5.4 Einführung und Klassen/Externe Definition von Methoden

5.4.1 Auslagern innerhalb einer Datei

Die Definitionen der Klassen `bruch`, `datum` und `person` sind mittlerweile größer geworden. Besonders die Methoden-Definitionen nehmen viel Platz weg.

Die Übersichtlichkeit leidet darunter. Wenn man einen schnellen Überblick haben will, reichen eigentlich die Attribute der Klasse und die Kopfzeilen jeder Methode. Wie die Methode nun intern implementiert ist, *will* man in der Regel gar nicht wissen.

Deshalb gibt es die Möglichkeit, in der Klasse nur die Prototypen der Methoden aufzulisten. Die Methoden-Definitionen können dann in weiter unten in derselben Datei folgen. Es ist also genau so wie bei den Prototypen der Funktionen in C – mit einer Ausnahme:

In C muss jede Funktion anders heißen als die andere. Denn Funktionen sind global und der Compiler „sieht“ alle gleichzeitig. In C++ dagegen liegen die Methoden in ihrer jeweils eigenen Klasse. Deshalb kann in C++ *jede* Klasse z. B. eine Methode namens `print()` und eine Methode namens `isok()` haben¹.

Die Lösung liegt darin, dass jede Methode außerhalb ihrer Klasse über den Klassennamen angesprochen wird. Der Klassenname ist quasi der Familienname der Methode, ihr eigener Name ist der Vorname. Der Operator, mit dem der Klassenname angegeben wird, heißt in C++ *Namensraum-Operator* und besteht aus zwei unmittelbar aufeinanderfolgenden Doppelpunkten.

In `pkw4a.cpp` sieht man, wie das praktisch aussieht:

```

1 #include <stdio.h>
2 #include <string.h>
3 /******//
4 struct pkwtyp
5 {
6     private:
7         char kennz[20];
8         int kmstand;
9     public:
10        pkwtyp(const char *knz, int kms);
11        void printpkw(void);
12    };
13    /*-----*/
14    pkwtyp::pkwtyp(const char *knz, int kms)
15    {
16        kennz[0] = '\0';
17        strcat(kennz, knz, 19);
18        kmstand=kms;
19    }
20    /*-----*/
21    void pkwtyp::printpkw(void)
22    {
23        printf("%s, %d\n", kennz, kmstand);
24    }
25    /******//
26    int main(void)
27    {
28        struct pkwtyp fw("BI-XY123", 20000);
29        fw.printpkw();
30        return 0;
31    }

```

¹... und das auch noch mit denselben Parametern, so dass das Überladen auch nicht helfen kann!

5.4.2 Auslagern in eine eigene Datei

Im obigen Quelltext können die Methoden auch unterhalb von `main()` liegen. Durch die Prototypen weiß der Compiler ja, ob die Parameter stimmen.

Wenn man noch weiter geht, kann man die Definitionen der Methoden auch aus der Datei herausnehmen und in eine eigene Datei packen. Und diese Datei kann man für sich compilieren. Der Benutzer bekommt dann nur die Klassendeklaration mit Attributen und Prototypen zu sehen. Das ist sinnvoll, wenn er gar nicht sehen *soll*, wie die Methoden realisiert worden sind.

Damit kann man in C++ genauso gut Module erzeugen wie in C. Wenn man nur eine Klasse hat, kann das so aussehen:

- Die Klassendeklaration (Attribute und Prototypen) kommt in die so genannte Headerdatei `klassenname.h`.
- Die Methodendefinitionen kommen in die Datei `klassenname.cpp`. Dort wird zu Beginn `klassenname.h` mit `#include` eingebunden.
- Das Hauptprogramm kommt in die Datei `main.cpp`. Auch dort wird `klassenname.h` eingebunden.
- Jedes Modul kann für sich mit `g++ klassenname.cpp` einzeln compiliert werden; das Ganze wird mit `g++ *.o` zusammengebunden. Oder man ruft `g++ *.cpp` auf und compiliert damit alles auf einmal.

Im Beispiel von `pkw.cpp` sieht das wie folgt aus. Zunächst ist da die Headerdatei `pkwtyp.h` mit der Klassendeklaration:

```

1  #ifndef _PKWTYP_H
2  #define _PKWTYP_H
3  struct pkwtyp
4  {
5  private:
6      char kennz[20];
7      int kmstand;
8  public:
9      pkwtyp(const char *knz, int kms);
10     void printpkw(void);
11 };
12 #endif

```

Hier kommt die Datei `pkwtyp.cpp` mit den Methodendefinitionen:

```

1  #include <stdio.h>
2  #include <string.h>
3  #include "pkwtyp.h"
4  /*-----*/
5  pkwtyp::pkwtyp(const char *knz, int kms)
6  {
7      kennz[0] = '\0';
8      strcat(kennz, knz, 19);
9      kmstand=kms;
10 }
11 /*-----*/
12 void pkwtyp::printpkw(void)
13 {
14     printf("%s, %d\n", kennz, kmstand);
15 }

```

Und zum Schluss muss es noch die Datei `main.cpp` mit der `main()`-Funktion geben:

```
1 #include "pkwtyp.h"
2 int main(void)
3 {
4     struct pkwtyp fw("BI-XY123",20000);
5     fw.printpkw();
6     return 0;
7 }
```

5.4.3 Auch hier: der Include-Wächter

Die Zeilen 1, 2 und 12 in der Headerdatei `pkwtyp.h` bilden den so genannten *Include-Wächter*². Er sorgt dafür, dass in jeder Datei, die `pkwtyp.h` einliest, diese Headerdatei nur einmal eingelesen wird. Wenn man viele Klassen hat, die sich zum Teil gegenseitig bedingen, kann es passieren, dass eine Headerdatei auf zwei Wegen eingelesen wird. Dann meckert der Compiler, dass er die eingelesenen Informationen schon kennt und bricht seine Arbeit ab. Und genau dagegen hilft eben der Include-Wächter:

Zeile 2 Er definiert ein Symbol, das (so gehört es sich) so heißt wie die Headerdatei.

Zeile 1 Aber wenn er dieses Symbol schon kennt, dann liest er den Bereich bis zu Zeile 11 gar nicht mehr ein.

Zeile 12 Hier endet der Bereich, der mit dem Include-Wächter geschützt wird. Das sollte gleichzeitig das Dateiende sein.

²Der Include-Wächter ist keine Erfindung von C++. Es gibt ihn genauso auch in C.