§1 Klassenkomponenten

Leitideen: *Klassen sind aus Datenkomponenten und Komponentenfunktionen zusammengesetzt. Die Komponentenfunktionen haben über eine Zeigervariable direkten Zugriff auf das Objekt.*

Die wesentlichen Eigenschaften einer Klasse können in einer Klassendeklaration angegeben werden, die sich auf eine Definition an anderer Stelle bezieht.

Konstruktoren dienen zur Initialisierung der Datenkomponenten, hierzu gehört auch die Parameterübergabe und die Rückgabe von Funktionswerten. Werden Ressourcen alloziert, so ist in der Regel ein Destruktor erforderlich.

Befreundete Funktionen haben dieselbe Aufrufsyntax wie Funktionen und Zugriff auf die Klassenkomponenten. Ähnliches gilt für statische Komp.funkt.

Statische Datenkomponenten haben in allen Objekten den gleichen Wert.

§1 Klassenkomponenten - Themenübersicht

- Vereinbarungen innerhalb von Klassen
- Gültigkeitsbereich bei Klassen I,II, Zugriffsattribute
- Komponentenfunktionen und this-Zeiger
- Konstante Komponentenfunktionen
- Konstruktoren - Eigenschaften
- Konstruktorinitialisierungsliste
- Spezielle Konstruktoren
- Zuweisungsoperator
- Statische Komponentenfunktionen
- Statische Datenkomponenten

Vereinbarungen innerhalb von Klassen

1. Datenkomponenten (konstant/nichtkonstant, statisch/nichtstatisch)
2. Komponentenfunktionen (konstant/nichtkonstant, statisch/nichtstatisch)
3. Konstruktoren (Spezialfälle: Standardkonstruktoren, Kopierkonstruktoren)
4. Destruktoren
5. Befreundete Funktionen

- Für Komponentenfunktionen Definition innerhalb der Klasse *oder* Deklaration innerhalb der Klasse und nachfolgende Definition außerhalb möglich.
- Konstruktoren und Destruktoren zählen zwar zu den Komponentenfunktionen, besitzen aber eine spezielle Vereinbarungs- und Aufrufsyntax.
- Befreundete Fkt. gehören *nicht* zu den Klassenkomp.
- Klassen*objekte* enthalten weder Funktionen noch statische Datenkomponenten.

Gültigkeitsbereich bei Klassen - Komponentenname

Def.: Teil des Programms, in dem ein Name bekannt ist

- Gültigkeitsbereich (scope) eines *Komponentennamens*:
  1. Ab Vereinbarungspunkt bis zum Ende der Klassenvereinbarung
  2. Alle Funktionsrümpfe, Konstruktorinitialisierungen und Parametervoreinstellungen der Komponentenfunktionen der Klasse (auch rückwärts und außerhalb!)
- Reihenfolge der Komponentenfunktionen soll (und darf!) keinen Einfluss auf die Programmbedeutung haben
- Reihenfolge der Datenkomponenten nicht ganz unwichtig (Initialisierungsreihenfolge)

Verwendung von Komponenten außerhalb ihrer Klasse

Betrachte Klasse *C*, Klassenkomponente *comp* und Objekt *c*

- Objektkomponente ansprechbar mit *c.comp* und $(\&c) \texttt{->} comp$
- Komponente *ohne* Objektbezug ansprechbar mit *C::comp*, z.B. Definition außerhalb der Klasse

Gültigkeitsbereich bei Klassen - Klassenname

- Gültigkeitsbereich eines *Klassennamens*:
  - Ab Vereinbarungspunkt bis zum Ende der Übersetzungseinheit (wie bei Variablen außerhalb von Funktionen)
- Beschränkung hier auf Klassen, die *außerhalb* von Funktionen vereinbart sind.
- Reihenfolge der Klassenvereinbarungen kann wichtig sein
- forward-Deklaration von Klassen möglich *(später!)*

Zugriffsattribute

Geben an, welche Funktionen Zugriff auf Komp.namen haben

- `public` - alle
- `private` - Komponentenfunktionen und mit der Klasse befreundete Funktionen
- `protected` - *später*

