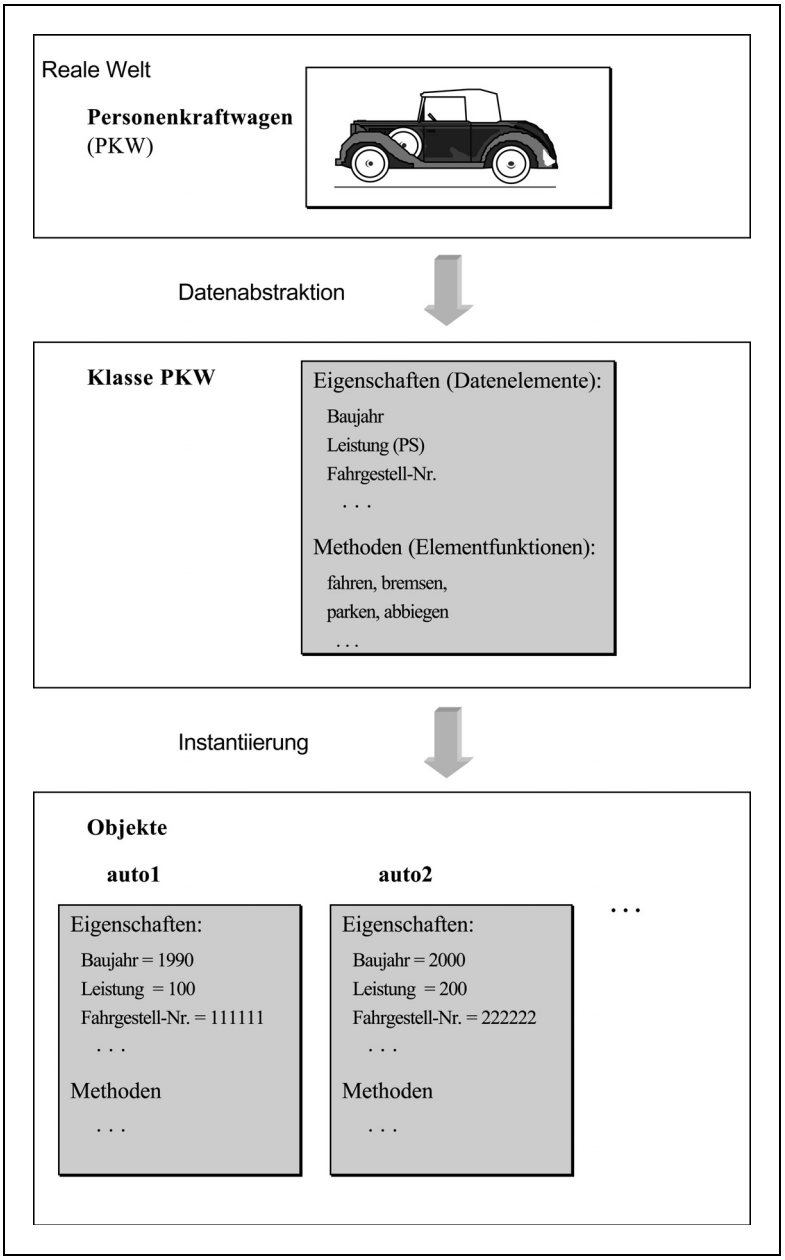
Kapitel 12

Definition von Klassen

Dieses Kapitel beschreibt, wie Klassen definiert und Instanzen von Klassen, also Objekte, verwendet werden. Daneben werden auch Structs und Unions als spezielle Klassen vorgestellt.

Klassen-Konzept



Klassen sind das wesentliche Sprachelement in C++ zur Unterstützung der objektorientierten Programmierung (OOP). Eine Klasse legt die Eigenschaften und Fähigkeiten von Objekten fest.

Datenabstraktion

Der Mensch *abstrahiert*, um komplexe Sachverhalte darzustellen. Dinge und Vorgänge werden auf das Wesentliche reduziert und mit einem Oberbegriff versehen. Klassen ermöglichen es, die Ergebnisse der Abstraktion direkter bei der Software-Entwicklung umzusetzen.

Der erste Schritt zur Lösung einer Problemstellung ist die *Analyse*. Bei der objektorientierten Programmierung besteht die Analyse in der Identifizierung und Beschreibung von Objekten sowie im Erkennen ihrer Beziehungen untereinander. Das Ergebnis der Beschreibung der Objekte sind Klassen.

