7.2 Dynamischer Speicher in Objekten/Kopierkonstruktor

7.2.1 Situation

Das Programmbeispiel aus dem letzten Abschnitt läßt sich bisher nicht zufriedenstellend zu Ende bringen. Unerklärliche Abstürze und Fehlverhalten (*Schutzverletzung* bzw. *segmentation fault*) treten auf. Um den Problemen auf den Grund gehen zu können, wird im folgenden nur noch mit `mitglied.h` und `mitglied.cpp` gearbeitet.

7.2.2 Beispiel: noch keine Probleme

- `mitglied.h`:

```
1 #ifndef _MITGLIED_H_
2 #define _MITGLIED_H_
3 class mitglied
4 {
5 private:
6     char *name;
7     char *motto;
8 public:
9     mitglied(void);
10    mitglied(const char *nname, const char *nmotto);
11    ~mitglied(void);
12    void setName(const char *nname);
13    const char *getName(void);
14    void setMotto(const char *nmotto);
15    const char *getMotto(void);
16    void printit(void);
17 };
18 #endif
```

- `mitglied.cpp`:

```
1 #include <iostream>
2 #include <string.h>
3 using namespace std;
4 #include "mitglied.h"
5 //-----
6 mitglied::mitglied(void)
7 {
8     cout << "mitglied::std-ctor" << endl;
9     name = new char[1];
10    name[0] = '\0';
11    motto = new char[1];
12    motto[0] = '\0';
13 }
14 //-----
15 mitglied::mitglied(const char *nname, const char *nmotto)
16 {
17     cout << "mitglied::ctor" << endl;
18     if (!nname) nname="";
19     if (!nmotto) nmotto="";
20
21     name = new char[ strlen(nname)+1];
```

```

22     strcpy(name, nname);
23     motto = new char[strlen(nmotto)+1];
24     strcpy(motto, nmotto);
25 }
26 //-----
27 mitglied::~mitglied(void)
28 {
29     cout << "mitglied::dtor" << endl;
30     delete [] motto;
31     delete [] name;
32 }
33 //-----
34 void mitglied::setName(const char *nname)
35 {
36     if(!nname) nname="";
37     if(name) delete [] name;
38     name = new char[strlen(nname)+1];
39     strcpy(name, nname);
40 }
41 //-----
42 const char *mitglied::getName(void){ return name;}
43 //-----
44 void mitglied::setMotto(const char *nmotto)
45 {
46     if(!nmotto) nmotto="";
47     if(motto) delete [] motto;
48     motto = new char[strlen(nmotto)+1];
49     strcpy(motto, nmotto);
50 }
51 //-----
52 const char *mitglied::getMotto(void){ return motto;}
53 //-----
54 void mitglied::printit(void)
55 {
56     cout << "Ich_heisse_" << name
57         << "_und_mein_Motto_ist:_" << motto << endl;
58 }

```

- main1.cpp

```

1 #include <iostream>
2 using namespace std;
3 #include "mitglied.h"
4
5 int main(void)
6 {
7     mitglied a("Heinz", "Laber_-_Rhabarber");
8     mitglied b;
9     mitglied c("Willi", "Keine_Ahnung_-_kein_Problem");
10    mitglied d("Egon", "Ohne_Preis_kein_Fleiss");
11    a.printit();
12    b.printit();
13    c.printit();
14    d.printit();

```

```

15     return 0;
16 }

```

- Übersetzung: `c++ mitglied.cpp main1.cpp`
- Ergebnis: 4 x Konstruktor, 4 x Destruktor

