance).unit = "km";
cout << (*distance).to_string() << endl;
delete distance;
```

```
Scalar* distance = new Scalar;
distance->value = 42.195;
distance->unit = "km";
cout << distance->to_string() << endl;
delete distance;
```

This-Pointer

```
string Scalar::to_string()
{
  return std::to_string(value) + " " + unit;
}
```

```
string to_string(Scalar* this)
{
  return std::to_string(this->value) + " " + this->unit;
}
```

```
string Scalar::to_string()
{
  return std::to_string(this->value) + " " + this->unit;
}
```

- Compiler erzeugt impliziten Parameter „This-Pointer“

Konstruktoren

```
struct Scalar
{
  double value;
  string unit;
  Scalar(double value, string unit);
  string to_string();
};

Scalar::Scalar(double value, string unit) :
  value(value), unit(unit)
{
  // zusätzlicher Initialisierungscode
}

int main()
{
  Scalar distance(42.195, "km");
  cout << distance.to_string() << endl;
```

Namensgleicheit Parameter / Member kein Problem

Konstruktoren

```
struct Scalar
{
  double value;
  string unit;
  Scalar(double value, string unit);
  string to_string();
};

Scalar::Scalar(double value, string unit) :
  value{value}, unit{unit}
{
  // zusätzlicher Initialisierungscode
}

int main()
{
  Scalar distance(42.195, "km");
  cout << distance.to_string() << endl;
```

C++ 11

Aufgaben

Aufgabe 18A

- der Typ „Point“ braucht einen geeigneten Konstruktor

Aufgabe 18B

- Sind mehrere Konstruktoren für den Typ „Point“ sinnvoll?

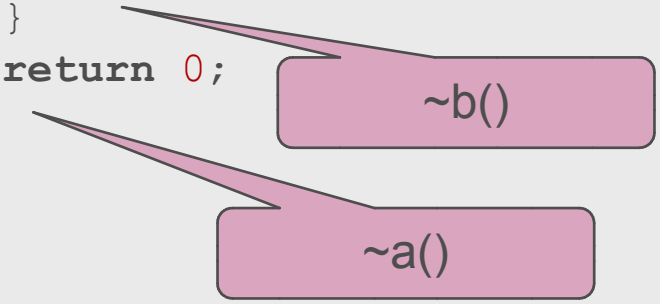
Destruktoren

```
struct IntArray
{
  int* values;
  IntArray(int count);
  ~IntArray();
};

IntArray::IntArray(int count)
{
  values = new int[count];
}

IntArray::~IntArray()
{
  delete [] values;
}
```

```
int main()
{
  IntArray a(10);
  a.values[0] = 123;
  {
    IntArray b(100);
    b.values[0] = a.values[0];
  }
  return 0;
}
```



Unions

```
union Color
{
  struct
  {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
    unsigned char alpha;
  }
  unsigned int value;
  unsigned char channel[4];
};

Color color;
color.value = 0xFF00FFFF;
cout << color.green;
cout << color.channel[0];
```

- unterschiedliche Varianten zum Zugriff auf dieselben Daten
- Alternative für `reinterpret_cast`
- Mittel für dynamisch typisierte „variant“ Objekte

Namensräume

```
namespace Math
{
  struct Scalar
  {
    double value;
    string unit;
  };
}

int main()
{
  Math::Scalar duration(24, "h");
}
```

- Innerhalb des Namensraums entfällt die Namespace-Angabe
- Namespaces können verschachtelt sein

Globale Variablen

- Regel Nr. 1: Vermeiden

```
void a(const Settings& settings)
{
  ...
}

void b(const Settings& settings)
{
  ...
}

int main()
{
  Settings settings;
  settings.load("settings.xml");
  a(settings);
  b(settings);
}
```

Globale Variablen

```
// globals.cpp
#include "globals.h"
Settings globalSettings("settings.xml");
```

```
// globals.h
#ifndef GLOBALS_H
#define GLOBALS_H
extern Settings globalSettings;
#endif
```

```
#include "globals.h"

int main()
{
  globalSettings.load(filename);
  a();
  b();
}
```

Globale Variablen

```
struct Scalar
{
  double value;
  string unit;
  static string defaultUnit;
};
```

```
string Scalar::defaultUnit("m");
```

```
int main()
{
  cout << Scalar::defaultUnit << endl;
}
```

Defines

```
#define PI 3.14159265358979323846

double y = sin(alpha * PI / 180.0);
```

```
double y = sin(alpha * 3.14159265358979323846 / 180.0);
```

- Relikt von C
- vom Präprozessor vor dem Compile-Vorgang ersetzt
- normalerweise sind globale Konstanten vorzuziehen
- auch für Makros (parametrisierbar)

Makros

```
#define MAX(a, b) a > b ? a : b

int peak = MAX(x, 99) + 1;
```

```
int peak = x > 99 ? x : 99 + 1;
```

- Makros bergen Fallen
- normalerweise benötigt es Klammern um jeden Parameter und um das Statement selbst
- fast immer sind Funktionen geschickter (Inlining)

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int peak = MAX(x, 99) + 1;
```

```
int peak = ((x) > (99) ? (x) : (99)) + 1;
```