In C++ ist eine Klasse ein selbstdefinierter Datentyp. Sie besteht aus Datenelementen, die die Eigenschaften beschreiben, und Elementfunktionen (= Methoden), die die Fähigkeiten der Objekte darstellen. Eine Klasse stellt das „Muster" dar, nach dem Objekte dieses Typs instantiiert, d.h. erzeugt, werden. Ein Objekt ist also eine Variable vom Typ einer bestimmten Klasse.

Datenkapselung

Bei der Definition einer Klasse wird festgelegt, welche Elemente der Klasse „privat", d.h. vor dem Zugriff von außen geschützt, sind und welche Elemente „öffentlich" verfügbar sein sollen. Ein Anwendungsprogramm arbeitet mit Objekten, für die es die „öffentlichen" Methoden der Klasse aufruft und so deren Fähigkeiten aktiviert.

Der Zugriff auf die Daten eines Objekts erfolgt selten direkt, d.h. Daten werden normalerweise als „privat" deklariert. Das Lesen und Verändern von Daten wird durch Methoden vorgenommen, die „öffentlich" deklariert sind und die den korrekten Zugriff auf die Daten sicherstellen können.

Ein wichtiger Aspekt dabei ist, dass ein Anwendungsprogramm die interne Darstellung der Daten nicht zu kennen braucht. Bei Bedarf kann die interne Darstellung der Daten auch geändert werden! Solange die Schnittstellen der öffentlichen Methoden unverändert bleiben, betreffen solche Änderungen das Anwendungsprogramm nicht. Es kann so von einer verbesserten Version einer Klasse profitieren, ohne dass im Anwendungsprogramm ein Byte verändert werden muss.

Ein Objekt kapselt also sein „Innenleben" von der Außenwelt ab und verwaltet sich mit Hilfe seiner Methoden selbst. Dies ist der konzeptionelle Kern der „Datenkapselung".

Definition von Klassen

Definitionsschema

```
class Demo
{
   private:

            // Hier die privaten Daten und Methoden

   public:

             // Hier die öffentlichen Daten und Methoden

};
```

Beispiel für eine Klasse

```
// konto.h
// Definition der Klasse Konto.
// ------------------------------------------------
#ifndef KONTO_H   // Mehrfaches Inkludieren verhindern.
#define KONTO_H
#include <iostream>
#include <string>
using namespace std;

class Konto
{
   private:              // Geschützte Elemente:
     string name;            // Kontoinhaber
     unsigned long nr;       // Kontonummer
     double stand;           // Kontostand

   public:               // Öffentliche Schnittstelle:
     bool init( const string&, unsigned long, double);
     void display();
};
#endif   //  KONTO_H
```

Die Definition einer Klasse legt den Namen der Klasse sowie die Namen und Typen der Klassenelemente (engl. *members*) fest.

Die Definition beginnt mit dem Schlüsselwort `class`, dem der Klassenname folgt. Im anschließenden Block werden die Elemente der Klassen deklariert: Datenelemente und Elementfunktionen dürfen einen beliebigen Typ haben, z. B. auch den Typ einer zuvor deklarierten Klasse. Zugleich werden die Elemente der Klasse aufgeteilt in:

- `private`-Elemente, die von außen nicht zugänglich sind
- `public`-Elemente, die öffentlich verfügbar sind

Die `public`-Elemente bilden die sog. *öffentliche Schnittstelle* (engl. *public interface*) der Klasse.

Nebenstehend ist ein Schema für die Definition einer Klasse angegeben. Der `private`-Bereich enthält in der Regel die Datenelemente, der `public`-Bereich die Methoden für den Zugriff auf die Daten. Auf diese Weise wird die Datenkapselung realisiert.

Im anschließenden Beispiel wird eine erste Klasse `Konto` zur Darstellung eines Bankkontos definiert. Die Datenelemente für den Namen des Kontoinhabers, die Kontonummer und den Kontostand sind wie üblich als `private` deklariert. Außerdem gibt es zwei öffentliche Methoden, nämlich `init()` zur Initialisierung und `display()` für die Anzeige der Daten auf dem Bildschirm.

Die Marken `private:` und `public:` können innerhalb einer Klasse beliebig eingesetzt werden:

- Sie dürfen beliebig oft, auch keinmal, und in beliebiger Reihenfolge verwendet werden. Dabei reicht der mit `private:` bzw. `public:` markierte Bereich bis zum nächsten `public:` bzw. `private:`.
- Die Voreinstellung für den Elementzugriff ist `private`. Wenn also weder eine `private`- noch eine `public`-Marke angegeben ist, so sind alle Elemente einer Klasse `private`.

