5.3 Einführung und Klassen/Konstruktor und Destruktor

5.3.1 Konstruktor statt `init()`-Funktion

Bei den bisherigen Beispielen gab es mehrere Probleme:

- Wenn man ein Objekt anlegt, ist sein Zustand unbestimmt (der Informatiker sagt dazu *undefiniert*, meint aber dasselbe).
- Man könnte die Attribute durch Initialisierung füllen, und so haben wir das bisher immer gemacht:

```
1    struct pkwtyp p={"MI-LK 123", 12345};
```

  Nur jetzt sind die Attribute mit `private` vor jedem Zugriff, aber auch vor der Initialisierung, geschützt.

- Eine Initialisierung direkt in der Klassendefinition ist nicht möglich[1]:

```
1  struct pkwtyp
2  {
3  private:
4     char kennz[20]="";   // geht so nicht
5     int kmstand=0;       // geht so nicht
6    ...
7  };
```

- Man kann dieses Problem dadurch lösen, dass man zu jeder Klasse eine `init()`-Methode schreibt. Sie gehört natürlich zu ihrer Klasse und hat daher Zugriff auf die Attribute:

```
1  struct pkwtyp
2  {
3  private:
4     char kennz[20];
5     int kmstand;
6  public:
7     void init(void)
8     {
9        kennz[0]='\0';
10       kmstand=0;
11    }
12 };
```

  Diese `init()`-Methode hat aber wieder zwei Nachteile:

- Sie wird nicht automatisch aufgerufen. Nach jeder Variablen-Vereinbarung *muss* die `init()`-Methode aufgerufen werden, sonst steht irgendein Unsinn im Objekt und kann das ganze Programm gefährden.

```
1     struct pkwtyp p, q, r, s, t;
2     p.init();
3     q.init();
4     r.init();
5     s.init();
6     t.init();
```

[1] Einschränkung: Im neuen C++-Standard wird genau das erlaubt.

Das ist also umständlich und gefährlich.

- Die `init()`-Methode kann man auch später noch aufrufen und den Kilometerstand wieder zurücksetzen. Aber das will ich nicht, wenn ich die Klasse konstruiere.

Man braucht also zu jeder Klasse eine `init()`-Methode,

a) die automatisch aufgerufen wird, sobald ein Objekt angelegt wird (egal, wie es angelegt wird)

b) die aber sonst *nie* aufgerufen werden kann

Diese Methode gibt es in C++. Sie heißt *Konstruktor* (abgekürzt Ctor). Der Konstruktor einer Klasse wird automatisch immer dann aufgerufen, wenn ein Objekt erstellt wird, z.B. bei Beginn der Lebensdauer eines statischen Objektes oder beim Anlegen eines dynamischen Objektes mit `new`. Anders kann man ihn nicht aufrufen.

5.3.2 Ein erster Standardkonstruktor für die Klasse `pkwtyp`

Wie wird nun in C++ der Konstruktor einer Klasse realisiert? Leider hat man nicht festgelegt, dass der Konstruktor immer `ctor()` heißt. Dann könnte man in jeder Klasse schnell den Konstruktor finden. Stattdessen hat der Konstruktor in jeder Klasse den gleichen Namen wie die Klasse selbst. In der Klasse `pkwtyp` heißt der Konstruktor also ebenfalls `pkwtyp()`. Hier ein Beispiel (`src/pkw3.cpp`):

```
1  #include <stdio.h>
2  #include <string.h>
3  struct pkwtyp
4  {
5  private:
6      char kennz[20];
7      int kmstand;
8  public:
9      pkwtyp(void)
10     {
11         kennz[0]='\0';
12         kmstand=0;
13     }
14     void printpkw(void)
15     {
16         printf("%s, %d\n", kennz, kmstand);
17     }
18 };
19 /*******************************************/
20 int main(void)
21 {
22     struct pkwtyp neuwagen; // jetzt wird er aufgerufen
23     neuwagen.printpkw();
24     return 0;
25 }
```

Zeile 9 Hier beginnt der Konstruktor. Er heißt genauso wie die zugehörige Klasse, nämlich `pkwtyp`.

Zeile 10+11 Der Konstruktor hat wie jede Methode Zugriff auf `private`-Elemente seiner Klasse.

Zeile 22 Hier wird ein Objekt der Klasse `pkwtyp` angelegt, es wird Speicherplatz belegt. Und direkt danach wird der Konstruktor `pkwtyp()` für dieses Objekt aufgerufen.

Zeile 8 Ein Konstruktor muss `public` sein, sonst kann man kein Objekt anlegen, das ihn benutzt.

Der Konstruktor ist also die Methode, die so heißt wie ihre eigene Klasse. Wenn man genau hinsieht, fällt auf (Zeile 9): Er hat gar keinen Rückgabetyp, nicht einmal `void`. Das ist logisch, denn einen Konstruktor kann man nicht aufrufen. Wenn ein Konstruktor einen Rückgabewert hätte, könnte man diesen Wert nirgendwo abholen.

