5.7 Einführung und Klassen/Element-Init.-Liste im Konstruktor

5.7.1 Aggregation und Konstruktoren

Die Klasse `zeitraumtyp` enthält nun also zwei Zeitpunkte `von` und `bis`. Für `zeitraumtyp` wurden auch allerhand Methoden erstellt. Aber wie ist es mit den Konstruktoren (und Destruktoren) der einzelnen Komponenten der Aggregation?

Das Beispielprogramm `minifirma2a.cpp` enthält eine Aggregation `minifirma` mit drei Objekten der Klasse `person`. Jeder Konstruktor ist durch eine `cout`-Zeile gekennzeichnet.

```
1  #include <iostream>
2  #include <string.h>
3  using namespace std;
4  class person
5  {
6  private:
7     char vorname[41];
8     char nachname[41];
9  public:
10    person(void)
11    {
12       cout << "std-ctor person" << endl;
13       strcpy(vorname,"Standard");
14       strcpy(nachname,"Standard");
15    }
16    person(const char *v, const char *n)
17    {
18       cout << "ctor_2 person" << endl;
19       strncpy(vorname,v,40); vorname[40]='\0';
20       strncpy(nachname,n,40); nachname[40]='\0';
21    }
22    void pdisplay(void)
23    {
24       cout << "Ich heisse " << vorname << " " << nachname << ".\n";
25    }
26 };
27
28 class minifirma
29 {
30 private:
31    char name[41];
32    person meister;
33    person geselle;
34    person azubi;
35 public:
36    minifirma(const char *n)
37    {
38       cout << "ctor_1 minifirma" << endl;
39       strncpy(name,n,40); name[40]='\0';
40    }
41    void fdisplay(void)
42    {
43       cout << "Firma: " << name << endl;
44       cout << "Meister:"; meister.pdisplay();
45       cout << "Geselle:"; geselle.pdisplay();
```

```
46        cout << "Azubi: "; azubi.pdisplay();
47    }
48 };
49
50 int main(void)
51 {
52    minifirma hwb("Elektro-Meier");
53    hwb.fdisplay();
54    return 0;
55 }
```

Der Aufruf ergibt:

Terminal

```
schueler@debian964:~$ src/minifirma2a.cpp; ./a.out
std-ctor person
std-ctor person
std-ctor person
ctor_1 minifirma
Firma: Elektro-Meier
Meister:Ich heisse Standard Standard.
Geselle:Ich heisse Standard Standard.
Azubi: Ich heisse Standard Standard.
```

Es zeigt sich: Zuerst werden die Standard-Konstruktoren der einzelnen Komponenten aufgerufen, anschließend der Konstruktor der Aggregation. Erst werden die Teile konstruiert, dann die Gesamtkonstruktion.

5.7.2 Initialisierung von Komponenten

Wie kann man nun den Personen von Elektro-Meier mit Hilfe eines Konstruktors Namen geben (damit nicht alle einen Standardnamen haben)?

Eine Möglichkeit zeigt das Programm `minifirma2b.cpp`:

```
1  #include <iostream>
2  #include <string.h>
3  using namespace std;
4  class person
5  {
6  private:
7     char vorname[41];
8     char nachname[41];
9  public:
10    person(void)
11    {
12       cout << "std-ctor person" << endl;
13       strcpy(vorname,"Standard");
14       strcpy(nachname,"Standard");
15    }
16    person(const char *v, const char *n)
17    {
18       cout << "ctor_2 person" << endl;
19       strncpy(vorname,v,40); vorname[40]='\0';
20       strncpy(nachname,n,40); nachname[40]='\0';
21    }
22    void pdisplay(void)
```

```
23      {
24          cout << "Ich␣heisse␣" << vorname << "␣" << nachname << ".\n";
25      }
26  };
27
28  class minifirma
29  {
30  private:
31      char name[41];
32      person meister;
33      person geselle;
34      person azubi;
35  public:
36      minifirma(const char *n)
37      {
38          cout << "ctor_1␣minifirma" << endl;
39          strncpy(name,n,40); name[40]='\0';
40      }
41      minifirma(const char *n, person m, person g, person a)
42      {
43           cout << "ctor_4␣minifirma" << endl;
44           meister = m;
45           geselle = g;
46           azubi   = a;
47           strncpy(name,n,40); name[40]='\0';
48      }
49      void fdisplay(void)
50      {
51          cout << "Firma:␣" << name << endl;
52          cout << "Meister:"; meister.pdisplay();
53          cout << "Geselle:"; geselle.pdisplay();
54          cout << "Azubi:␣";  azubi.pdisplay();
55      }
56  };
57
58  int main(void)
59  {
60      person mei("Hannes","Bauer");
61      person ges("Alex","Volt");
62      person azu("Simon","Ohm");
63      minifirma hwb("Elektro-Meier", mei, ges, azu);
64      hwb.fdisplay();
65      return 0;
66  }
```

Der Aufruf ergibt:

Terminal

```
schueler@debian964:~$ g++ src/minifirma2b.cpp; ./a.out
ctor_2 person
ctor_2 person
ctor_2 person
std-ctor person
std-ctor person
std-ctor person
```

```
ctor_4 minifirma
Firma:
...
```

Man erstellt also drei Objekte außerhalb der Aggregation (3 Konstruktoren). Dann wird die Aggregation mit 3+1 Konstruktoren aufgebaut. Zum Schluss werden die drei Objekte in die Aggregation kopiert.