Namensgebung

Jede Software wird mit bestimmten Regeln für die Vergabe von Namen erstellt. Diese orientieren sich oft am Zielsystem und den verwendeten Klassenbibliotheken. In diesem Buch wird eine übliche Konvention eingehalten, um Klassen und Klassenelemente zu unterscheiden: Die Namen von Klassen beginnen mit einem Großbuchstaben, die Namen von Elementen mit einem Kleinbuchstaben.

Elemente verschiedener Klassen dürfen die gleichen Namen haben. So kann beispielsweise auch eine andere Klasse ein Element `display()` besitzen.

Definition von Methoden

Die Methoden der Klasse Konto

```
// konto.cpp
// Definition der Konto-Methoden init() und display().
// ---------------------------------------------------
#include "konto.h"          // Definition der Klasse
#include <iostream>
#include <iomanip>
using namespace std;

// Die Methode init() kopiert die übergebenen Argumente
// in die privaten Elemente der Klasse.
bool Konto::init(const string& i_name,
                 unsigned long i_nr,
                 double        i_stand)
{
   if( i_name.size() < 1)     // Kein leerer Name
         return false;
   name  = i_name;
   nr    = i_nr;
   stand = i_stand;
   return true;
}

// Die Methode display() zeigt die privaten
// Daten am Bildschirm an.
void Konto::display()
{
   cout << fixed << setprecision(2)
        << "--------------------------------------\n"
        << "Kontoinhaber:  " << name  << '\n'
        << "Kontonummer:   " << nr    << '\n'
        << "Kontostand:    " << stand << '\n'
        << "--------------------------------------\n"
        << endl;
}
```

Eine Klassendefinition ist erst dann vollständig, wenn auch die Methoden definiert sind. Erst anschließend können Objekte dieser Klasse verwendet werden.

Bei der Definition einer Methode muss auch der Klassenname angegeben werden. Dieser wird vom Funktionsnamen durch den Bereichsoperator `::` getrennt.

Syntax:
```
typ klassenname::funktionsname(parameterliste)
{   . . . }
```

Ohne Angabe des Klasse würde eine gewöhnliche globale Funktion definiert.

Innerhalb einer Methode können *alle* Elemente einer Klasse direkt mit ihrem Namen angesprochen werden. Der Bezug zur Klasse ist automatisch gegeben. Insbesondere können Methoden derselben Klasse sich gegenseitig direkt aufrufen.

Der Zugriff auf die privaten Elemente ist nur innerhalb von Methoden derselben Klasse möglich. Damit stehen die `private`-Elemente vollständig unter der Kontrolle der Klasse.

Bei der Definition einer Klasse wird noch kein Speicherplatz für die Datenelemente reserviert. Das geschieht erst mit der Definition eines Objekts. Wird dann eine Methode für ein bestimmtes Objekt aufgerufen, so arbeitet die Methode mit den Daten dieses Objekts.

Modulare Programmierung

Eine Klasse wird gewöhnlich in verschiedenen Quelldateien verwendet. In einem solchen Fall ist es notwendig, die Definition einer Klasse in eine *Header-Datei* zu stellen. Ist beispielsweise die Definition der Klasse `Konto` in der Datei `konto.h` enthalten, so kann jede Quelldatei, die die Header-Datei inkludiert, mit der Klasse `Konto` arbeiten.