Komponentenfunktionen - this

- `this`: Zeiger auf das Objekt, für das die Komponentenfunktion aufgerufen wurde
  In *c.f()*: `this` $\hat{=}$ `&`*c* bzw. `*this` $\hat{=}$ *c*
- *Bsp.:*
```
class Complex {
        Complex quadriere() {
          double re2=re*re-im*im,
                 im2=2*re*im;
          re=re2; im=im2;
          return *this; }
      };
```
- Quadrieren ändert Objekt (Seiteneffekt)
- Wert des Ausdruck `z.quadriere()`: geändertes z
- Einsatz von this-Zeigern manchmal zur Verdeutlichung, dass Komponentenfunktionen auf Objekten operieren:

  Bsp.: `double real() { return (*this).re; }`

Konstante Komponentenfunktionen

- Konstante Komp.funktionen dürfen Objekt *nicht* verändern
  Kennzeichn. durch `const`-Attribut (*nach* Parameterliste!)

  T f() `const` Aufruf durch *c.f()*, falls *c* Variable, Konstante, Referenz oder Referenz auf Konstante

  T f() Aufruf durch *c.f()*, falls *c* Variable oder Referenz

  Bsp.: `double real() const { return re;}`
- Erforderlich bei Anwendung auf Konstanten oder Referenzen auf Konstanten
- Unterscheidung später wichtig bei Implementierung von Behältertypen (Indexoperator)
- Unterschiedliche Datentypen für `this`-Zeiger:

  `C *this` nichtkonstante Komponentenfunktion

  `const C *this` konstante Komponentenfunktion

Konstruktoren - Eigenschaften

Zweck: Initialisierung eines Klassenobjekts - ggf. Allokation und Belegung des unformatierten Speicherplatzes (raw storage)

- Name der Konstruktorfunktion $\equiv$ Klassenname
- Kein Ergebnistyp, auch nicht `void`
- Konstruktoren gehören wie Destruktoren zu den Komponentenfunktionen
  Expliziter Aufruf: `C()`, *nicht* `c.C()`
- `const`-Attribut unzulässig, dennoch Initialisierung auch von konstanten Objekten möglich
- Unterschiedliche Konstruktoren in einer Klasse möglich, am wichtigsten:
  Standardkonstruktor und Kopierkonstruktor - werden u.U. automatisch erzeugt, falls nicht definiert.

Konstruktorinitialisierungsliste

Zweck: Erstinitialisierung einzelner Datenkomponenten entsprechend den Konstruktorargumenten *vor* Ausführung des Konstruktorblocks

Bsp.:

```
class Polynom {
  vector<double> a;           // Pol.koeff.
  // vector<double> a(n+1); // unzulaessig
  Polynom(int n): a(n+1) {a[n]=1;}
  // Konstruktor fuer Monom x^n
}
```

- Erstinitialisierung nach Vereinbarungsreihenfolge der Komponenten, *nicht* nach Reihenfolge in der Konstr.initial.liste
- Nicht in der Konstr.initial.liste aufgeführte Komponenten werden mit dem Standardkonstruktor (Klassen) initialisiert.
- Weitere Zuweisungen im Konstruktorblock möglich
- Initialisierung von Komp., die Konstanten oder Referenzen sind, in C++98 *nur* über die Konstr.inital.liste möglich

Spezielle Konstruktoren

Standardkonstruktor (default constructor)

- Standardkonstruktor $\equiv$ Konstr. mit leerer Argumentliste
- Ggf. automatische Erzeugung mit Zugriffsattribut public: Ruft Standardkonstruktoren auf (Klassen), initialisiert aber *nicht* Datenkomponenten (eingebaute Typen)
- Einsatz z.B. bei der Vereinbarung einer Klassenvariable ohne Argumentliste

Kopierkonstruktor (copy constructor)

- Initialisiert ein Klassenobjekt durch ein anderes
- Parametertyp: `C&` bzw. `const C&`
  Unzulässig: `C` bzw. `const C`
- Ggf. automatische Erzeugung mit Zugriffsattribut public: Initialisiert Komponenten durch Zuweisung (eingebaute Datentypen) bzw. durch ihre Kopierkonstruktoren (Klassen)
- Einsatz z.B. bei Parameterübergaben oder Rückgabe von Funktionswerten
- *Keine* Verwendung bei Zuweisungen!

