Klassenkomponenten

- Attribute
- Methoden
- Konstruktoren
- Destruktoren
- Vereinb. befreundete Funktionen

Komponente k, Objekt $o \Longrightarrow$ i.d.R. Zugriff durch $o.k$, Aufruf durch $o.k(\ldots)$

Gültigkeitsbereich von Namen von Klassenkomponenten

- Gesamte Vereinbarung der Klasse selbst
- Alle Klassenkomponenten
- Außerhalb davon: $C::k$

Reihenfolge von Vereinbarungen von Methoden nicht relevant, Attribute i.A. schon

Nachgestellte Definition von Klassenkomponenten

post_def.cpp

```cpp
#include <iostream>

using namespace std;

class Complex {
  private:
    double re, im;

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

    double real();
    double imag();

    friend Complex conj(Complex);
};

Complex::Complex(double re_, double im_)
  : re(re_), im(im_) {}

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

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

  return z;
}
```

```cpp
int main() {
  Complex z0, z1{4.0}, z2{1.0, 2.0}, z3 = conj(z2);

  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;

  return 0;
}
```

```
Re z0 = 0   Im z0 = 0
Re z1 = 4   Im z1 = 0
Re z2 = 1   Im z2 = 2
Re z3 = 1   Im z3 = -2
```

Gültigkeitsbereich von Namen von Klassen

- Ab Vereinbarung bis Ende der Übersetzungseinheit
- Reihenfolge von Klassen i.A. wichtig

Nachgestellte Definition von Klassenkomponenten (fehlerhaft)

post_def_f.cpp

```
Complex::Complex(double re_, double im_)
  : re(re_), im(im_) {}

class Complex {
  private:
    double re, im;

  public:
    Complex(double=0, double=0);
};
```

```
post_def_f.cpp:1:1: error: ‘Complex’ does not name a type
    1 | Complex::Complex(double re_, double im_)
      | ^~~~~~
```

`this`

- `this` ist Zeiger, also Typ $C*$
- `quadriere` hat Seiteneffekt
- `quadriere` gibt Kopie von neuem Zustand zurück

Methode zum Quadrieren

```
class Complex {
  Complex quadriere() {
    double re2 = re*re - im*im,
      im2 = 2*re*im;
    re = re2; im = im2;
    return *this;
  }
};
```

Verwendung von `this`

return_this.cpp

```cpp
#include <iostream>

using namespace std;

class Complex {
  private:
    double re, im;

  public:
    Complex (double re_=0, double im_=0) : re(re_), im(im_) {}

    Complex quadriere() {
      double re2 = re*re - im*im,
        im2 = 2*re*im;
      re = re2; im = im2;
      return *this;
    }

    friend ostream& operator<<(ostream& out, Complex z) {
      return out << "(" << z.re << "," << z.im << ")";
    }
    friend istream& operator>>(istream& in, Complex& z) {
      char c1, c2, c3;
      double re_, im_;

      in >> c1 >> re_ >> c2 >> im_ >> c3;
      if (c1 != '(' || c2 != ',' || c3 != ')')
        in.setstate(ios::failbit);
      z = Complex(re_, im_);

      return in;
    }
};

int main() {
  Complex z;
  cout << "z: "; cin >> z;

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

  return 0;
}
```

```
z: (2.0, 4)
z davor:     (2,4)
z quadriert: (-12,16)
z danach:    (-12,16)
```

Verwendung von `this` für Attributzugriff

attrs_this.cpp

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

using namespace std;

class Complex {
  private:
    double re, im;

  public:
    Complex (double re_=0, double im_=0) : re(re_), im(im_) {}

    double real() { return (*this).re; }
    double imag() { return   this->im; }

    double abs() {
      return sqrt( (*this).real()*(*this).real() +
     ↪  (this->imag())*(this->imag()) );
    }
};

int main() {
  Complex z{1.0, 2.0};

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

  return 0;
}
```

```
Re z = 1   Im z = 2   |z| = 2.23607
```

Zugriffsattribute

Klassenkomponenten können sein:

- `private`
- `public`

`private` nicht sichtbar außerhalb Klassenvereinbarung/Klassenkomponenten

`private` ist Voreinstellung

Fehlermeldung bei Verstoß gegen Zugriffsattribute

private_f.cpp