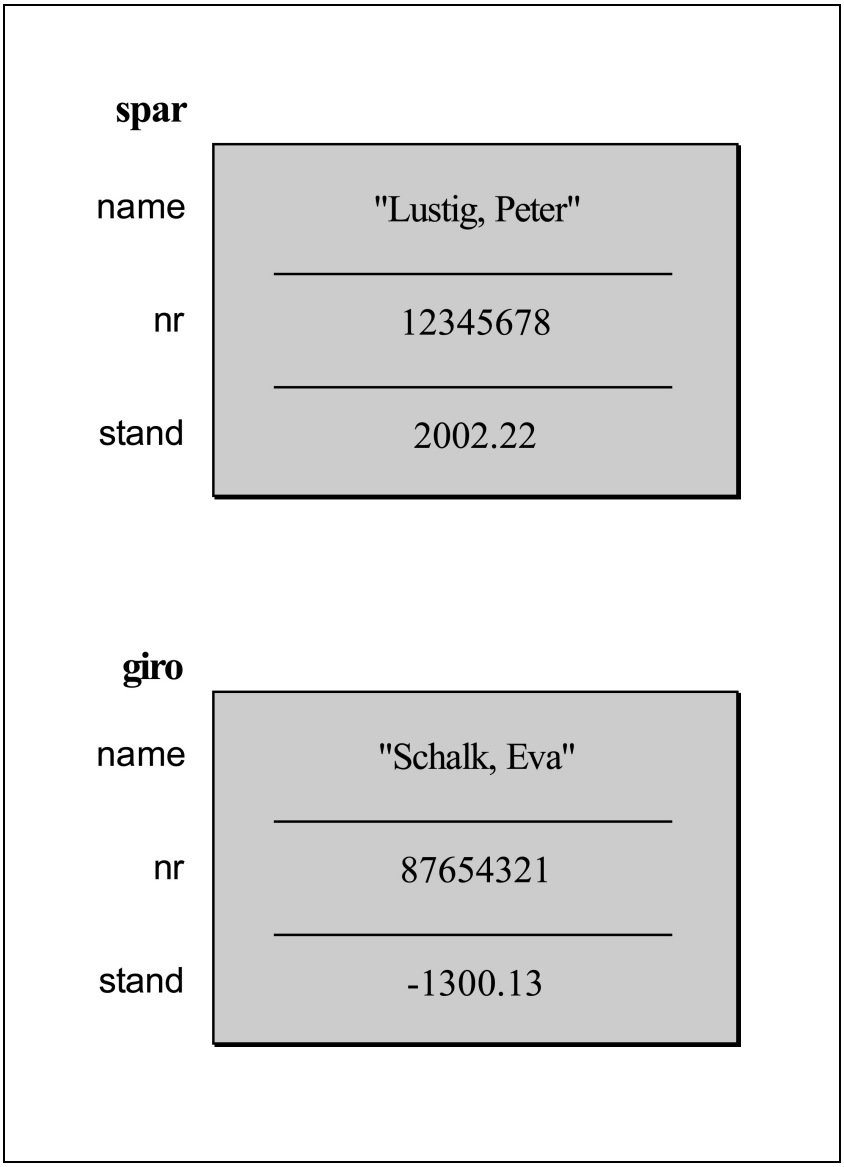
Die Definition der Methoden ist aber in jedem Fall in einer Quelldatei vorzunehmen. So sollten die Methoden der Klasse `Konto` beispielsweise in der Quelldatei `konto.cpp` definiert werden.

Der Quellcode des Anwenderprogramms, das z.B. die `main`-Funktion enthält, ist unabhängig von der Klasse und wird daher auch in separaten Quelldateien gespeichert. Die Trennung der Klassen vom Anwenderprogramm erleichtert die Wiederverwendbarkeit von Klassen.

In einer integrierten Entwicklungsumgebung erstellt der Programmierer ein *Projekt* zur Verwaltung der verschiedenen Module. Sämtliche Quelldateien werden in das Projekt eingefügt. Beim Kompilieren und Linken des Projekts werden veränderte Quelldateien automatisch neu kompiliert und mit dem Anwendungsprogramm zusammengebunden.

Definition von Objekten

Die Objekte giro und spar im Speicher



Mit einer Klasse wird ein neuer Datentyp definiert, für den Variablen, also Objekte, definiert werden können. Ein Objekt heißt auch *Instanz* einer Klasse.

Objekte definieren

Die Definition eines Objekts erfolgt wie üblich durch Angabe des Datentyps und des Objektnamens.

Syntax: `klassenname objektname1 [, objektname2,...]`

In der folgenden Definition wird ein Objekt `giro` vom Typ `Konto` erzeugt:

Beispiel:
```
Konto giro;     // oder: class Konto giro;
```

Für die Datenelemente des Objekts `giro` wird jetzt der entsprechende Speicherplatz reserviert. Das Objekt `giro` besteht selbst aus den Teilobjekten `name`, `nr` und `stand`.

Objekte im Speicher

Werden mehrere Objekte vom Typ derselben Klasse deklariert, etwa

Beispiel:
```
Konto giro, spar;
```

so besitzt jedes Objekt seine eigenen Datenelemente. Auch das Objekt `spar` hat die Datenelemente `name`, `nr` und `stand`. Diese belegen allerdings im Hauptspeicher einen anderen Speicherplatz als die von `giro`.

