Klassen

⟨Klasse⟩ → "`class`" [⟨Name⟩] "`{`" ⟨Komponente⟩ "`};`"
⟨Komponente⟩ → [⟨Zugriff⟩ "`:`"] {⟨Komponente⟩}
| ⟨DeklAng⟩ ⟨AttrKomp⟩ {"`,`" ⟨AttrKomp⟩} "`;`"
| ⟨Funktionsdefinition⟩
⟨AttrKomp⟩ → ⟨Deklarator⟩ ["`=`" ⟨InitialAusdruck⟩]
⟨Zugriff⟩ → "`private`" | "`public`"
⟨DeklAng⟩ → {⟨Typ⟩ | "`friend`"}

Klassen sind zusammengesetze Datentypen. Komponenten von Klassen sind über ihren Namen ansprechbar. Sie können sowohl in externen Bibliotheken, jedoch auch in eigenen Programmen vereinbart werden.

Als Komponenten von Klassen können sowohl Attribute (*data members*), wie auch Methoden (*function members*) vereinbart werden. Klassen werden verwendet indem Objekte der Klasse instanziiert werden. Jedes Objekt der Klasse hat genug Platz für jeweils seine eigene Kopien aller Attribute. Methoden werden auf Objekten aufgerufen.

Bestimmte Attribute regeln welche Teile des Programs Zugriff auf einzelne Komponenten haben. Standardmässig nur auf als `public` markierte Komponenten von außerhalb der Klasse direkt zugegriffen werden. Eine Klassenvereinbarung kann Vereinbarungen von befreundete Funktionen enthalten (`friend`), welche Zugriff auf alle Klassenkomponenten haben.

`class` C `{ ... };`	Klasse vereinbaren (nur außerhalb von Funktionen)
C c	Variable von Typ C vereinbaren; c ist Objekt der Klasse C Verwendet zur Initialisierung von c Standardkonstruktor
C c`(`init_1`,` init_2`, ...)`	Variable von Typ C vereinbaren; c initialisieren mit gegebenen Konstruktorargumenten Klammern dürfen nicht leer sein
C c`{`init_1`,` init_2`, ...}`	*direct-list-initialization*; Var. von Typ C vereinbaren; c init. mit gegebenen Konstruktorargumenten
C`()`	Konstruiere temporäres Objekt von Typ C mit Standardkonstruktor; Ausdruck nimmt temporäres Objekt als Wert an
C`{`init_1`,` init_2`, ...}`	Konstruiere temporäres Objekt von Typ C
c`.`name	Ausdruck mit Wert des aktuellen Werts des Attributs name des Objekts c
c`.`f`(...)`	Aufruf der Klassenmethode f auf das Objekt c
cp`->`name	`(*`cp`).`name
cp`->`f`(...)`	`(*`cp`).`f`(...)`

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.
Ihr Aufruf erfolgt wie oben angegeben mit eigenen syntaktischen Konstruktoren.

Zwischen Parameterliste und Funktionskörper eines Konstruktors kann eine Liste von Komponenteninitialisierungen (*member initializer list*) eingeschoben sein, die statt der Standardinitialiserungen (*default member initializer*) wie sie direkt in der Klasse vereinbart wurden, verwendet werden sollen.

Beispiel (Komplexe Zahlen — rudimentär). Im Folgenden vereinbaren wir eine Klasse zur Approximation komplexer Zahlen $z = x + iy$ durch zwei Werte vom Typ `double` um die kartesischen Koordination abzubilden.

Klasse

```
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; }

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

      return z;
    }
};
```

Im folgenden Hauptprogramm konstruieren wir mehrere Werte vom Typ `Complex`; sowohl konstant wie auch durch Einlesen von der Standardeingabe. Es werden zusätzlich einige Komponenten- und befreundete Funktionen aufgerufen.

class_demo.cpp

```
#include <iostream>

using namespace std;
```

Klasse

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

```
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 z4 = 4
```

Überladung von Operatoren

Ein beliebiger[1] Operator ∘ kann als Funktion `operator∘` definiert werden. Binäre Ausdrücke der Form $x \circ y$ werden dann übersetzt in einen Aufruf `operator∘`(x, y).

Analog können auch unäre Operatoren definiert werden.

Die `operator∘`-Funktion darf auch Methode einer Klasse sein. In diesem Fall reduziert sich die Parameterliste entsprechend.

Vorrang und Syntax derart definierter Operatoren entspricht denen der in die Sprache eingebauten.

Bekanntestes Beispiel ist die Ein- und Ausgabe vermöge der arithmetischen Shiftoperatoren >> und <<, wie sie auch in der Standardbibliothek bereitgestellt wird.

