Klassenkomponenten

Gültigkeitsbereich und Zugriffsattribute

Betrachtet werden nur Klassen, die außerhalb von Funktionen vereinbart sind.

Der Gültigkeitsbereich eines *Komponentennamens* erstreckt sich ab dem Vereinbarungspunkt bis zum Ende der Klassenvereinbarung. Zusätzlich umfaßt er alle Funktionsrümpfe, Konstruktorinitialisierungen und Parametervoreinstellungen der Komponentenfunktionen dieser Klasse.

Außerhalb seines Gültigkeitsbereichs (und innerhalb des Gültigkeitsbereichs des *Klassennamens*) kann der Komponentenname *name* der Klasse C über C`::`*name* angesprochen werden. (Für Klassenobjekte c stehen natürlich der Auswahloperator `.` und, sofern nicht andersweitig überladen, `->` zur Verfügung, d.h. *c.name* bzw. `(&`*c*`)->`*name*).

Der Gültigkeitsbereich des *Klassennamens* außerhalb der Klasse entspricht den Regeln für entsprechende Variablenvereinbarungen und reicht vom Vereinbarungspunkt bis zum Ende der Übersetzungseinheit.

Komponentenfunktionen können innerhalb der Klasse deklariert und außerhalb definiert werden. Dasselbe gilt für befreundete Funktionen. (Befreundete Funktionen gehören allerdings nicht zur Klasse, ihre Vereinbarung innerhalb der Klasse wirkt so, als ob sie unmittelbar nach der Klasse vereinbart wären.)

Auf diese Weise ist es möglich, Definition und Deklaration einer Klasse zu trennen.

Bsp.: Klassendeklaration mit nachfolgender Klassendefinition

```
class Complex {
  private:
    double re,im;

  public:
    Complex(double=0,double=0);           // Konstruktordeklaration

    double real();                        // Komponentenfunktionsdeklaration
    double imag();                        // Komponentenfunktionsdeklaration

    friend Complex conj(Complex);         // Dekl. einer befreundeten Funktion
};


Complex::Complex(double Re,double Im)     // Konstruktordefinition
  : re(Re), im(Im) {}

double Complex::real() { return re; }     // Komponentenfunktionsdefinition
double Complex::imag() { return im; }     // Komponentenfunktionsdefinition

Complex conj(Complex z)                   // Def. der befreundeten Funktion
{
  z.im = -z.im; return z;
}
```

Ob der angesprochene Komponentenname auch verwendet werden kann, hängt von den Zugriffsattributen ab:
Das Attribut `private` gestattet die Verwendung der so gekennzeichneten Komponentennamen nur den Komponentenfunktionen und den mit `friend` deklarierten Funktionen und Klassen.
Das Attribut `public` erlaubt überall die Verwendung der Komponentennamen.

Für `struct` ist `public` und für `class` ist `private` voreingestellt.

Zusätzlich zu den Attributen `private` und `public` gibt es noch das Attribut `protected`, das später im Zusammenhang mit der Vererbung behandelt wird.

Unabhängig von der Vereinbarungsreihenfolge innerhalb der Klasse kann jede Komponentenfunktion andere Klassenkomponenten und befreundete Funktionen verwenden. (Letzeres nur, wenn die befreundete Funktion einen Parameter vom Klassentyp hat.) Der Zugriff befreundeter Funktionen auf die Klassenkomponenten ist ebenfalls reihenfolgeunabhängig.

Komponentenfunktionen und `this`

In Komponentenfunktionen steht `this` für einen Zeiger auf die Klassenvariable (Klassenobjekt), für die die Komponentenfunktion aufgerufen wird.

Bsp.: Komponentenfunktion, die $z \in \mathbb{C}$ quadriert, entspr. verändert und als Ergebnis liefert

```
class Complex {
  private:
    double re,im;
  public:
            :
    Complex quadriere() {
      double re2=re*re-im*im, im2=2*re*im;
      re = re2; im = im2;     // Objekt aendern
      return *this;           // (Veraendertes) Objekt als Ergebnis
    }
            :
};

int main()
{
  Complex z;
  cout << "z: "; cin  >> z;
  cout << "z zuvor:     " << z             << endl;
  cout << "z quadriert: " << z.quadriere() << endl;
  cout << "z danach:    " << z             << endl;
  return 0;
}
```

Ausgabe:

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

Bemerkung: An Stelle des Komponentennamens *name* könnte innerhalb der Komponentenfunktionen `*this.`*name* bzw. `this->`*name* verwendet werden. Entsprechendes gilt für die Komponentenfunktionen.