```cpp
#include <iostream>

using namespace std;

class Complex {
  private:
    double re, im;

  public:
    Complex (double re_=0, double im_=0) : re(re_), im(im_) {}
};

int main() {
  Complex z{1, 2};

  cout << z.re << endl;

  return 0;
}
```

```
private_f.cpp: In function ‘int main()’:
private_f.cpp:16:13: error: ‘double Complex::re’ is private
↪  within this context
   16 |   cout << z.re << endl;
      |             ^~
private_f.cpp:7:12: note: declared private here
    7 |     double re, im;
      |            ^~
```

Konstante Methoden

- Dürfen Attribute nicht verändern
- Auf konstante Objekte können nur konstante Methoden angewandt werden
- `this` vom Typ `const` $C*$

Vereinbarung einer konst. Methode

```
class Complex {
  double real() const {
    return re;
  }
};
```

Verwendung von konstanten Methoden

const_methods.cpp

```
#include <iostream>

using namespace std;

class Complex {
  private:
    double re, im;

  public:
    Complex (double re_=0, double im_=0) : re(re_), im(im_) {}

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

int main() {
  Complex z1{1, 2};
  const Complex z2{2, 3};

  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 = 2
Re z2 = 2   Im z2 = 3
```

Verwendung von konstanten Methoden (fehlerhaft)

const_methods_f.cpp

```cpp
#include <iostream>

using namespace std;

class Complex {
  private:
    double re, im;

  public:
    Complex (double re_=0, double im_=0) : re(re_), im(im_) {}

    double real() {
      re *= 2;
      return re;
    }
    double imag() {
      return im;
    }
};

int main() {
  Complex z1{1, 2};
  const Complex z2{2, 3};

  cout <<    "Re z1 = " << z1.real()
       << "   Im z1 = " << z1.imag() << endl;
  cout <<    "Re z2 = " << z2.real()
       << "   Im z2 = " << z2.imag() << endl;
  return 0;
}
```

```
const_methods_f.cpp: In function ‘int main()’:
const_methods_f.cpp:27:35: error: passing ‘const Complex’ as
↪  ‘this’ argument discards qualifiers [-fpermissive]
   27 |   cout <<    "Re z2 = " << z2.real()
      |                           ~~~~~~~^~
const_methods_f.cpp:12:12: note:   in call to ‘double
↪  Complex::real()’
   12 |     double real() {
      |            ^~~~
const_methods_f.cpp:28:35: error: passing ‘const Complex’ as
↪  ‘this’ argument discards qualifiers [-fpermissive]
   28 |        << "   Im z2 = " << z2.imag() << endl;
      |                           ~~~~~~~^~
const_methods_f.cpp:16:12: note:   in call to ‘double
↪  Complex::imag()’
   16 |     double imag() {
      |            ^~~~
```

Konstruktoren

- Syntaktisch wie Methoden
- Name ist identisch mit Name der Klasse
- Kein Ergebnistyp, auch nicht `void`
- Können nicht `const` sein
- Mehrere Konstruktoren pro Klasse möglich
- Standardkonstruktor und Kopierkonstruktor im Zweifel automatisch erzeugt

Konstruktor

```cpp
class Complex {
  Complex (double x, double y) {
    re = x;
    im = y;
  }
};
```

Konstruktor-Initialisierungslisten

- Gibt an wie Attribute initialisert werden sollen
- Abgetrennt mit : direkt nach Parameterliste
- Nicht erwähnte Attribute ⟹ wie Standardkonstruktor

Konstruktor m. Initialisierungsliste

```
class Complex {
  Complex (double x, double y)
    : re(x), im(y) {}
};
```

Beispiele für Konstruktoren

polynom_eval.cpp

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

using namespace std;

class Polynom {
  private:
    vector<double> coeff;

  public:
    Polynom() {}
    Polynom(int n) : coeff(n + 1) { coeff[n] = 1; }
    Polynom(const vector<double>& v) : coeff(v) {}

    double eval(double x) {
      double s = 0, xpot = 1;
      for (vector<double>::size_type i = 0; i < coeff.size();
      ↪  i++) {
        s += coeff[i] * xpot;
        xpot *= x;
      }
      return s;
    }
};

int main() {
  int n;
  cout << "n: "; cin >> n;

  vector<double> v(n + 1);
  cout << "a[0] ... a[" << n << "]: ";
  for (int i = 0; i <= n; i++) cin >> v[i];

  double x;
  cout << "x: "; cin >> x;

  Polynom p{v};
  cout << "p(x) = " << p.eval(x) << endl;

  return 0;
}
```

```
n: 3
a[0] ... a[3]: 0 1 2 3
x: 1.1
p(x) = 7.513
```

```
n: 0
a[0] ... a[0]: 7
x: 3
p(x) = 7
```