Destruktor

Zweck: Vernichtung eines Klassenobjekts mit dem Ziel der Freigabe des belegten Speicherplatzes

- Funktionskopf: `~C()`
- Kein Ergebnistyp, auch nicht `void`
- Ggf. automatische Erzeugung mit Zugriffsattribut public: Ruft Destruktoren der Datenkomponenten in umgekehrter Vereinbarungsreihenfolge auf
- Destruktoren vor allem nötig zur Freigabe dynamisch allozierten Speichers
- Explizite Destruktoraufrufe: nur selten erforderlich
- Implizite Destruktoraufrufe: Blockende für lokale Variablen, Programmende für statische Variablen
- In der Regel keine “garbage collection“ in C++, aber fast derselbe Effekt mit *korrekt* geschriebenen Destruktoren
- Verwendung der STL-Behälter macht Schreiben von Destruktoren oft überflüssig

Zuweisungsoperator

- Überladen des Gleichheitszeichens *nur* durch Komp.fkt.: `a=b` $\rightarrow$ `a.operator=(b)`
- Parametertyp: `C` bzw. `C&` bzw. `const C&`
- Ggf. automatische Erzeugung mit Zugriffsattribut public: Nimmt Zuweisungen für Datenkomponenten vor (eingebaute Datentypen) bzw. ruft ihre Zuweisungsoperatoren auf (Klassen)
- *Bereits erwähnt:* Zuweisungsoperator $\neq$ Kopierkonstr. Ein Unterschied: Zuweisung nur an initialisierte Objekte
- *Vorsicht:* Gleichheitszeichen in Variablendef. bewirkt Aufruf des Kopierkonstruktors und *nicht* des Zuweisungsoperators

Unterschied zum Kopierkonstruktor beim vektor-Beispiel

- delete-Aufruf, weil interner C-Vektor bereits belegt ist und evtl. die Länge geändert wird

Statische Datenkomponenten

Zweck: Speicherung von Werten, die in *allen* Objekten einer Klasse gleich sind.

Bsp.:

```
struct Kreis {
  double r;
  static const double pi;
}
const double Kreis::pi = acos(-1.0)
```

- Deklaration innerhalb der Klasse
- Definition außerhalb der Klasse mit Namenszugriffoperator
- Statische Datenkomponenten oft besser als globale Variable
- Größe von Objekten durch die Hinzunahme statischer Datenkomponenten unverändert
- Anmerkung: Die Größe der Objekte hängt *nur* von der Größe (und Anordnung) der nichtstatischen *Daten*komponenten ab. Ausnahme: Virtuelle Funktionen (*später!*)

Statische Komponentenfunktionen

- Deklarationsangabe `static` (anstelle von `friend`)
- Zugriff auf die privaten Komponenten von Klassenobjekten
- Jedoch: Kein this-Zeiger
- Außerhalb Klasse: Aufruf durch `C::f()`
  [sogar `c.f()` möglich, `c.` wird ignoriert]
- Innerhalb Klasse: Aufruf auch möglich, wenn kein Parameter vom Klassentyp vorhanden.
  (Unterschied zu friend-Funktionen!)
- Vereinbarungsreihenfolge innerhalb der Klasse unwichtig (wie bei normalen Komponentenfunktionen, Konstruktoren und Destruktoren)
- Falls public, Wirkung ähnlich wie befreundete Funktionen. Vermeidung von Namenskonflikten
- Falls private, gut geeignet für Hilfsfunktionen, die in der Klasse untergebracht werden sollen, aber keinen Parameter vom Klassentyp haben

Vergleich befreundeter Fkt. und stat. Komp.fkt.

- Befreundete Funktionen gehören nicht zur Klasse, selbst wenn sie dort definiert sind.
  [Verhalten als ob unmittelbar nach der Klasse definiert]
- Zugriff auf die privaten Klassenkomponenten, kein this-Zeiger (ebenso wie bei stat. Komp.fkt.)
- Außerhalb der Klasse: Aufruf durch `f()` (ab Vereinbarungspunkt)
- Beliebige Vereinbarungsreihenfolge in der Klasse wie bei Komponentenfunktionen (sofern mindestens ein Parameter vom Klassentyp vorhanden)
- STL: häufiger befreundete Funktionen als statische Komponentenfunktionen
  [Durch Includedateien Einschub der Klassendefinition und somit auch der befreundeten Funktionen, daher direkter Aufruf von `f()` möglich, `C::f()` wäre umständlicher]