Konstante Komponentenfunktionen

Konstante Komponentenfunktionen werden durch das Attribut `const` gekennzeichnet, das direkt auf die Parameterliste folgt. Sie dürfen im aufgerufenen Objekt *keine* Komponenten verändern. Die Auswahl konstanter Komponentenfunktionen erfolgt wie bei nichtkonstanten Funktionen, nur das der hypothetische erste Parameter vom Typ `const` T`&` ist.

In *konstanten* Komponentenfunktionen ist `this` ein Zeiger auf eine Konstante (`const` C `*this`). Konstante Komponentenfunktionen können im Unterschied zu nichtkonstanten Komponentenfunktionen auch für Konstanten der Klasse C aufgerufen werden.

Bsp.: Konstante Komponentenfunktionen für Real- und Imaginärteil

```
class Complex {
  private:
    double re,im;

  public:
    Complex (double Re=0, double Im=0): re(Re), im(Im) { } // Konstruktor

    double real () const { return re; }   // konstante Komponentenfunktion
    double imag () const { return im; }   // konstante Komponentenfunktion
};


int main()
{
  Complex z(1.0,2.0);
  const Complex c(2.0,3.0);

  cout << "Re z = " << z.real() << "   Im z = " << z.imag() << endl;
  cout << "Re c = " << c.real() << "   Im c = " << c.imag() << endl;

  return 0;
}
```

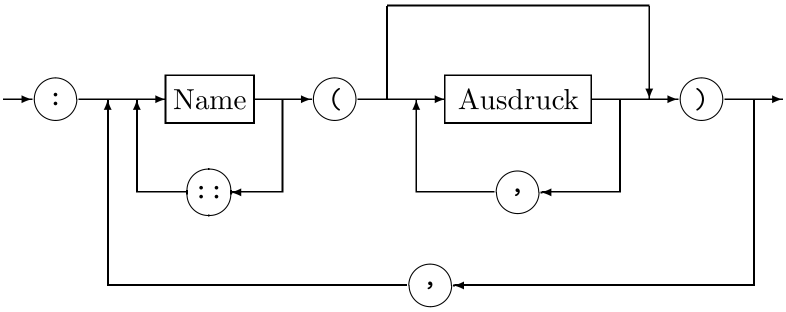
Konstruktoren

Konstruktoren haben den gleichen Namen wie ihre Klasse und zählen zu den Komponentenfunktionen. Für sie darf *kein* Ergebnistyp, auch nicht `void`, angegeben werden.
Ihr Aufruf (über den Klassennamen) dient der Initialisierung der Klassenobjekte.

Konstruktoren dürfen *nicht* als `const` vereinbart werden, sie können aber dennoch konstante Klassenobjekte initialisieren.

Zwei Arten von Konstruktoren sind besonders wichtig und werden u.U. automatisch erzeugt: Standardkonstruktor (default constructor) und Kopierkonstruktor (copy constructor).

Konstruktorinitialisierungsliste



In Konstruktordefinitionen können so Komponenten durch die angegebenen Ausdrücke initialisiert werden, für Referenzen und Konstanten ist das in C++98 die einzige Möglichkeit. Die Initialisierung erfolgt in der Reihenfolge der Komponenten in der Klassenvereinbarung. Nicht in der Konstruktorinitialisierungsliste aufgeführte Komponenten werden mit ihrem Standardkonstruktor (Klassen) initialisiert, jedoch nicht Komponenten eingebauter Datentypen.

Standardkonstruktoren

Der Standardkonstruktor (default constructor) ist diejenige Komponentenfunktion mit Klassennamen, die mit leerer Argumentliste aufgerufen werden kann.
Falls in der Klasse kein Konstruktor vereinbart ist, wird der Standardkonstruktor automatisch mit Zugriffsattribut `public` erzeugt.

Bsp.: Verschiedene Konstruktoren und Standardkonstruktoren
```
class Complex {
  private:
    double re,im;
  public:
    Complex(double Re=0, double Im=0) : re(Re), im(Im) {}
};

class Polynom {
  private:
    vector<double> a;
  public:
    Polynom() {}                                        // Nullpolynom (Laenge 0)
    Polynom(int n): a(n+1) { a[n]=1; }                  // Monom x^n
}

class Vektor {
  private:
    double *ap;
    int    len;
  public:
    Vektor() : ap(0),len(0) {}                          // Vektor der Laenge 0
    Vektor(int n, double x=0): len(n) {                 // Vektor der Laenge n
      ap = new double [n];
      for (int i=0; i<n; ++i) ap[i]=x;
    }
};
```