Kopierkonstruktoren

Der Kopierkonstruktor (copy constructor) dient zur Initialisierung eines neuen Klassenobjekts durch ein anderes. Es handelt sich dabei um eine Komponentenfunktion mit Klassenname (Bez. C), die mit dem Parametertyp C`&` bzw. `const` C`&` vereinbart werden muß. Unzulässig sind die Parametertypen C und `const` C.

Falls nicht vorhanden, wird der Kopierkonstruktor automatisch mit Zugriffsattribut `public` erzeugt und initialisiert die Komponenten des Klassenobjekts schrittweise durch Zuweisung (eingebaute Datentypen) oder durch Aufruf der Kopierkonstruktoren (Klassen).

Der Kopierkonstruktor kommt bei der Parameterübergabe und der Rückgabe von Funktionswerten zum Einsatz. Er wird allerdings *nicht* für Zuweisungen benutzt.

Bsp.: Kopierkonstruktoren für die oben aufgeführten Klassen

```
Complex(const Complex& z) : re(z.re), im(z.im) {}
                          :
Polynom(const Polynom& p) : a(p.a) {}
                          :
Vektor(const Vektor& a) {
  len = a.len;
  ap = new double[len];
  for (int i=0; i<len; ++i)
    ap[i] = a.ap[i];
}
```

Die Definition der Kopierkonstruktoren für die Klassen `Complex` und `Polynom` wäre hier nicht erforderlich, weil sie das Verhalten der automatisch erzeugten Kopierkonstruktoren wiedergeben.

Destruktoren

`~`C`()` als Funktionskopf vereinbart einen Destruktor für die Klasse C. Wie die Konstruktoren hat auch der Destruktor keinen Ergebnistyp (auch nicht `void`).

Der Destruktoraufruf vernichtet das Klassenobjekt, auf das er angewendet wird. Ein weiterer Destruktoraufruf für ein bereits zerstörtes Klassenobjekt ist nicht erlaubt, auch wenn er implizit erfolgt.
Implizite Destruktoraufrufe erfolgen u.a. beim Verlassen eines Blocks für dort definierte Objekte mit temporärer Lebensdauer (automatische Variablen).

Falls nicht vorhanden, wird der Destruktor automatisch erzeugt und ruft die Destruktoren der Klassenkomponenten in umgekehrter Reihenfolge der Konstruktoren auf.

Bsp.: Destruktoren für die Klasse `Vektor`

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

Für die Klassen `Complex` und `Polynom` sind keine expliziten Destruktordefinitionen erforderlich und sollten vermieden werden. Dagegen ist die Destruktordefinition für die Klasse `Vektor` notwendig, um „Speicherlecks“ zu vermeiden.

Bsp.: Speicherleck

```
#include <iostream>
#include <new>
#include <unistd.h>      // Fuer sleep-Funktion aus C

using namespace std;

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

void f(void)
{
  int n=1000000;
  Vektor a(n);
  return;
}

int main()
{
  for (int i=1; i<100; ++i) {
    cout << i << ".Aufruf von f()" << endl;
    f();
    sleep(1);
  }
  return 0;
}
```

Zuweisungsoperatoren

Der Zuweisungsoperator kann nur durch eine Komponentenfunktion überladen werden, zulässig ist in der Vereinbarung genau ein Parameter vom Typ C, C`&` oder `const` C`&`.

Falls keine derartige Komponentenfunktion vorhanden ist, wird sie erzeugt und führt komponentenweise die Zuweisung aus.

Die Zuweisung darf nicht mit den Kopierkonstruktoren verwechselt werden, auch wenn das in Initialisierungen gelegentlich vorkommende Gleichheitszeichen das nahelegt.

Bsp.:

```
Complex a(4.0,5.0);          // Initialisierung (Konstruktor)
Complex b(a);                // Initialisierung (Kopierkonstruktor)
Complex c=a;                 // Initialisierung (Kopierkonstruktor)
Complex d=Complex(1.0,2.0); // Initialisierung (Kopierkonstruktor)
Complex e,f;                 // Initialisierung (Standardkonstruktor)
e=c;                         // Zuweisung (Zuweisungsoperator)
f=Complex(3.0,5.0);          // Zuweisung (Zuweisungsoperator)
```

Ein wesentlicher Unterschied ist, dass die Zuweisung auf bereits initialisierte Klassenobjekte angewandt wird.

Bsp.: Zuweisungsoperator für die Klasse `Vektor`

```
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;
}
```

Der für die Klasse `Vektor` definierte Kopierkonstruktor und der Zuweisungsoperator stellen die „Wertsemantik" sicher, d.h. bei Nichtreferenz-Parameterübergaben und Zuweisungen werden Kopien der Objekte angelegt. Das entspricht dem Verhalten der Standard Template Library (STL) für die Behälterdatentypen.

Statische Datenkomponenten

Datenkomponenten können innerhalb einer Klassenvereinbarung als `static` *deklariert* werden. Ihre *Definition* muß außerhalb der Klassenvereinbarung erfolgen und eindeutig sein. (Für die Definition außerhalb der Klasse wird dann der Namenszugriffsoperator `::` benötigt.)

Statische Datenkomponenten haben für alle Objekte ihrer Klasse denselben Wert. Sie sind `nicht` über `this`-Zeiger ansprechbar und sind auch *nicht* in den Klassenobjekten enthalten. Ihre Initialisierung erfolgt außerhalb der Klasse und außerhalb von Funktionen.

Der Zweck statischer Datenkomponenten ist die Vermeidung globaler Variablen.

Bsp.: Klasse `Kreis` *mit statischer Datenkomponente für* π

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

using namespace std;

class Kreis {

  private:
    double r;
    static const double pi;

  public:
    Kreis(double r_=0): r(r_) {}
    double radius()  { return r; }
    double umfang()  { return 2*pi*r; }
    friend double flaeche(Kreis k) { return pi*k.r*k.r; }
};

const double Kreis::pi = acos(-1.0);
```

```
int main()
{
  double r;
  cout << "r: "; cin >> r;
  Kreis kreis(r);
  cout << "Radius="  << kreis.radius()  << "  "
       << "Umfang="  << kreis.umfang()  << "  "
       << "Flaeche=" << flaeche(kreis)  << endl;
  return 0;
}
```

Bemerkung: Die Initialisierung statischer Datenkomponenten innerhalb der Klasse ist in C++11 nur sehr eingeschränkt möglich. Hier könnte man `static constexpr double pi = M_PI` verwenden. (Mit `const` würde es nicht funktionieren.)

Statische Komponentenfunktionen

Statische Komponentenfunktionen können im Unterschied zu statischen Datenkomponenten innerhalb einer Klasse definiert werden.

Statische Komponentenfunktionen haben Zugriff auf die `private`-Komponenten von Klassenobjekten, allerdings ist der `this`-Zeiger für sie nicht definiert. (Die Aufrufe *c*`.`*f*`()` und *cp*`->`*f*`()` sind zwar möglich, der vordere Teil *c*`.` bzw. *cp*`->` wird aber ignoriert.)

Außerhalb der Klasse erfolgt der Aufruf üblicherweise über den Namenszugriffsoperator, d.h. *C*`::`*f*`()`), die Verwendung von *f*`()` wie bei befreundeten Funktionnen ist nicht möglich.

Bsp.: Statische Komponentenfunktion an Stelle von befreundeter Funktion

```
                     :
class Kreis {
                     :
  public:
                     :

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

                     :

int main()
{

  cout <<           ...

       << "Flaeche=" << Kreis::flaeche(kreis)  << endl;     // geaendert

                     :
}
```