Einschub: Initialisierung von Vektoren

- Init. mit $(n) \Longrightarrow$ Länge n, Elemente mit Standardkonstruktor
- Init. mit $\{\ldots\} \Longrightarrow$ Inhalt

Initialisierung eines Vektors auf gegebene Länge

```
vector<double> v(n + 1);
```

Initialisierung eines Vektors mit gegebenem Inhalt

```
vector<double> v1{7};
vector<double> v2{1, 2, 3, 4};
```

Standardkonstruktor

- Konstruktor ohne Parameter
- Automatisch erzeugt gdw. keine Konstruktoren vereinbart

Beispiele für Standardkonstruktoren

default_constr.cpp

```cpp
#include <vector>

using namespace std;

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() {}
    Polynom(int n) : a(n + 1) { a[n] = 1; }
};

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

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

int main() {
  Complex z1, z2{2}, z3{3, 4};
  const Complex i{0, 1};

  Polynom p, q{5};
  const Polynom cp, cq{5};

  Vektor v, w{3};
  const Vektor cv, cw{3};

  return 0;
}
```

Automatische Erzeugung von Konstruktoren

default_constr_f.cpp

```cpp
#include <vector>

using namespace std;

class Complex {
  private:
    double re, im;

  public:
    Complex(double re_, double im_) : re(re_), im(im_) {}
};

int main() {
  Complex z1, z3{3, 4};
  const Complex i{0, 1};

  return 0;
}
```

```
default_constr_f.cpp: In function ‘int main()’:
default_constr_f.cpp:14:11: error: no matching function for
↪  call to ‘Complex::Complex()’
   14 |   Complex z1, z3{3, 4};
      |           ^~
default_constr_f.cpp:10:5: note: candidate:
↪  ‘Complex::Complex(double, double)’
   10 |     Complex(double re_, double im_) : re(re_), im(im_) {}
      |     ^~~~~~~
default_constr_f.cpp:10:5: note:   candidate expects 2
↪  arguments, 0 provided
default_constr_f.cpp:5:7: note: candidate: ‘constexpr
↪  Complex::Complex(const Complex&)’
    5 | class Complex {
      |       ^~~~~~~
default_constr_f.cpp:5:7: note:   candidate expects 1 argument,
↪  0 provided
default_constr_f.cpp:5:7: note: candidate: ‘constexpr
↪  Complex::Complex(Complex&&)’
default_constr_f.cpp:5:7: note:   candidate expects 1 argument,
↪  0 provided
```

Kopierkonstruktor

- Konstruktor der *Referenz* auf anderes Objekt der Klasse als einzigen Parameter nimmt
- Parameter vom Typ
  - *C*`&`
  - `const` *C*`&`
- Nicht *C* oder `const` *C*
- Automatisch erzeugt, falls nicht vereinbart
- Implizit verwendet bei Parameterübergabe und Rückgabe aus Funktionen
- Keine Verwendung bei Zuweisungen (`=`)

Beispiele für Kopierkonstruktoren

copy_constr.cpp

```cpp
#include <vector>

using namespace std;

class Polynom {
  private:
    vector<double> a;

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

    Polynom(const Polynom& p) : a(p.a) {}
};

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

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

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

int main() {
  Polynom p, q{5}, r(q);
  const Polynom cp, cq{5}, cr(q);

  Vektor v, w{3}, u(w);
  const Vektor cv, cw{3}, cu(w);

  return 0;
}
```

Destruktor

- Syntaktisch wie eine Methode, Name $\sim C$
- Kein Ergebnistyp, auch nicht `void`
- Automatisch erzeugt, falls nicht vereinbart
- Explizite Destruktoraufrufe möglich, `delete`-Operator
- Implizite Destruktoraufrufe die Norm
  - Nach Auswertung des Ausdrucks bei temporären Objekten
  - Ende des Gültigkeitsbereichs bei Variablen

Destruktor

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

Speicherleck

vector_leak_demo.cpp

```cpp
#include <iostream>

using namespace std;

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

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

    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:12)
==…==    by 0x4011B9: main (vector_leak_demo.cpp:24)
==…==
==…== 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)
```