5.7.3 Vereinfachung durch die Element-Initialisierer-Liste

Kann man das auch einfacher und schneller haben, ohne die Standardkonstruktoren der Komponenten?

Die Vereinfachung ist in `minifirma2c.cpp` eingebaut:

```
1  #include <iostream>
2  #include <string.h>
3  using namespace std;
4  class person
5  {
6  private:
7     char vorname[41];
8     char nachname[41];
9  public:
10    person(const char *v, const char *n)
11    {
12       cout << "ctor_2 person" << endl;
13       strncpy(vorname,v,40); vorname[40]='\0';
14       strncpy(nachname,n,40); nachname[40]='\0';
15    }
16    void pdisplay(void)
17    {
18       cout << "Ich heisse " << vorname << " " << nachname << ".\n";
19    }
20 };
21
22 class minifirma
23 {
24 private:
25    char name[41];
26    person meister;
27    person geselle;
28    person azubi;
29 public:
30    minifirma(const char *n, person m, person g, person a):
31              meister(m), geselle(g), azubi(a)
32    {
33        cout << "ctor_4 minifirma" << endl;
34        strncpy(name,n,40); name[40]='\0';
35    }
36    void fdisplay(void)
37    {
38       cout << "Firma: " << name << endl;
39       cout << "Meister:"; meister.pdisplay();
40       cout << "Geselle:"; geselle.pdisplay();
```

```
41        cout << "Azubi: "; azubi.pdisplay();
42    }
43 };
44
45 int main(void)
46 {
47    person mei("Hannes","Bauer");
48    person ges("Alex","Volt");
49    person azu("Simon","Ohm");
50    minifirma hwb("Elektro-Meier", mei, ges, azu);
51    hwb.fdisplay();
52    return 0;
53 }
```

Jetzt ergibt der Aufruf :

Terminal

```
schueler@debian964:~$ g++ src/minifirma2c.cpp; ./a.out
ctor_2 person
ctor_2 person
ctor_2 person
ctor_4 minifirma
```

Jetzt wird also kein Standardkonstruktor mehr aufgerufen. Wie wird nun konstruiert? Die entscheidende Zeile ist:

```
1 minifirma(const char *n, person m, person g, person a):
2           meister(m), geselle(g), azubi(a)
```

Die Liste hinter dem Doppelpunkt ist die *Element-Initialisierer-Liste* (EIL). Sie besagt: `meister` wird mit `(m)` initialisiert, `geselle` mit `(g)` und `azubi` mit `(a)`. Mit der EIL kann man zu jeder Komponente festlegen, mit welchem Konstruktor und mit welchen Parametern sie konstruiert werden soll.

Es geht z.B. auch so:

```
1 minifirma(const char *n, const char *mv, const char *mn, person g):
2           meister(mv, mn), geselle(g)
```

Jetzt wird zu `meister` der Konstruktor aufgerufen, der zwei Zeichenketten erfordert, zu `geselle` wird der Inhalt von `g` kopiert und zu `azubi` wird der Standard-Konstruktor benutzt.

5.7.4 Kopierkonstruktor

In `minifirma2c.cpp` taucht in der Ausgabe kein Konstruktor für die Komponenten von `minifirma` auf. Ist keiner vorhanden?

Neben dem automatischen Standardkonstruktor (ASK), der nur bei Fehlen aller anderen Konstruktoren vorhanden ist, gibt es noch einen weiteren automatisch generierten Konstruktor, den Kopierkonstruktor (cp-ctor). Er tritt in Kraft bei:

a) Übergabe eines Objektes an die Parameterliste einer Funktion

```
1    vergleiche_datum(d1,d2);
```

b) Rückgabe eines Objektes durch eine Funktion

```
1    d1 = scandatum();
```

c) Initialisierung eines Objektes durch ein anderes

```
1  datum d1(31,12,2012);
2  datum d2(d1); // oder datum d2=d1;
```

d) Initialisierung eines Objektes durch ein anderes in der EIL

```
1  zeitraumtyp(datumtyp d1, datumtyp d2): von(d1), bis(d2)
```

Der Kopierkonstruktor hat den folgenden Prototyp:

```
1  persontyp(persontyp &p);
```

Man kann ihn, wenn man will, durch eine eigene Version überschreiben:

```
1  persontyp(persontyp &p)
2  {
3      strcpy(vorname, p.vorname);
4      strcpy(nachname, n.nachname);
5      cout << "cp-ctor von person" << endl;
6  }
```

Das Ergebnis (`minifirma2d.cpp`):

Terminal

```
ctor_2 person
ctor_2 person
ctor_2 person
cp-ctor person
cp-ctor person
cp-ctor person
cp-ctor person
cp-ctor person
cp-ctor person
ctor_4 minifirma
...
```

In der Regel braucht man das Überschreiben des Kopierkonstruktors nur, wenn man Objekte mit dynamisch erzeugtem Speicher (`malloc()/free()` oder `new/delete`) verarbeiten will. Andernfalls braucht man sich um ihn nicht zu kümmern.