Klassenkomponenten

Gültigkeitsbereich Der syntaktische Gültigkeitsbereich der Namen von Klassenkomponenten erstreckt sich zunächst über die Vereinbarung der gesamten Klasse, insb. als auch vor dem Ort an dem die Komponente selbst vereinbart wird. Zusätzlich schließt der syntaktische Gültigkeitsbereich alle Definitionen anderer Klassenkomponenten innerhalb der selben Übersetzungseinheit ein, auch wenn diese außerhalb der Klassenvereinbarung stehen.

Außerhalb des syntaktischen Gültigkeitsbereichs eines Komponentennamens k der Klasse C, jedoch innerhalb des syntaktischen Gültigkeitsbereichs von C, steht C`::`k zur Verfügung um die Klassenkomponente anzusprechen. Insbesondere Definitionen von Klassenkomponenten außerhalb der Deklaration der Klasse müssen daher mit den derart *qualifizierten* Namen erfolgen.

Der syntaktische Gültigkeitsbereich des Namens der Klasse selbst folgt den Regeln einer gewöhnlichen Deklaration, erstreckt sich also vom Punkt der Vereinbarung bis zum Ende der Übersetzungseinheit.

Separat definierte Klassenkomponenten

```
class Complex {
  private:
    double x, y;

  public:
    Complex(double Re = 0, double Im = 0);

    double real();
    double imag();

    friend Complex conj(Complex);
};

Complex::Complex(double Re, double Im) : x(Re), y(Im) {}

double Complex::real() { return x; }
double Complex::imag() { return y; }

Complex conj(Complex z) {
  z.y = -z.y;

  return z;
}
```

Zugriffsattribute Zusätzlich zum syntaktischen Gültigkeitsbereich der Namen der Klassenkomponenten gibt es auch Zugriffsbeschränkungen anhand des Zugriffsattributs, insb. `private` und `public`. Komponenten, die bei Vereinbarung mit `private` markiert wurden, stehen nur innerhalb anderer Klassenkomponenten und befreundeten Funktionen und Klassen (`friend`) zur Verfügung. Als `public` deklarierte Klassenkomponenten stehen überall zur Verfügung.

Für mit dem Schlüsselwort `class` deklarierte Klassen ist `private` die Voreinstellung, wenn nichts Anderes angegeben wird.

`this`

In Komponentenfunktionen der Klasse *C* steht der Name `this` als Zeiger vom Typ *C*`*` auf das aktuelle Objekt zur Verfügung.

Innerhalb der Komponenten einer Klasse können die Namen *n* anderer Klassenkomponenten aufgefasst werden als verkürzte Notation äquivalent zu `this->`*n* bzw. `(*this).`*n*.

Beispiel. Im folgenden Programm wird eine Methode `square` definiert, die bei Aufruf die komplexe Zahl quadriert und, unter Verwendung von `this`, eine Kopie des derart angepassten Objektes auch als Rückgabewert zurück gibt.

this_demo.cpp

```cpp
#include <iostream>

using namespace std;

class Complex {
  private:
    double x, y;

  public:
    Complex(double Re = 0, double Im = 0) : x(Re), y(Im) {}

    double real() { return x; }
    double imag() { return y; }

    Complex square() {
      double x2 = x*x - y*y, y2 = 2*x*y;
      x = x2; y = y2;
      return *this;
    }

    friend ostream& operator<<(ostream& out, Complex z) {
      return out << "(" << z.real() << "," << z.imag() << ")";
    }
};

int main() {
  Complex z{1, 1};

  cout << "z davor:     " << z << endl;
  cout << "z quadriert: " << z.square() << endl;
  cout << "z danach:    " << z << endl;

  return 0;
}
```

```
z davor:     (1,1)
z quadriert: (0,2)
z danach:    (0,2)
```

Konstante Methoden

Methoden können als `const` markiert werden. In diesem Fall hat für eine Klasse C `this` den Typ `const` $C*$ innerhalb der Methode. Dies hat zur Folge, dass der Compiler sicherstellt, dass die Methode keine Änderungen an den Attributen des Objekts vornimmt.

Ist ein Objekt einer Klasse `const`, so können auch nur konstante Methoden auf das Objekt aufgerufen werden.

