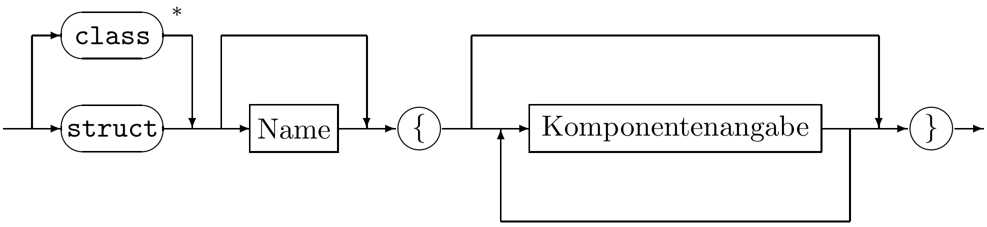
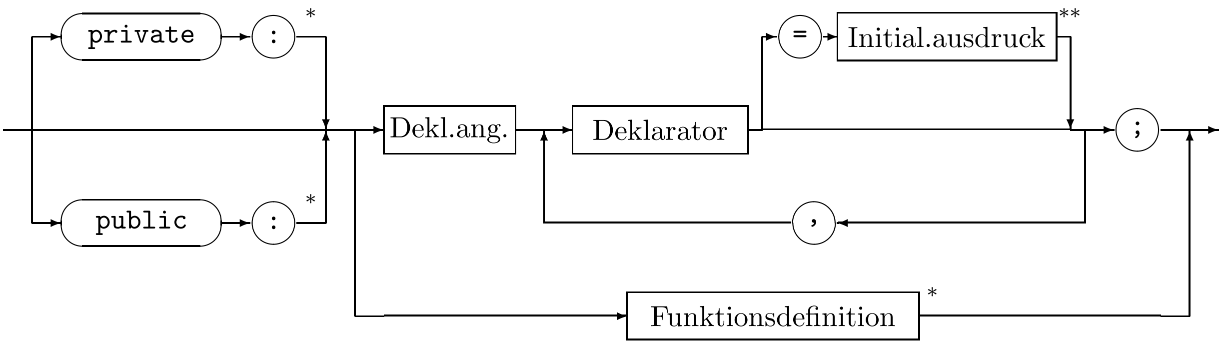
Einfache Klassen [Auszug Programmieren I]

Klassenvereinbarung[1]



Komponentenangabe[1]



Klassen sind zusammengesetzte Datentypen, die von einem Benutzer oder in einer externen Bibliothek definiert sind und deren Komponenten über ihren Namen ansprechbar sind. Neben Datenkomponenten (Attribute) gehören auch Komponentenfunktionen (Methoden, andere Bez.: Elementfunktionen, Memberfunktionen) zu einer Klasse. Zusätzlich können in einer Klasse auch *befreundete* Funktionen vereinbart sein, sie zählen aber nicht zu den Komponentenfunktionen. Zugriffsattribute regeln, welche Funktionen auf die Klassenkomponenten zugreifen können.

Operationen:

`class` *C* {...};	Klassendefinition (außerhalb von Fkt.)
C c	Variablenvereinbarung (nach Voreinst. initial.)
C c($init_1$, $init_2$,...)	Variablenvereinbarung (initialisiert)
C c{ $init_1$, $init_2$,...}**	Variablenvereinbarung (initialisiert)
C c={ $init_1$, $init_2$,...}**	Variablenvereinbarung (Initialisierungsliste)
C()	Temporärobjekt in Ausdrücken (nach Voreinst. initial.)
C($init_1$, $init_2$,...)	Temporärobjekt in Ausdrücken (initialisiert)
c.*name*	Komponentenauswahl
c.*f*(...)	Aufruf einer Komponentenfunktion
cp->*name*	entspricht (**cp*).*name*, sofern *cp* Zeiger auf *C*
cp->*f*(...)	entspricht (**cp*).*f*(...), sofern *cp* Zeiger auf *C*

*nur C++
**nur C++11
[1] vereinfacht

Bsp.: Komplexe Zahlen (sehr rudimentär)

```
#include <iostream>

using namespace std;

class Complex {
  private:
    double re,im;

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

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

    friend Complex conj(Complex z)  // keine Komponentenfunktion
      { z.im = -z.im; return z; }
};


int main()
{
  Complex z0, z1(4.0), z2(1.0,2.0), z3;

  z3 = conj(z2);

  double x4,y4;
  cout << "Re z4, Im z4: "; cin >> x4 >> y4;
  Complex z4(x4,y4);

  cout << "Re z0 = " << z0.real() << "   Im z0 = " << z0.imag() << endl;
  cout << "Re z1 = " << z1.real() << "   Im z1 = " << z1.imag() << endl;
  cout << "Re z2 = " << z2.real() << "   Im z2 = " << z2.imag() << endl;
  cout << "Re z3 = " << z3.real() << "   Im z3 = " << z3.imag() << endl;
  cout << "Re z4 = " << z4.real() << "   Im z4 = " << z4.imag() << endl;

  return 0;
}
```

Ausgabe:

```
Re z4, Im z4: 3 4
Re z0 = 0   Im z0 = 0
Re z1 = 4   Im z1 = 0
Re z2 = 1   Im z2 = 2
Re z3 = 1   Im z3 = -2
Re z4 = 3   Im z3 = 4
```

- In die Konstruktordefinition kann zwischen der Parameterliste und dem Funktionsblock eine Konstruktorinitialisierungsliste eingeschoben sein. Sie dient der Initialisierung der Datenkomponenten vor Ausführung des Funktionsblocks des Konstruktors.
- Parametervoreinstellungen sind auch für Konstruktordefinitionen möglich.

