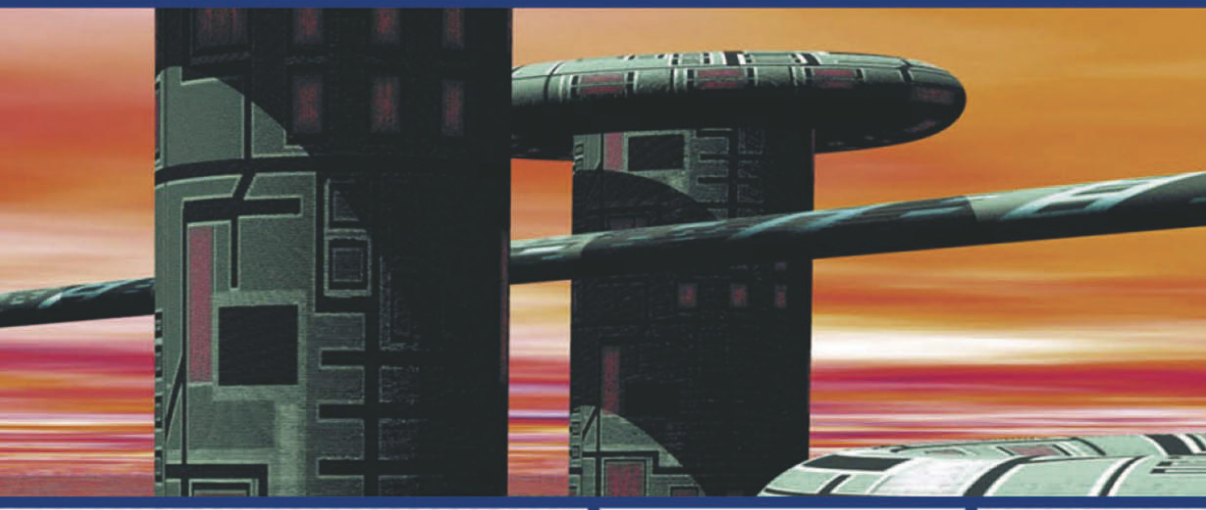
Marko Meyer

C++ programmieren



im Klartext

PEARSON
Studium

MARKO MEYER

C++ PROGRAMMIEREN

IM KLARTEXT

ein Imprint von Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Wenn wir nun versuchen, auf einen Datenmember direkt zuzugreifen ...

```
1  int main()
2  {
3      Kunde K("Müller", "Helmut", "Musterdorf");
4
5      K.kaufe(99.95);
6
7      // Fehlerhafter Zugriff:
8      cout << "Umsatz von Herrn Müller: " << K.Umsatz << endl;
9  }
```

... meldet der Compiler einen Fehler bei der Compilierung:

```
kunde.h:19: error: 'double Kunde::Umsatz' is private
main.cc:10: error: within this context
```

Auf diese Weise haben wir effektiv die Forderung durchgesetzt, dass der Nutzer der Klasse niemals direkt auf die Datenmember zugreifen kann – wir können diese also nach unseren Bedürfnissen verändern, ohne dass ein Nutzer seinen Code ändern muss.

Leider haben wir es den Nutzern von `struct Kunde` nun unmöglich gemacht, sich den Umsatz oder die Anzahl von Transaktionen eines Kunden anzuschauen. Wir haben die Nutzer auf die von uns in Form von Memberfunktionen bereitgestellte *Schnittstelle* der Klasse `Kunde` beschränkt.

Infolgedessen ist es unsere Pflicht, dafür zu sorgen, dass diese Schnittstelle alle Aktionen enthält, die für die sinnvolle Benutzung von Objekten der Klasse erforderlich sein könnten. Wir bieten also Memberfunktionen an, mit deren Hilfe wir den Umsatz und die Anzahl der Transaktionen lesen können, sowie Namen, Vornamen und Wohnort:

```
1   // kunde.h:
2
3   struct Kunde
4   {
5   private:
6       // Datenmember
7   public:
8       // Memberfunktionen bisher
9
10      std::string name() const;
11      std::string vorname() const;
12      std::string wohnort() const;
13      double umsatz() const;
14      int transaktionen() const;
15  };
16
```

```
17  // kunde.cc:
18  // Memberfunktionen wie bisher:
19  string Kunde::name() const
20  {
21      return Name;
22  }
23
24  string Kunde::vorname() const
25  {
26      return Vorname;
27  }
28
29  string Kunde::wohnort() const
30  {
31      return Wohnort;
32  }
33
34  double Kunde::umsatz() const
35  {
36      return Umsatz;
37  }
38
39  int Kunde::transaktionen() const
40  {
41      return Transaktionen;
42  }
```