7.2.3 Änderung des Programms: Initialisierung eines Objektes

- In der Zeile `mitglied b=a` wird ein Objekt direkt bei der Deklaration durch ein anderes Objekt initialisiert (das ist schon bei Standard-C erlaubt, dort allerdings nur für einfache Datentypen wie `int` oder `double`).
- `main2.cpp`

```

1 #include <iostream>
2 using namespace std;
3 #include "mitglied.h"
4
5 int main(void)
6 {
7     mitglied a("Heinz", "Laber__Rhabarber");
8     mitglied b=a;
9     mitglied c("Willi", "Keine_Ahnung__kein_Problem");
10    mitglied d("Egon", "Ohne_Preis__kein_Fleiss");
11    a.printit();
12    b.printit();
13    c.printit();
14    d.printit();
15    return 0;
16 }

```

- Übersetzung: `c++ mitglied.cpp main2.cpp`
- Ergebnis: Der Destruktor von `a` schlägt offenbar fehl.

7.2.4 Versuch einer Fehlerbehebung

- Weitere Änderung: `b.setName("fritz");`
- `main3.cpp`

```

1 #include <iostream>
2 using namespace std;
3 #include "mitglied.h"
4
5 int main(void)
6 {
7     mitglied a("Heinz", "Laber__Rhabarber");
8     mitglied b=a;
9     mitglied c("Willi", "Keine_Ahnung__kein_Problem");
10    mitglied d("Egon", "Ohne_Preis__kein_Fleiss");
11    b.setName("Walter");
12    a.printit();
13    b.printit();

```

```

14 |     c.printit ();
15 |     d.printit ();
16 |     return 0;
17 | }

```

- Übersetzung: `c++ mitglied.cpp main3.cpp`
- Ergebnis: Wiederum ein Fehlschlag (oder ein merkwürdiges Verhalten),
 - zuerst bei der Aushgabe von `a`,
 - dann beim Destruktor von `a`.

7.2.5 Woher rührt dieses Problem?

- Durch die Initialisierung von `b` mit dem Inhalt von `a` wird jede Komponente von `a` nach `b` kopiert, auch die Zeiger `name` und `motto`.

7.2.6 Wie kann man das lösen?

- Hier ist es sinnvoll, die Ausgaben der Programme vorher und nachher zu vergleichen:
 - Vorher: 4 x ctor, 4 x dtor
 - Nachher: 3 x ctor, 4 x dtor
- Offenbar wird bei der Erstellung eines Objektes mit anschließender Initialisierung mit dem Inhalt eines anderen Objektes nicht mehr unser Konstruktor aufgerufen, sondern ein automatisch erzeugter Konstruktor des Systems.
- Der automatisch erzeugte Standard-Konstruktor des Systems kann es nicht sein, der wird ja nicht erzeugt, da wir (mindestens) einen eigenen Konstruktor für diese Klasse gebaut haben.
- Also muss es ein anderer automatisch erzeugter Konstruktor sein: Der Kopier-Konstruktor.

7.2.7 Wann wird ein Kopier-Konstruktor aufgerufen?

Der Kopierkonstruktor wird aufgerufen

- bei der Initialisierung eines Objektes mit dem Inhalt eines anderen Objektes,
 - Hinweis: In C ist die erste Form vorgeschrieben.
 - In C++ gilt die zweite Form als üblich und die erste nur als Kurzform für die zweite.
 - Man soll daraus sehen: Es handelt sich *nicht* um eine Zuweisung, sondern um eine Methode der Klasse des Objektes `b`; das Objekt `a` ist nur ihr Parameter!

```

1 | mitglied b=a;
2 | mitglied b(a);

```

- bei der Rückgabe eines Objektes mit `return` und

```

1 | return b;

```

- bei der Übergabe eines Objektes als Call-By-Value-Parameter an eine Funktion oder Methode,

```

1  verein::aufnahme(mitglied mg){...}
2  ...
3  int main(void)
4  {
5      ...
6      sportverein.aufnahme(b); // hier wird ein Kopierkonstruktor
7                               // benutzt!
8      ...
9  }