Überladen von Operatoren

Ein Operator @ kann (mit wenigen Ausnahmen) mittels einer Funktion `operator@` für nicht eingebaute Datentypen definiert werden, der binäre Ausdruck $x@y$ wird dann in `operator@(`x`,`y`)` umgesetzt. (Überladen ist auch für unäre Operatoren möglich, als Funktionen können auch Komponentenfunktionen eingesetzt werden). Vorrang und Syntax entspricht der für die eingebauten Operanden.

Bekanntestes Beispiel für einen überladenen Operator ist die Ein/Ausgabe mittels `>>` bzw. `<<` in der Standardbibliothek.

Bsp.: Addition und Ein/Ausgabe für komplexe Zahlen

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double re,im;
  public:
    Complex (double Re=0, double Im=0): re(Re), im(Im) { }

    friend Complex operator+(Complex z1, Complex z2)
      { return Complex(z1.re+z2.re, z1.im+z2.im); }

    friend ostream& operator<<(ostream& stream, Complex z)
      { stream << "(" << z.re << "," << z.im << ")" ;
        return stream; }

    friend istream& operator>>(istream& stream, Complex& z)
      { char c1,c2,c3;
        double x,y;
        stream >> c1 >> x >> c2 >> y >> c3;
        if (c1!='(' || c2!=',' || c3!=')')
          stream.setstate(ios::failbit);
        z = Complex(x,y);
        return stream; }
};

int main()
{
  Complex z1,z2;
  cout << "z1 z2: "; cin  >> z1 >> z2;
  cout << "z1+z2 = " << z1+z2 << endl;
  return 0;
}
```

Ausgabe:

```
z1 z2: (1,2) (3,4)
z1+z2 = (4,6)
```

Datentypen für komplexe Zahlen in der Standardbibliothek (`<complex>`)

Vordefiniert sind die Datentypen `complex<float>`, `complex<double>`, `complex<long double>`.

Im folgenden stehen x, y für reellwertige und w, z für komplexwertige Zahlen vom Typ T, worin T `float`, `double` oder `long double` bezeichnet.

Funktion	*Wirkung*
`complex<T> z`	vereinbart z mit Wert 0
`complex<T> z(x)`	vereinbart z mit Wert x
`complex<T> z(x,y)`	vereinbart z mit Wert $x + iy$
`+ - * /`	arith. Grundoperationen (unär,binär)
`= += -= *= /=`	Zuweisungen
`== !=`	Vergleiche
`cin >> z`	Eingabe im Format x, (x), (x, y)
`cout << z`	Ausgabe im Format (x, y)
`z.real() z.imag()`	re z, im z
`real(z) imag(z) conj(z)`	re z, im z, $\bar{z}$
`abs(z) norm(z)`	$\|z\|$, $\|z\|^2$
`arg(z)`	`atan2(imag(z),real(z))`
`polar(r,phi)`	$re^{i\varphi}$
`sin(z) cos(z) tan(z)`	trig. Funktionen
`exp(z) sinh(z) cosh(z) tanh(z)`	e^z, hyberbol. Funkt.
`log(z)`	$\ln z$, Verzweig. in $(-\infty, 0)$, $\operatorname{im} \ln x = \pi$ $(x < 0)$
`log10(z)`	$\ln z / \ln 10$, ln wie vorige Zeile
`pow(z,w)`	$\exp(w \ln z)$, ln wie oben
`sqrt(z)`	$\sqrt{z}$, Verzweig. in $(-\infty, 0)$, $\operatorname{im}\sqrt{x} > 0$ $(x < 0)$

Beispiel:

```
#include <iostream>
#include <complex>
#include <cmath>
#include <limits>

using namespace std;

int main()
{
  complex<double> z, i(0.0,1.0);
  cout.precision(numeric_limits<double>::digits10);

  cout << "(Re,Im): ";
  cin >> z;
  cout << "sqrt(z)     = " << sqrt(z)        << endl
       << "sqrt(1.0+i) = " << sqrt(1.0+i)    << endl;
  return 0;
}
```

Bildschirmausgabe:

```
(Re,Im): (2,3)
sqrt(z)     = (1.67414922803554,0.895977476129838)
sqrt(1.0+i) = (1.09868411346781,0.455089860562227)
```