Für beide Objekte werden jedoch dieselben Methoden aufgerufen. Der Maschinencode der Methoden ist nur einmal im Speicher abgelegt, und zwar auch dann, wenn noch kein Objekt der Klasse deklariert wurde.

Eine Methode wird stets für ein bestimmtes Objekt aufgerufen. Sie arbeitet dann mit den Datenelementen *dieses* Objekts. So ergibt sich der nebenstehende Speicherinhalt, wenn für jedes Objekt die Methode `init()` mit den entsprechenden Werten aufgerufen wurde.

Initialisierung von Objekten

Die Objekte der Klasse `Konto` wurden bisher ohne eine explizite Initialisierung definiert. Daher wird auch jedes Teilobjekt ohne eine explizite Initialisierung erzeugt. Der String `name` ist damit leer, da dies in der Klasse `string` so festgelegt ist. Der Anfangswert der beiden Datenelemente `nr` und `stand` ist dagegen undefiniert. Wie bei anderen Variablen auch ist der Inhalt dieser Datenelemente `0`, wenn das Objekt global oder als `static` definiert wird.

Für Objekte kann genau festgelegt werden, wie diese „aufgebaut“ oder „abgebaut“ werden. Diese Aufgaben erledigen *Konstruktoren* und *Destruktoren*. Konstruktoren übernehmen insbesondere die Initialisierung von Objekten. Dazu erfahren Sie später mehr.

Verwendung von Objekten

Beispielprogramm

```
// konto_t.cpp
// Mit Objekten der Klasse Konto arbeiten.
// -----------------------------------------------

#include "konto.h"

int main()
{
   Konto giro1, giro2;

   giro1.init("Munter, Gabi", 3512345, 99.40);
   giro1.display();

//   giro1.stand += 100;   // Fehler: stand ist private

   giro2 = giro1;        // ok: Zuweisung von Objekten
                         // möglich.
   giro2.display();      // ok

                         // Neue Werte für giro2
   giro2.init("Liebig, Ernst", 3512345, 199.40);

   giro2.display();
                           // Referenz verwenden:
   Konto& munter = giro1;  // munter ist Aliasname
                           // für Objekt giro1.
   munter.display();   // munter kann wie das
                       // Objekt giro1 verwendet werden.
   return 0;
}
```

Punktoperator

Ein Anwendungsprogramm, das mit Objekten einer Klasse arbeitet, kann nur auf die `public`-Elemente des Objekts zugreifen. Dies geschieht mit Hilfe des *Punktoperators*.

Syntax: `objekt.element`

Dabei ist `element` ein Datenelement oder eine Methode.

Beispiel:
```
Konto giro;
giro.init("Lustig, Peter",1234567,-1200.99);
```

Hierbei stellt der Ausdruck `giro.init` die `public`-Methode `init` der Klasse `Konto` dar. Diese wird mit drei Argumenten für das Objekt `giro` aufgerufen.

Der Aufruf von `init()` kann nicht durch direkte Zuweisungen ersetzt werden.

Beispiel:
```
giro.name  = "Lustig, Peter"; // Fehler, da
giro.nr    = 1234567;         // Datenelemente
giro.stand = -1200.99;        // private.
```

Der Zugriff auf die `private`-Elemente eines Objekts ist außerhalb der Klasse nicht zulässig. Deshalb ist es auch nicht möglich, einzelne Datenelemente der Klasse `Konto` auf dem Bildschirm anzuzeigen.

Beispiel:
```
cout << giro.stand;          // Fehler
giro.display();              // ok
```

Die Methode `display()` zeigt alle Datenelemente von `giro` an. Eine Methode wie `display()` kann immer nur für ein Objekt aufgerufen werden. Die Anweisung

```
display();
```

würde zu einer Fehlermeldung führen, da es eine globale Funktion `display()` nicht gibt. Welche Daten sollten auch angezeigt werden?

Zuweisung von Objekten

Die Zuweisung `=` ist der einzige Operator, der standardmäßig für jede Klasse definiert ist. Dabei müssen Quell- und Zielobjekt zur selben Klasse gehören. Die Zuweisung erfolgt so, dass die einzelnen Datenelemente des Quellobjekts dem entsprechenden Element des Zielobjekts zugewiesen werden.

Beispiel:
```
Konto giro1, giro2;
giro2.init("Reich, Franzi",350123, 10000.0);
giro1 = giro2;
```

Hier werden die Datenelemente von `giro2` in die entsprechenden Datenelemente von `giro1` kopiert.