Die Memberfunktionen werden nur dafür benutzt, Datenmember zu lesen, daher sollten wir sie als `const` deklarieren. Wie in Kapitel 2.2 erläutert, können wir damit erreichen, dass der Compiler uns meldet, wenn wir aufgrund eines Programmierfehlers versuchen sollten, die Datenmember zu verändern.

Mit den angegebenen Funktionen erlauben wir dem Nutzer unserer Klasse lediglich den Lesezugriff. Wir möchten jedoch auch die Änderung von Name, Vorname und Wohnort des Kunden erlauben. Lediglich Umsatz und Transaktionszähler bleiben unter Kontrolle der Klasse – sie werden durch die Benutzung der Methode `kaufe` implizit verändert:

```
1   // kunde.h:
2
3   struct Kunde
4   {
5   private:
6       // Datenmember
7
8   public:
9       // Memberfunktionen wie bisher
10      void name(std::string const&);
```

```
11      void vorname(std::string const&);
12      void wohnort(std::string const&);
13  };
14
15  //kunde.cc:
16
17  // Memberfunktionen wie bisher:
18  void Kunde::name(string const& n)
19  {
20      Name = n;
21  }
22
23  void Kunde::vorname(string const& v)
24  {
25      Vorname = v;
26  }
27
28  void Kunde::wohnort(string const& w)
29  {
30      Wohnort = w;
31  }
```

Wir verwenden die gleichen Namen für die Memberfunktionen zum Ändern der Werte wie für die Memberfunktionen zum Lesen der Werte. Der Compiler kann die verschiedenen Funktionen dann anhand der übergebenen Parameter unterscheiden – dies entspricht dem in Kapitel 1.5 dargestellten Prinzip der Überladung für einfache Funktionen, das folglich auch für Memberfunktionen verwendet werden kann. Memberfunktionen können zusätzlich noch anhand ihrer `const`-heit überladen werden. Eine `const`-Memberfunktion kann also neben einer nicht-`const`-Memberfunktion mit gleichem Namen und gleicher Parameterliste existieren. Die `const`-Memberfunktion wird verwendet, wenn die Funktion auf einem als `const` deklarierten Objekt aufgerufen wird.

Wir wollen die soeben entwickelten Memberfunktionen in einer entsprechenden Hauptfunktion `main` testen:

```
1  int main()
2  {
3      // wie bisher
4
5      K1.wohnort("Musterdorf");
6
7      cout << "Herr " << K1.name() << " wohnt jetzt in " << K1.wohnort()
8           << "." << endl;
9  }
```

Ich habe das neue Programm kunde2 genannt und erhalte beim Start folgende Ausgabe:

```
[mme@endeavour Kunde3]: ./kunde2
Kunde Helmut Müller mit ID 0 aus Musterdorf hatte 1 Transaktion
und 99.95 EUR Umsatz.
Kunde Herbert Meier mit ID 1 aus Beispielstadt hatte 2 Transaktionen
und 176 EUR Umsatz.
Herr Meier wohnt jetzt in Musterdorf.
```

Wir sehen, dass sich der Wohnort des zweiten Kunden nach dem Aufruf der Funktion `Kunde::wohnort(string const&)` geändert hat. Außerdem können wir das Funktionieren der Lesefunktionen verifizieren.

Bei Klassen mit vielen Datenmembern ist es ratsam, genau zu prüfen, auf welche davon direkter Lese- und/oder Schreibzugriff gewährt werden soll. Da die Entwicklung der Zugriffsfunktionen stets nach dem gleichen Schema abläuft, gibt es in Entwicklungsumgebungen gegebenenfalls Möglichkeiten, Zugriffsfunktionen „auf Knopfdruck“ automatisch generieren zu lassen.