Was schreibt man nun in einen Konstruktor? Zunächst kann man in einen Konstruktor, wie in jede Methode, alles schreiben, was man für richtig hält. Es gibt also keine besonderen Einschränkungen. Nur zurückgeben kann man aus einem Konstruktor nichts. Normalerweise wird man die Attribute initialisieren, Dateien öffnen oder andere Dinge tun, um das Objekt in einen Zustand zu bringen, dass man damit arbeiten kann.

5.3.3 Weitere Konstruktoren für die Klasse `pkwtyp`

Der im Beispiel dargestellte Konstruktor initialisiert jedes Objekt gleich. Jedes Kennzeichen bekommt den Inhalt `""` und jeder Kilometerstand den Wert `0`. So einen Konstruktor, der alle gleich behandelt, nennt man Standard-Konstruktor.

Aber es gibt noch mehr: Bei der `init()`-Funktion konnte man Parameter mitgeben. Das kann man beim Konstruktor auch:

```
1 struct pkwtyp
2 {
3   ...
4 public:
5     pkwtyp(int mein_kmstand)
6     {
7         kmstand=mein_kmstand;
8     }
9 };
```

Sobald dieser Konstruktor in der Klasse `pkwtyp` steht, kann man ein Objekt auch wie folgt anlegen:

```
1 struct pkwtyp p(1234);
```

Nun wird beim Anlegen der Wert `1234` als aktueller Parameter in den ersten formalen Parameter des Konstruktors kopiert – wie bei einem Funktionsaufrif. So bekommt der Kilometerstand von `p` beim Anlegen den Wert 1234.

Man kann auch einen Konstruktor mit mehreren Parametern anlegen:

```
1 struct pkwtyp
2 {
3   ...
4 public:
5     pkwtyp(const char *mein_kennz, int mein_kmstand)
6     {
7         kennz[0]='\0';
8         strncat(kennz, mein_kennz, 19);
9         kmstand=mein_kmstand;
10    }
11 };
```

Jetzt kann man ein Objekt so anlegen:

```
1 struct pkwtyp p("OP–EL␣9000", 1234);
```

Übrigens kann man alle diese Konstruktoren gleichzeitig in derselben Klasse haben, obwohl sie alle gleich heißen. Das liegt daran, dass auch bei Methoden das **Overloading** erlaubt ist.

5.3.4 Anlegen von Objekten mit und ohne Parameter

Beim Anlegen von Objekten muss man auf die Schreibweise achten:

a) Mehrere Parameter:

```
1    struct pkwtyp p("OP–EL␣9000", 1234);
```

Die Parameter werden wie beim Funktionsaufruf in eine Paar runder Klammern geschrieben.

b) Ein Parameter:

```
1    struct pkwtyp p(1234);
2    struct pkwtyp p=1234;
```

Hier gibt es zwei Schreibweisen. Die obere ist eine logische Fortsetzung der Klammerschreibweise, die untere wird gerne dann verwendet, wenn es sich um eine Umwandlung handelt wie hier:

```
1    class bruch x=7.0;
```

Hier suggeriert die Schreibweise, dass der Wert `7.0` in das Objekt, den Bruch umgewandelt wird.

c) Kein Parameter:

```
1    struct pkwtyp p;
```

Diese Schreibweise ist kompatibel zu C.
Aber Vorsicht: Die Schreibweise `struct pkwtyp p();` wäre falsch. Sie beschreibt nämlich `p` als den Prototypen einer Funktion mit dem Rückgabewert `struct pkwtyp`. Das ist aber etwas ganz anderes.

Wenn ich ein Objekt mit bestimmten Parametern anlegen will, muss ich natürlich vorher den entsprechenden Konstruktor erstellt haben.

5.3.5 Konstruktoren, Standardkonstruktoren und automatische Standardkonstruktoren

Durch die bisherigen Beispiel ermuntert schreibe ich folgendes Programm (`pkw3defekt.cpp`):

```
1  #include <stdio.h>
2  #include <string.h>
3  struct pkwtyp
4  {
5  private:
6     char kennz[20];
7     int kmstand;
8  public:
9     pkwtyp(const char *knz, int kms)
10    {
```

```
11          kennz[0]='\0';
12          strncat(kennz,knz,19);
13          kmstand=kms;
14      }
15      void printpkw(void)
16      {
17          printf("%s, %d\n", kennz, kmstand);
18      }
19  };
20  /*********************************************/
21  int main(void)
22  {
23      struct pkwtyp fw("BI–XY123",20000);
24      struct pkwtyp neuwagen;
25      struct pkwtyp flotte[80];
26      neuwagen.printpkw();
27      return 0;
28  }
```

Leider bekomme ich eine Fehlermeldung:

Terminal

```
schueler@debian964:~$ g++ pkw3defekt.cpp
pkw3defekt.cpp: In function 'int main()':
pkw3defekt.cpp:24:18: error: no matching function for call to
                            'pkwtyp::pkwtyp()'
   24 |     struct pkwtyp neuwagen;
      |                   ^~~~~~~~
```

Zeile 24 sieht eigentlich ganz harmlos aus. Es soll nur ein Objekt vom Typ `struct pkwtyp` angelegt werden, ganz ohne Parameter. Und genau das ist offenbar jetzt nicht mehr möglich. Aber warum?