Beispiel (Ein- und Ausgabe vermöge Shiftoperatoren für komplexe Zahlen). Wir vereinbaren eine rudimentäre Klasse zur Approximation komplexer Zahlen in kartesischen Koordinaten durch Werte vom Typ `double`.

In `operator>>` lesen wir die Trennzeichen im Eingabeformat *manuell* ein und prüfen erst danach, ob sie sind, wie erwartet. Falls nicht setzen wir die Fehlerkomponente im gegebenen Eingabestrom.

Klasse

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

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

    friend Complex operator+ (Complex z1, Complex z2) {
      return Complex{ z1.x+z2.x, z1.y+z2.y };
    }

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

    friend istream& operator>> (istream& stream, Complex& z) {
      char c1, c2, c3;
      double x, y;
```

[1] mit wenigen Ausnahmen

```
      stream >> c1 >> x >> c2 >> y >> c3;
      if (c1 != '(' || c2 != ',' || c3 != ')')
        stream.setstate(ios::failbit);

      z = Complex{x, y};

      return stream;
    }
};
```

operators_demo.cpp

```
#include <iostream>

using namespace std;
```

Klasse

```
int main()
{
  Complex z1, z2;

  cout << "z1 z2: ";
  cin >> z1 >> z2;

  cout << "z1+z2 = " << z1+z2 << endl;

  return 0;
}
```

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

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

Die Datentypen `complex<float>`, `complex<double>` und `complex<long double>` sind spezialisiert vordefiniert.

Im Folgenden stehen x, y, r und φ für Variablen vom Typ T und z und w für Variablen vom Typ `complex<`T`>`. T ist hierbei ein numerischer Datentyp (`float`, `double`, `long double`).

`complex<`T`>` z	Vereinbart z mit Wert 0
`complex<`T`>` z{x}	Vereinbart z mit Wert $x + 0i$
`complex<`T`>` z{x, y}	Vereinbart z mit Wert $x + iy$
z + w	Ausdruck mit Wert $z + w$
- * /	Weitere arithmetische Grundoperatoren (- auch unär)
z = ...	Zuweisung

`z += w`	Weist z den Wert $z + w$ zu; nimmt als Ausdruck den neuen Wert von z an
`-= *= /=`	Weitere arithmetische Zuweisungsoperatoren
`z == w`	`true` wenn $z = w$, sonst `false`
`!=`	Negierte Version von `==`
`cin >> z`	Einlesen von z von der Standardeingabe in Format x, `(`x`)` oder `(`x`,` y`)` mit x und y Zeichenketten, die als T eingelesen werden können
`cout << z`	Ausgabe von z auf Standardausgabe im Format `(`x`,` y`)` mit x und y Zeichenketten
`z.real()` `z.imag()`	$\operatorname{Re} z$, $\operatorname{Im} z$
`real(z)` `imag(z)` `conj(z)`	$\operatorname{Re} z$, $\operatorname{Im} z$, $\bar{z}$
`abs(z)` `norm(z)`	$\lvert z\rvert$, $\lvert z\rvert^2$
`arg(z)`	$\operatorname{atan2}(\operatorname{Im} z, \operatorname{Re} z)$
`polar(r, φ)`	$re^{i\varphi}$
`sin(z)` `cos(z)` `tan(z)`	Trigonometrische Funktionen
`sinh(z)` `cosh(z)` `tanh(z)`	Hyperbolische Funktionen
`exp(z)`	$\exp(z)$
`log(z)`	$\log(z)$, Hauptzweig des Log., für $z \in (-\infty, 0)$ wird $\ln\lvert z\rvert + \pi i$ geliefert
`log10(z)`	$\frac{\ln z}{\ln 10}$, ln wie oben
`pow(z, w)`	$\exp(w \cdot \log(z))$
`sqrt(z)`	$\sqrt{z}$, Hauptzweig der Quadratwurzel

Beispiel (Ein-/Ausgabe von komplexen Zahlen der Standardbibliothek). Wir lesen einen Wert vom Typ `complex<double>` von der Standardeingabe ein, deklarieren einen weiteren, berechnen zwei abgeleitete Werte und geben diese aus.

complex_demo.cpp

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

using namespace std;

int main()
{
  complex<double> z, i{0.0, 1.0};

  cout << "(Re,Im): ";
  cin >> z;

  cout << fixed << setprecision(numeric_limits<double>::digits10);
```

```
  cout << "sqrt(z)     = " << sqrt(z)       << endl
       << "sqrt(1.0+i) = " << sqrt(1.0 + i) << endl;

  return 0;
}
```

```
(Re,Im): (2,3)
sqrt(z)     = (1.674149228035540,0.895977476129838)
sqrt(1.0+i) = (1.098684113467810,0.455089860562227)
```