Beispiel. Wir definieren eine rudimentäre Klasse zur Darstellung komplexer Zahlen mit zwei konstanten Methoden für Real- bzw. Imaginärteil.

const_method_demo.cpp

```cpp
#include <iostream>

using namespace std;

class Complex {
  private:
    double x, y;

  public:
    Complex(double Re = 0, double Im = 0) : x(Re), y(Im) {}

    double real() const { return x; }
    double imag() const { return y; }
};

int main() {
  Complex z1{1.0, 0.0};
  const Complex z2{2.0, 3.0};

  cout << "Re z1 = " << z1.real() << "  Im z1 = " << z1.imag() << endl;
  cout << "Re z2 = " << z2.real() << "  Im z2 = " << z2.imag() << endl;

  return 0;
}
```

```
Re z1 = 1  Im z1 = 0
Re z2 = 2  Im z2 = 3
```

Konstruktoren

Konstruktoren verhalten sich in ihrer Vereinbarung syntaktisch ähnlich als wären sie Komponentenfunktionen mit Namen identisch zu dem der Klasse. Im Gegensatz zu (Komponenten-)Funktionen kennen Konstruktoren *keinerlei* Ergebeistyp; nicht einmal `void`. Es darf daher auch kein Ergebnistyp als Teil der Vereinbarung angegeben werden.

Standardkonstruktoren Falls innerhalb einer Klasse kein Konstruktor vereinbart ist, wird der Standardkonstruktor mit Zugriffsattribut `public` automatisch erzeugt. Hierbei werden alle Attribute des Objekts ebenfalls mit ihrem Standardkonstruktor initialisiert bzw. bei eingebauten Typen garnicht initialisiert.

Konstruktorinitialisierungsliste In Konstruktordefinitionen kann angegeben werden, wie die Attribute der Klasse initialisert werden sollen, noch bevor der Code des Konstruktors ausgeführt wird. Die Initialisierung erfolgt in der Reihenfolge der Vereinbarungen der Komponenten innerhalb der Klasse. Nicht in der Konstruktorinitialisierungsliste aufgeführte Attribute werden initialisiert wie durch den Standardkonstruktor.

⟨Kons'init'liste⟩ → “:” ⟨Init'⟩ {“,” ⟨Init'⟩}
⟨Init'⟩ → ⟨Name⟩ “(” [⟨Ausdruck⟩ {“,” ⟨Ausdruck⟩}] “)”

Beispiel. Im Folgenden werden mehrere Klassen mit verschiedenen Arten von Konstruktoren definiert:

Konstruktoren

```
class Complex {
  private:
    double x, y;

  public:
    Complex(double Re = 0, double Im = 0) : x(Re), y(Im) {}

    friend ostream& operator<<(ostream& out, Complex z) {
      return out << "(" << z.x << "," << z.y << ")";
    }
};

class Polynom {
  private:
    vector<double> a; // Koeffizienten

  public:
    Polynom(int n): a(n+1) { a[n] = 1; } // Monom x^n

    int grad() const { return a.size() - 1; }
};

class Vektor {
  private:
    double* ap; // Zeiger auf/C-Array von doubles
    int    len; // Länge

  public:
    Vektor(): ap(nullptr), len(0) {} // Leerer Vektor
    Vektor(int n, double x = 0): len(n) { // n Kopien von x
      ap = new double[n];
      for (int i = 0; i < n; i++) ap[i] = x;
    }

    friend ostream& operator<<(ostream& out, Vektor v) {
      for (int i = 0; i < v.len; i++) {
        if (i > 0) out << ", ";
```

```
        out << v.ap[i];
      }
      return out;
    }
};
```

constructors_demo.cpp

```
#include <iostream>
#include <vector>

using namespace std;

Konstruktoren

int main() {
  Polynom p{2};
  cout << p.grad() << endl;

  Complex c{1.0, 2.0};
  cout << c << endl;

  Vektor v{3, 1.0};
  cout << v << endl;
}
```

```
2
(1,2)
1, 1, 1
```

Kopierkonstruktoren Der Kopierkonstruktor einer Klasse *C* dient dazu neue Objekte der Klasse zu initialisieren indem eine Kopie gemacht wird eines anderen Objektes der selben Klasse. Es handelt sich hierbei um einen Konstruktor mit einzigem Referenzparameter `C&` bzw. `const C&` insb. also *nicht* *C* bzw. `const C`.