2.6 class

Im vorhergehenden Abschnitt haben wir den Zugriffsschutz auf Datenmember und Memberfunktionen diskutiert. Wir haben ebenfalls bereits erkannt, dass innerhalb einer `struct` ohne Angabe von Zugriffsschutzmarken stets alle Member als `public` deklariert sind.

In C++ existiert zur Definition einer eigenen Klasse auch das Schlüsselwort `class`. Dieses kann anstelle von `struct` verwendet werden. Dabei ändert sich der Zugriffsschutz: ohne Angabe von Zugriffsschutzmarken sind alle Member als `private` deklariert. Dies unterstützt den Autor der Klasse dabei, bewusst zu entscheiden, welche Member er als öffentlich nutzbare Schnittstelle verwenden will, und welche nicht.

Wir werden künftig stets `class` zur Definition einer Klasse verwenden. Es sei aber noch einmal darauf hingewiesen, dass `struct` den selben Zweck erfüllt, wobei lediglich ein anderer Standard-Zugriffsschutz gegeben ist.

Im Folgenden sehen wir noch einmal einen Überblick über die momentan zum Projekt gehörende Klasse und ihre Implementierung:

```
1   // kunde.h
2   class Kunde
3   {
4       // Standard-Zugriffsschutz ist private
5
6       static int max_ID;
7
```

```
8       // Stammdaten
9       int ID;
10      std::string Name;
11      std::string Vorname;
12      std::string Wohnort;
13
14      // Umsatz und Anzahl der Einzeltransaktionen
15      double Umsatz;
16      int Transaktionen;
17
18  public:
19      Kunde()
20          : ID(max_ID++), Umsatz(0), Transaktionen(0)
21      {}
22
23      Kunde(std::string const& Name,
24            std::string const& Vorname,
25            std::string const& Wohnort);
26      Kunde(Kunde const& K);
27
28      void kaufe(double Umsatz);
29      std::ostream& schreibe(std::ostream& out) const;
30
31      std::string name() const;
32      std::string vorname() const;
33      std::string wohnort() const;
34      double umsatz() const;
35      int transaktionen() const;
36
37      void name(std::string const&);
38      void vorname(std::string const&);
39      void wohnort(std::string const&);
40
41      static int get_max_ID();
42  };
43
44  // kunde.cc
45  int Kunde::max_ID(0);
46
47  Kunde::Kunde(string const& name, string const& vorname,
48               string const& wohnort)
49      :ID(max_ID++), Name(name), Vorname(vorname), Wohnort(wohnort),
50       Umsatz(0), Transaktionen(0)
51  {
```

```
52  }
53
54  Kunde::Kunde(Kunde const& K)
55      :ID(K.ID), Name(K.Name), Vorname(K.Vorname), Wohnort(K.Wohnort),
56       Umsatz(K.Umsatz), Transaktionen(K.Transaktionen)
57  {
58  }
59
60  void Kunde::kaufe(double umsatz)
61  {
62          Umsatz += umsatz;
63          Transaktionen++;
64  }
65
66  ostream& Kunde::schreibe(ostream& out) const
67  {
68          out << "Kunde " << Vorname << " " << Name
69              << " mit ID " << ID <<" aus " << Wohnort << " hatte "
70              << Transaktionen
71              << ((Transaktionen == 1)?" Transaktion":" Transaktionen")
72              << " und " << Umsatz << " EUR Umsatz." << endl;
73
74          return out;
75  }
76
77  int Kunde::get_max_ID()
78  {
79      return Kunde::max_ID;
80  }
81
82  string Kunde::name() const
83  {
84      return Name;
85  }
86
87  string Kunde::vorname() const
88  {
89      return Vorname;
90  }
91
92  string Kunde::wohnort() const
93  {
94      return Wohnort;
95  }
96
97  double Kunde::umsatz() const
98  {
99      return Umsatz;
100 }
```

```
101
102  int Kunde::transaktionen() const
103  {
104      return Transaktionen;
105  }
106
107  void Kunde::name(string const& n)
108  {
109      Name = n;
110  }
111
112  void Kunde::vorname(string const& v)
113  {
114      Vorname = v;
115  }
116
117  void Kunde::wohnort(string const& w)
118  {
119      Wohnort = w;
120  }
```

2.7 Operatoren

Basisdatentypen wie `int`, `double` oder `char` können mit Operatoren verknüpft werden. Für eigene Klassen kann der Entwickler dieses Verhalten durch Definition spezieller Funktionen und Memberfunktionen erreichen.