```

- nicht jedoch bei der Übergabe eines Objektes als Zeiger oder als Call-By-Reference-Parameter an eine Funktion oder Methode.

```

1  verein::aufnahme1(mitglied &mg){...}
2  verein::aufnahme2(mitglied *pmg){...}
3  ...
4  int main(void)
5  {
6      ...
7      sportverein.aufnahme1(b); // kein Kopierkonstruktor
8      sportverein.aufnahme2(&b); // hier auch nicht!
9      ...
10 }

```

7.2.8 Wann baut man einen eigenen Kopierkonstruktor?

- Der Kopierkonstruktor *kann* durch einen selbstgebauten Kopierkonstruktor ersetzt werden.
- Der automatisch erzeugte Kopierkonstruktor kopiert einfach Element für Element von `a` nach `b`; es entsteht eine so genannte flache Objektkopie.
- Für Objekte ohne dynamischen Speicher (also alle Objekte ohne durch Zeigervariablen realisierte Attribute reicht diese flache Objektkopie vollkommen aus.
- Bei Objekten mit Zeigervariablen (z.B. `char* name` oder `char* motto`) dagegen ist das problematisch, da `a.name` und `b.name` auf denselben Speicherbereich zeigen. Ändert man den Namen von `a`, ändert sich auch der von `b`. Und wenn `a.name` mit `delete` gelöscht wird (der Destruktor von `a` sollte so etwas tun), wird damit auch `b.name` ungültig.
- Deshalb muss ein selbstgebauter Kopierkonstruktor eine so genannte tiefe Objektkopie anlegen: Jeder Zeiger eines Attributes des Objekts muss auf einen eigenen, selbst allozierten Speicherbereich zeigen.
- **Erfahrungswert:** Immer, wenn ein Objekt Zeigervariablen enthält, muss man einen eigenen
 - a) Kopierkonstruktor,
 - b) Destruktor,
 - c) Zuweisungsoperator (siehe später!)

schreiben.

7.2.9 Wie baut man einen eigenen Kopierkonstruktor?

- Jeder Konstruktor mit folgender Parameterliste ersetzt den automatisch erzeugten Kopierkonstruktor:

```
1 (const <klasse> &)
```

- Beispiel für einen Konstruktor der Klasse `mitglied`:

```
1 mitglied::mitglied(const mitglied &mg)
2 {
3 }
```

- Es handelt sich um eine Festlegung innerhalb der Sprache C++, die allerdings sehr sinnvoll ist:
 - Durch `const` wird verhindert, dass `mg` verändert werden kann (der Compiler passt dadurch darauf auf).
 - Durch `&` wird dafür gesorgt, dass kein Objekt kopiert werden muss, denn dafür ist ja gerade der Kopierkonstruktor gedacht; außerdem wäre das sehr ineffizient.
 - Statt des Call-By-Reference wäre auch eine Übergabe per Zeiger denkbar, wurde aber in der Sprache C++ nicht ausgewählt.
 - Weiteres Argument für `&` (oder `*`): Bei der Deklaration des Kopierkonstruktors in der Klasse `x` ist die Klasse `x` noch nicht vollständig bekannt; damit ist innerhalb der Klasse `x` nur `&rx` und `*px` erlaubt.
- Da `b` und `a` zur gleichen Klasse gehören, darf der Kopierkonstruktor von `b` auf die privaten Datenelemente (wie z.B. `name`) von `a` zugreifen, was in diesem Fall auch geschieht (wäre hier allerdings nicht notwendig gewesen; es gibt ja die Methode `mitglied::getname()`).
- Ergebnis:

```
1 mitglied::mitglied(const mitglied &mg)
2 {
3     name = new char[ strlen(mg.name)+1];
4     strcpy(name, mg.name);
5     motto = new char[ strlen(mg.motto)+1];
6     strcpy(motto, mg.motto);
7 }
```

Dieser selbst geschriebene Kopierkonstruktor muss auch in der Headerdatei genannt werden, damit er von anderen Modulen (z.B. von `main()` aus) gesehen wird – andernfalls würde wieder der automatische Kopierkonstruktor aufgerufen und der selbst geschriebene übersehen.