Falls nicht explizit vereinbart wird der Kopierkonstruktor automatisch mit Zugriffsattribut `public` erzeugt. Hierbei werden alle Attribute wiederum durch Aufruf ihrer Kopierkonstruktoren (oder, bei eingebauten Datentypen, durch direkte Zuweisung) kopiert.

Der Kopierkonstruktor kommt bei der Übergabe eines Objekts als Parameter an eine Funktion und bei der Rückgabe von Objekten durch Funktionen zum Einsatz. Er wird *nicht* für Zuweisungen verwendet (vgl. Zuweisungsoperatoren).

Beispiel. Wir vereinbaren eine rudimentäre Klasse für Vektoren von `doubles` inkl. Kopierkonstruktor. Wir definieren eine Funktion `f`, die ein Vektor-Objekt als Parameter akzeptiert. Beim Aufruf von `f` aus `main` wird eine Kopie eines Vektor-Objekts erzeugt. Hierfür wird der von uns definierte Kopierkonstruktor verwendet. Dass es sich wirklich um eine Kopie handelt demonstrieren wir dadurch, dass wir innerhalb von `f` den Inhalt des Objekts anpassen und zum Vergleich sowohl danach, wie auch im Hauptprogramm ausgeben.

vector_copy_demo.cpp

```cpp
#include <iostream>

using namespace std;

class Vektor {
  private:
    double* ap;
    int    len;

  public:
    Vektor(): ap(nullptr), len(0) {} // Leerer Vektor
    Vektor(int n, double x = 0): len(n) { // n Kopien von x
      ap = new double[n];
      for (int i = 0; i < n; i++) ap[i] = x;
    }

    Vektor(Vektor& v): len(v.len) {
      ap = new double[len];
      for (int i = 0; i < len; i++) ap[i] = v.ap[i];
    }

    friend ostream& operator<<(ostream& out, Vektor v) {
      for (int i = 0; i < v.len; i++) {
        if (i > 0) out << ", ";
        out << v.ap[i];
      }
      return out;
    }
    double& operator[](int i) {
      return ap[i];
    }
};

void f(Vektor v) { // Vektor als Parameter um Kopie zu provozieren
  v[1] = 2;
  cout << v << endl;
}

int main() {
  Vektor v{3, 1};

  f(v);

  cout << v << endl;

  return 0;
}
```

```
1, 2, 1
1, 1, 1
```

Destruktoren

Für eine Klasse *C* trägt der zugehörige Destruktor den Namen ~*C*. Er wird analog zu Konstruktoren innerhalb der Klasse und ohne jeden Rückgabetyp vereinbart.

Beispiel. Wir definieren einen Destruktor für eine rudimentäre Klasse für Vektoren von `doubles` (wie oben).

Destruktor

```
~Vektor() {
  delete[] ap;
}
```

Aufgabe des Destruktors ist es etwaigen Speicher der mit dem Objekt assoziiert ist (z.B. Attribute für die im Konstruktor mit `new` Speicher zugewiesen wurde) freizugeben. Der Destruktor wird automatisch aufgerufen, wenn die Lebensdauer des Objektes endet. Bei Variablen also wenn der syntaktische Gültigkeitsbereich der Variable (der Block, der die Deklaration enthält) verlassen wird. Bei temporären Objekten, die im Rahmen eines Ausdrucks konstruiert wurden, direkt nachdem der enthaltende Ausdruck ausgewertet wurde.

Ein zweiter Aufruf des Destruktors, egal ob dieser explizit oder automatisch erfolgt, ist nicht zulässig.

Der Destruktor einer Klasse kann mithilfe des `delete`-Operators explizit aufgerufen werden. I.d.R. ist dies nur für Objekte nötig für die vorher vermöge `new` Speicher explizit zugewiesen wurde.

Falls kein explizit Destruktor definiert wurde, wird automatisch ein leerer Destruktor erzeugt.

Wird der Destruktor aufgerufen (egal ob explizit definiert oder automatisch erzeugt), wird nach Ende der Ausführung des Codes des Destruktors automatisch der jeweilige Destruktor für jedes Attribut aufgerufen. Dies geschieht in umgekehrter Reihenfolge der Deklaration der Attribute innerhalb der Klasse.