Diese Funktionen tragen im Funktionsnamen das Schlüsselwort operator, sowie das Operatorenzeichen selbst.

Operatoren können sowohl als Member- als auch als eigenständige Funktionen implementiert werden, wobei für bestimmte Operatoren vorzugsweise die eine oder andere Implementationsweise verwendet wird.

Wir werden uns im Folgenden mit der Implementierung einiger ausgewählter Operatoren beschäftigen. Weitere Hinweise können dann der Literatur entnommen werden.

2.7.1 Zuweisungsoperator (operator=)

Der Zuweisungsoperator = wird benutzt, um den Wert eines Objekts (einer Variable) in ein anderes Objekt (Variable) zu übertragen. Der Wert des Zielobjektes geht dabei verloren.

Beispiele für Zuweisungen:

```
1          // Definition von Variablen/Objekten:
2          int i;
3          int j;
4          Kunde K1;
5          Kunde K2;
6
```

```
7           // ... Modifikation der Objekte ...
8
9           // Zuweisungen:
10          i = j;
11          K2 = K1;
12
13          // ...
```

Zuweisungen können verkettet werden:

```
1           int i, j, k;
2
3           // ...
4           i = j = k;
```

Dabei haben am Ende sowohl `i`, als auch `j` den Wert von `k`. Dies erfordert, dass das Ergebnis einer Zuweisungsoperation der zugewiesene Wert sein muss.

Da die Zuweisung – wie alle anderen Operationen auch – als Funktion dargestellt wird, können wir aus den bisherigen Darstellungen folgende Forderungen an diese Funktion ableiten:

- Die Rückgabe einer Zuweisungsoperation ist der zugewiesene Wert (Verkettung),
- übergeben wird der Wert, der zugewiesen werden soll (rechte Seite der Zuweisung),
- die Zuweisung ist eine Memberfunktion, denn der Wert eines Objektes (also seine Datenmember) soll geändert werden.

Eine weitere Forderung, deren Hintergrund sich in Abschnitt 11.2.1 zeigen wird, ist die Zusicherung, dass Zuweisungen eines Objektes an sich selbst erkannt und behandelt werden sollen.

Wir wollen für unsere `Kunden`-Klasse einen Zuweisungsoperator entwickeln:

```
1   // kunde.h:
2   class Kunde
3   {
4
5           // Datenmember wie bisher
6
7   public:
8           // Konstruktoren wie bisher
9
10          Kunde& operator=(Kunde const&);
11
12          // Memberfunktionen wie bisher
13  };
14
```

```
15  // kunde.cc:
16  Kunde& Kunde::operator=(Kunde const& K)
17  {
18      if(this != &K) {
19          ID = K.ID;
20          Name = K.Name;
21          Vorname = K.Vorname;
22          Wohnort = K.Wohnort;
23          Umsatz = K.Umsatz;
24          Transaktionen = K.Transaktionen;
25      }
26
27      return *this;
28  }
```

In der Implementierung des Zuweisungsoperators vergleichen wir zunächst, ob wir das Objekt an sich selbst zuweisen. Dazu verwenden wir das Schlüsselwort `this`, das einen Zeiger auf das Objekt darstellt, für das die betreffende Funktion gerade aufgerufen wird. Wir vergleichen `this` mit der Adresse des (als `const`-Referenz) übergebenen Objektes, das wir zuweisen wollen. Nur wenn ein Unterschied zwischen `this` und der Adresse des übergebenen Objektes besteht, haben wir zwei Objekte an verschiedenen Speicherplätzen vorliegen (und damit zwei unterschiedliche Objekte) und wir weisen jeden Datenmember des übergebenen Objektes an den jeweiligen Datenmember des Objektes, auf dem die Funktion aufgerufen wird, zu.

Am Ende der Funktion geben wir das eigene Objekt als Rückgabewert zurück. Damit stellen wir sicher, dass die Verkettung von Zuweisungsoperationen möglich ist.