Beispiel für Destruktor

vector_destructor.cpp

```cpp
#include <iostream>

using namespace std;

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

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

    ~Vektor() {
      delete[] ap;
    }

    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_destructor
==…==
==…==
==…== HEAP SUMMARY:
==…==     in use at exit: 0 bytes in 0 blocks
==…==   total heap usage: 3 allocs, 3 frees, 73,736 bytes
↪  allocated
==…==
==…== All heap blocks were freed -- no leaks are possible
==…==
==…== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
↪  from 0)
```

Zuweisungsoperator

- `z1=z2` $\Longrightarrow$ `z1.operator=(z2)`
- Nur als Methode
- Automatisch erzeugt, falls nicht vereinbart
- Wird nicht in Initialisierung verwendet, daher *Zielobjekt* immer bereits initialisert
- U.U. muss Speicher bei Zuweisung freigegeben werden

Überladener Zuweisungsoperator

```cpp
class Complex {
  Complex& operator=(Complex& z) {
    re = z.re; im = z.im;
    return *this;
  }
};
```

Zuweisungsoperator für Klasse `Vektor`

vector_assign.cpp

```cpp
#include <iostream>

using namespace std;

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

  public:
    Vektor(int n = 0, double x = 0) : len(n) {
      ap = new double[n];
      for (int i = 0; i < n; i++) ap[i] = x;
    }
    Vektor(const Vektor& a) : len(a.len) {
      ap = new double[len];
      for (int i = 0; i < len; i++)
        ap[i] = a.ap[i];
    }
    ~Vektor() { delete[] ap; }

    Vektor& operator=(const Vektor& b) {
      delete[] ap;
      len = b.len;
      ap = new double[len];
      for (int i = 0; i < len; i++)
        ap[i] = b.ap[i];
      return *this;
    }

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

    double& at(int i) { return ap[i]; }
};

int main() {
  int n; cout << "n: "; cin >> n;
  Vektor a(n); cout << "a[0] ... a[" << (n - 1) << "]: ";
  for (int i = 0; i < n; i++) cin >> a.at(i);

  Vektor b, c{a};
  b = a;

  a.at(0) = b.at(n - 1) = 7;

  cout << "a: " << a << ", " << "b: " << b << ", " << "c: " << c
  ↪  << endl;
}
```

```
n: 4
a[0] ... a[3]: 1 2 3 4
a: 7 2 3 4, b: 1 2 3 7, c: 1 2 3 4
```

Bedeutung von =: Kopierkonstruktor oder Zuweisungsoperator

`Complex a{4.0, 5.0}`	Konstruktor
`Complex b{a}`	Kopierkonstruktor
`Complex c = a`	——"——
`Complex d = Complex{1.0, 2.0}`	——"——
`Complex e, f`	Standardkonstruktor
`e = c`	Zuweisungsoperator
`f = Complex{3.0, 5.0}`	——"——

Statische Datenkomponenten

- Deklaration innerhalb der Klasse
- Definition außerhalb der Klasse
- Zugriff/Definition mit `::`
- Besser als globale Variablen

Kreis

```
class Kreis {
  static const double pi;

  double r;
};

const double Kreis::pi = 3.14;
```

Initialisierung statischer Datenkomponenten als `constexpr`

- `constexpr` wird zur Kompilierzeit ausgewertet
- Definition statischer Datenkomponenten *innerhalb von Klasse* nur als `constexpr`

kreis_constexpr.cpp

```
#include <cmath>

class Kreis {
  static constexpr double pi
    = acos(-1);

  double r;
};

int main() {}
```

Statische Komponentenfunktionen

- Ähnlich zu befreundeten Funktionen
- `static` statt `friend`
- Kein `this`
- Aufruf innerhalb der Klasse ohne Qualifikation
- Aufruf außerhalb der Klasse mit `::`

Statische Komponentenfunktion

```
static double flaeche(Kreis k) {
  return pi*k.r*k.r;
}
```

Beispiel zur Verwendung einer statischen Komponentenfunktion

circle_static_this_demo.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;
    }
    double zylinder(double h) {
      return h * flaeche(*this);
    }
};

int main() {
  Kreis k{0.5};
  cout << "Radius:       " << k.radius() << endl
       << "Umfang:       " << k.umfang() << endl
       << "Flaeche:      " << Kreis::flaeche(k) << endl
       << "Zylindervol.: " << k.zylinder(2) << endl;

  return 0;
}
```

```
Radius:       0.5
Umfang:       3.14159
Flaeche:      0.785398
Zylindervol.: 1.5708
```