Beispiel. Wir definieren eine rudimentäre Klasse für Vektoren von `doubles` ohne Destruktor (wie oben). Wir demonstrieren, dass hierdurch ein *Speicherleck* auftreten kann – es wird Speicher zugewiesen jedoch vor Ende des Programms nie freigegeben. Programme mit Speicherlecks verursachen Probleme dadurch, dass sie bei langer Laufzeit i.d.R. unbegrenzt viel Speicher zugewiesen und somit alle dem System zur Verfügung stehenden Ressourcen verbrauchen.

vector_leak_demo.cpp

```
#include <iostream>

using namespace std;

class Vektor {
  private:
    double* ap; // Zeiger auf/C-Array von doubles
    int     len; // Länge

  public:
    Vektor(): ap(nullptr), len(0) {} // Leerer Vektor
    Vektor(int n, double x = 0): len(n) { // n Kopien von x
      ap = new double[n];
      for (int i = 0; i < n; i++) ap[i] = x;
```

```
    }

    /* Auskommentiert
    Destruktor
    */

    double sum() {
      double s = 0;
      for (int i = 0; i < len; i++) s += ap[i];
      return s;
    }
};

int main() {
  Vektor a(1);
  cout << a.sum() << endl;
}
```

```
==…== Memcheck, a memory error detector
==…== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==…== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==…== Command: vector_leak_demo
==…==
==…==
==…== HEAP SUMMARY:
==…==     in use at exit: 8 bytes in 1 blocks
==…==   total heap usage: 3 allocs, 2 frees, 73,736 bytes allocated
==…==
==…== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==…==    at 0x48440F3: operator new[](unsigned long) (in …)
==…==    by 0x4012B1: Vektor::Vektor(int, double) (vector_leak_demo.cpp:13)
==…==    by 0x4011B9: main (vector_leak_demo.cpp:31)
==…==
==…== LEAK SUMMARY:
==…==    definitely lost: 8 bytes in 1 blocks
==…==    indirectly lost: 0 bytes in 0 blocks
==…==      possibly lost: 0 bytes in 0 blocks
==…==    still reachable: 0 bytes in 0 blocks
==…==         suppressed: 0 bytes in 0 blocks
==…==
==…== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Zuweisungsoperatoren

Der Zuweisungsoperator = kann nur durch eine Methode überladen werden (nicht durch eine befreundete Funktion). Der Operator für eine Klasse *C* muss genau einen Parameter vom Typ *C*, *C*& oder `const` *C*& akzeptieren.

Falls kein Zuweisungsoperator explizit überladen wird, wird automatisch einer erzeugt, der Zuweisung der Attribute komponentenweise vornimmt. Hierfür wird ggf. wiederum der Zuweisungsoperator der Attribute aufgerufen.

Zuweisungen dürfen nicht mit Aufrufen des Kopierkonstruktors verwechselt werden, obwohl Initialiserungen vermöge des Kopierkonstruktors ebenfalls ein Gleichheitszeichen = enthalten.

Beispiel. Diverse Anweisungen, manche mit Zuweisungen oder Kopierkonstruktoraufrufen:

`Complex a{4.0, 5.0}`	Initialisierung vermöge Konstruktor
`Complex b{a}`	Initialisierung vermöge Kopierkonstruktor
`Complex c = a`	Initialisierung vermöge Kopierkonstruktor
`Complex d = Complex{1.0, 2.0}`	Initialisierung vermöge Kopierkonstruktor
`Complex e, f`	Initialisierung vermöge Standardkonstruktor
`e = c`	Zuweisung mit Zuweisungsoperator
`f = Complex{3.0, 5.0}`	Zuweisung mit Zuweisungsoperator

Wesentlicher Unterschied zwischen Kopierkonstruktor und Zuweisungsoperator ist, dass der Zuweisungsoperator auf bereits initialisierte Objekte angewandt wird. Beim Kopierkonstruktor ist nur der Parameter bereits initialisiert.

Beispiel. Wir definieren einen Zuweisungsoperator für eine rudimentäre Klasse für Vektoren von `doubles` (vgl. oben)

Zuweisungsoperator für Vektor