Aus Gründen der Kompabilität mit C hat man in C++ verfügt, dass jede C++-Klasse einen sogenannten *automatischen Standardkonstruktor (ASK)* erhält. Er ist leer, tut also nichts:

```
1       pkwtyp(void)
2       {}
```

Dieser automatische Standardkonstruktor ist also in jeder C++-Klasse automatisch und unsichtbar mit dabei.

Und jetzt die Besonderheit: Das gilt nur, solange kein weiterer Konstruktor existiert. Sobald man auch nur einen eigenen Konstruktor für eine Klasse geschrieben hat, ist der automatische Standardkonstruktor verschwunden. Dann muss man sich einen eigenen Standardkonstruktor schreiben. Der kann auch leer sein. Wenn man das weiß, ist es kein Problem. Das berichtigte Programm sieht damit so aus (`pkw3b.cpp`):

```
1   #include <stdio.h>
2   #include <string.h>
3   struct pkwtyp
4   {
5   private:
6       char kennz[20];
7       int kmstand;
8   public:
9       pkwtyp(const char *knz, int kms)
10      {
```

```
11          kennz[0]='\0';
12          strncat(kennz,knz,19);
13          kmstand=kms;
14      }
15      pkwtyp(void) /* Standard-Konstruktor */
16      {
17          kennz[0]='\0';
18          kmstand=0;
19      }
20      void printpkw(void)
21      {
22          printf("%s, %d\n", kennz, kmstand);
23      }
24  };
25  /*****************************************/
26  int main(void)
27  {
28      struct pkwtyp fw("BI-XY123",20000);
29      struct pkwtyp neuwagen;
30      struct pkwtyp flotte[80];
31      neuwagen.printpkw();
32      return 0;
33  }
```

5.3.6 Konstruktoren und Arrays von Objekten

Wenn man ein Array von Objekten anlegen will, braucht man entweder einen Standardkonstruktor oder einen Konstruktor mit einem Parameter:

```
1   pkwtyp flotte[4]; // Standard-Konstruktor, 4x aufgerufen
2   pkwtyp fuhrpark[3]={45,20,20};//Konstr.mit 1x int, 3x aufgerufen
3   pkwtyp garage[2]={"AU-DI 4000",1234,"BM-W 1602",5678};//verboten
```

Die unterste Zeile ist in C++ nicht erlaubt, weil sie schnell zu Mehrdeutigkeiten führt, besonders in Verbindung mit automatischer Typkonvertierung.

5.3.7 Destruktor statt `exit()`-Funktion

Analog zum Konstruktor gibt es in vielen Sprachen auch einen Destruktor (abgekürzt Dtor). Er wird immer aufgerufen, wenn die Lebensdauer eines statischen Objektes endet und beim Freigeben von Speicherplatz mit `delete` oder `delete[]`. Man kann ihn dazu benutzen, einen Speicherbereich aufzuräumen oder offene Dateien zu schließen. Oft braucht man ihn aber nicht.

5.3.8 Destruktor in C++

Der Destruktor heißt in C++ wie der Klassenname mit einer vorangestellten Tilde:

```
1   struct pkwtyp
2   {
3    ...
4   public:
5       ~pkwtyp(void)
6       {
7           kennz[0]='\0';
8           kmstand=0;
```

```
9     }
10 };
```

Zeile 5 Hier ist er, wie der Konstruktor ohne Rückgabetyp.

Zeile 7+8 Hier kann man Daten überschreiben, damit man später im RAM keine Rest mehr davon findet (Sicherheit)

Zeile 4 Auch der Destruktor muss `public` sein, sonst könnte man ihn nicht benutzen.

Wie schon der Konstruktor hat auch der Destruktor keinen Rückgabewert. Darüberhinaus ist seine Parameterliste *immer* leer. Er kann also auch nicht überladen werden. Wenn man keinen eigenen Destruktor schreibt, existiert ein leerer Standarddestruktor.

Im Gegensatz zum Konstruktor könnte man den Destruktor übrigens auch explizit aufrufen (dazu später im Zusammenhang mit Vererbung).

5.3.9 Konstruktor und Destruktor sichtbar machen

Die Aufrufe von Konstruktoren und Destruktoren erfolgen unsichtbar. Wenn man sie sichtbar machen will, kann man einfache Bildschirmausgaben hineinprogrammieren:

```
1  struct pkwtyp
2  {
3    ...
4  public:
5      pkwtyp(void)
6      {
7          printf("ctor\n");
8      }
9      ~pkwtyp(void)
10     {
11         printf("dtor\n");
12     }
13 };
```

5.3.10 Konstruktor und Destruktor in UML

Konstruktor und Destruktor tauchen im Klassendiagramm der UML nicht auf.