Um den Zusammenhang zwischen den verschiedenen an der Zuweisung beteiligten Objekten darzustellen, wollen wir die Anweisung

```
1           K1 = K2;
```

aus dem obigen Beispiel auseinandernehmen. Wir nehmen dabei an, dass `K1` und `K2` Objekte der Klasse `Kunde` sind. Der Compiler versteht die obige Zuweisung wie den Aufruf einer Memberfunktion namens `operator=`:

```
1           K1.operator=(K2);
```

Wenn wir nun die Implementierung des Zuweisungsoperators betrachten, so sehen wir, dass `this` in diesem Fall der Adresse von `K1` entspräche, während `K2` das übergebene Objekt `K` repräsentiert. Das Ergebnis der Zuweisung (bzw. des Aufrufs der Memberfunktion `operator=`) wird in diesem Fall verworfen.

Stellen wir uns vor, wir hätten zusätzlich zu den beiden eben erwähnten Objekten ein drittes, nämlich `K3`, ebenfalls eine Instanz der Klasse `Kunde`. Wir schreiben nun folgende Zuweisung:

```
1           K3 = K2 = K1;
```

Der Compiler sieht dies als zweifachen Aufruf der Memberfunktion `operator=`:

```
1           K3.operator=(K2.operator=(K1));
```

Dies bedeutet, dass das Ergebnis des Aufrufs von `K2.operator=(K1)` diesmal nicht verworfen, sondern als Übergabeparameter für die zweite Zuweisungsoperation verwendet wird. Wir erkennen hieran, wie wichtig es ist, dass Operatoren Ergebnisse zurückliefern, damit sie verkettbar sind.

Genau wie der Defaultkonstruktor, der Destruktor und der Copykonstruktor, wird für jede von uns entwickelte Klasse ein Zuweisungsoperator automatisch definiert, wenn wir dies nicht tun. Dieser erledigt prinzipiell ähnliche Arbeiten wie der Copykonstruktor:

- Für alle Datenmember mit Basisdatentypen wird der Wert des zuzuweisenden Datenmembers übertragen,
- für alle Datenmember mit Klassentypen wird der Zuweisungsoperator der jeweiligen Klasse mit dem zuzuweisenden Datenmember als Parameter aufgerufen.

In denselben Fällen, in denen wir gezwungen sind, einen eigenen Copykonstruktor und einen eigenen Destruktor anzugeben, sind wir auch veranlasst, einen Zuweisungsoperator zu schreiben. Dies führt zur *Dreierregel*: *Wenn eine Klasse einen eigenen Copykonstruktor benötigt, so benötigt sie stets auch einen eigenen Destruktor und einen eigenen Zuweisungsoperator.*

2.7.2 Ausgabeoperator (operator<<)

Der Ausgabeoperator dient der Ausgabe von Objekten der Klasse auf einem Ausgabekanal. Er wird stets als eigenständige Funktion implementiert, denn die Angabe als Memberfunktion ist aus technischen Gründen unmöglich. Betrachten wir dazu folgende Gegenüberstellung:

```
1          Kunde K1, K2;
2          // ...
3          K1 = K2;
4          cout << K1;
```

Wir sehen hier zunächst die Definition von zwei Objekten der Klasse `Kunde`. Danach folgt eine Zuweisung, wobei der Compiler selbige in folgender Form auffasst:

```
1          K1.operator=(K2);
```

Per Analogieschluss können wir feststellen, dass der Compiler die Ausgabe des Objektes `K1` auf dem Ausgabekanal `cout` folgendermaßen interpretiert:

```
1          cout.operator<<(K1);
```

Wie wir wissen, ist `cout` ein Objekt des Typs (der Klasse) `ostream` aus dem Namespace `std`. Da es sich dabei um den Namensraum der C++-Standardbibliothek handelt, können wir nicht einfach in der Klasse `ostream` eine Memberfunktion ergänzen. Vielmehr versucht der Compiler, sofern er keine Memberfunktion der oben angegebenen Form findet, eine stand-alone-Funktion der folgenden Form zu verwenden:

```
1          operator<<(cout, K1);
```

Copyright

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und andere eBooks können Sie rund um die Uhr und legal auf unserer Website

(http://www.informit.de)

herunterladen