```
// Rückgabewert als Referenz um Kopie zu vermeiden
Vektor& operator=(const Vektor& v) {
  delete[] ap; // this ist bereits initialisiert; Speicher muss daher zunächst freigegeben
  ↪  werden

  len = v.len;
  ap = new double[len];
  for (int i = 0; i < len; i++) ap[i] = v.ap[i];

  return *this;
}
```

Mit dieser Definition von Zuweisungsoperator und dem Kopierkonstruktor, wie oben, hat die Klasse `Vektor` *Wertesemantik*. D.h., dass bei Zuweisungen und nicht-Referenz Parameterübergaben stets Kopien angelegt werden.

Dies entspricht auch dem Verhalten der STL-Behälterklassen wie z.B. `vector<T>`.

Wünscheswert ist dies um unbeabsichtigte Änderungen an den Werten durch z.B. temporäre Änderungen innerhalb aufgerufener Funktionen weitgehend auszuschließen.

Statische Klassenkomponenten

Klassenkomponenten können als `static` deklariert werden.

Statische Datenkomponenten Bei statischen Datenkomponenten muss die *Definition* außerhalb erfolgen und eindeutig sein.

Statische Datenkomponenten einer Klasse sind nicht, wie Attribute, direkt mit den Objekten der Klasse verknüpft. Die Klasse fungiert in diesem Sinne eher als *Namensraum* (`namespace`) um Werte semantisch mit ihr zu assoziieren. Statische Datenkomponenten sind insofern nicht in Objekten der Klasse enthalten und dort auch nicht mit dem Punkt- `.` oder Pfeiloperator `->` ansprechbar. Auch nicht in `this` innerhalb einer Klassenkomponente.

Beispiel. Wir vereinbaren eine Klasse zur Modellierung von Kreisen mit einer statischen Datenkomponente für die Kreiszahl π.

circle_demo.cpp

```
#include <iostream>
#include <cmath>

using namespace std;

class Kreis {
  public:
    static const double pi;

  private:
    double r;

  public:
    Kreis(double r_ = 1): r(r_) {}
    double radius() { return r; }
    double umfang() { return 2*r*pi; }
    friend double flaeche(Kreis k) { return pi*k.r*k.r; }
};

const double Kreis::pi = acos(-1.0);

int main() {
  Kreis k{1.0};
  cout << "Radius:  " << k.radius() << endl
       << "Umfang:  " << k.umfang() << endl
       << "Flaeche: " << flaeche(k) << endl;

  return 0;
}
```

```
Radius:  1
Umfang:  6.28319
Flaeche: 3.14159
```

Statische Komponentenfunktionen Bei statischen Komponentenfunktionen darf, im Gegensatz zu statischen Datenkomponenten, die Definition auch innerhalb der Vereinbarung der Klasse erfolgen.

Statische Komponentenfunktionen verhalten sich i.W. gleich zu befreundeten Funktionen, nur können sie außerhalb der Klasse nicht mit Ihrem Namen f aufgerufen werden, sondern als $C::f$. Auch hier fungiert die Klasse also als Namensraum.

Beispiel. Wir vereinbaren die Klasse zur Modellierung von Kreisen erneut, vereinbaren diesmal jedoch die

Funktion `flaeche` als statisch statt befreundet.

Zudem vereinbaren wir `pi` als `constexpr`. Die Vereinbarung als `constexpr` erlaubt es bei statischen Klassenkomponenten die Definition innerhalb der Klasse vorzunehmen.

Generell führt die Vereinbarung als `constexpr` dazu, dass die Auswertung bereits beim Kompiliervorgang geschieht und das Resultat als Konstante in die Binärdatei eingebunden wird. Dies vermeidet prinzipiell unnötige Auswertungen zur Laufzeit und führt dadurch zu schneller Programmen.

circle_static_demo.cpp

```cpp
#include <iostream>
#include <cmath>

using namespace std;

class Kreis {
  public:
    static constexpr double pi = acos(-1.0);

  private:
    double r;

  public:
    Kreis(double r_ = 1): r(r_) {}
    double radius() { return r; }
    double umfang() { return 2*r*pi; }
    static double flaeche(Kreis k) { return pi*k.r*k.r; }
};

int main() {
  Kreis k{0.5};
  cout << "Radius:  " << k.radius() << endl
       << "Umfang:  " << k.umfang() << endl
       << "Flaeche: " << Kreis::flaeche(k) << endl;

  return 0;
}
```

```
Radius:  0.5
Umfang:  3.14159
Flaeche: 0.785398
```