Zeiger auf Objekte

Beispielprogramm

```
// ptrObj.cpp
// Mit Zeigern auf Objekte der Klasse Konto arbeiten.
// ------------------------------------------------
#include "konto.h"   // Inkludiert <iostream>, <string>
bool getKonto( Konto *pKonto);            // Prototyp
int main()
{
   Konto giro1, giro2, *ptr = &giro1;

   ptr->init("Munter, Gabi",       // giro1.init(...)
             3512345, 99.40);
   ptr->display();                 // giro1.display()

   ptr = &giro2;         // ptr auf giro2 zeigen lassen.
   if( getKonto( ptr))   // Neue Kontodaten einlesen
      ptr->display();    // und anzeigen.
   else
      cout << "Ungültige Eingabe!" << endl;
   return 0;
}
// ------------------------------------------------
// getKonto() liest von der Tastatur die Daten für
// ein neues Konto ein, das per Zeiger übergeben wird.
bool getKonto( Konto *pKonto)
{
   string name, line(50,'-');  // Lokale Hilfsvariablen
   unsigned long nr;
   double startkapital;

   cout << line << '\n'
        << "Daten für ein neues Konto eingeben: \n"
        << "Kontoinhaber: ";
   if( !getline(cin,name) || name.size() == 0)
     return false;
   cout << "Kontonummer:  ";
   if( !(cin >> nr))           return false;
   cout << "Startkapital: ";
   if( !(cin >> startkapital)) return false;
   // Alle Eingaben ok
   pKonto->init( name, nr, startkapital);
   return true;
}
```

Ein Objekt vom Typ einer Klasse hat – wie jedes andere Objekt auch – eine Adresse im Hauptspeicher. Diese kann einem passend deklarierten Zeiger zugewiesen werden.

Beispiel:
```
Konto spar("Bill, Claudia",654321, 123.5);
Konto *ptrKonto = &spar;
```

Hier werden ein Objekt `spar` und eine Zeigervariable `ptrKonto` definiert. Der Zeiger `ptrKonto` wird so initialisiert, dass er auf das Objekt `spar` zeigt. Damit ist `*ptrKonto` das Objekt `spar` selbst, und mit

Beispiel:
```
(*ptrKonto).display();
```

wird die Methode `display()` für das Objekt `spar` aufgerufen. Die Klammern sind hier notwendig, da der Punktoperator einen höheren Vorrang als der Operator `*` hat.

Pfeiloperator

Statt der Kombination der Operatoren `*` und `.` kann auch einfach der Pfeiloperator `->` verwendet werden.

Syntax:
```
objektZeiger->element
```

Dieser Ausdruck ist äquivalent zu:

```
(*objektZeiger).element
```

Der Pfeiloperator besteht aus dem Minus-Zeichen und dem Größer-Zeichen.

Beispiel:
```
ptrKonto->display();
```

Diese Anweisung ruft die Methode `display()` für das Objekt auf, das durch `ptrKonto` adressiert wird, also für das Objekt `spar`. Sie ist gleichbedeutend mit der Anweisung im letzten Beispiel.

Der Unterschied zwischen den Operatoren `.` und `->` ist also, dass der linke Operand des Punktoperators ein Objekt sein muss, der linke Operand des Pfeiloperators dagegen ein Zeiger auf ein Objekt.

Zum Beispielprogramm

Zeiger auf Objekte werden häufig als Parameter von Funktionen eingesetzt. Eine Funktion, die die Adresse eines Objekts als Argument erhält, kann direkt mit dem Objekt des Aufrufers arbeiten. Dies demonstriert das nebenstehende Programm. Es liest mit Hilfe der Funktion `getKonto()` die Daten für ein neues Konto ein. Beim Aufruf wird die Adresse des Kontos übergeben:

```
getKonto( ptr)  // oder:  getKonto( &giro1)
```

Mit dem übergebenen Zeiger und der Methode `init()` kann die Funktion dann die neuen Daten in das Objekt des Aufrufers schreiben.

Structs

Beispielprogramm

```
// structs.cpp
// Eine Struktur definieren und verwenden.
// ------------------------------------------------
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
struct Vertreter   // Definition der Struktur Vertreter
{
  string name;          // Name des Vertreters.
  double umsatz;        // Sein Umsatz pro Monat.
};
inline void print( const Vertreter& v)
{
   cout << fixed << setprecision(2)
        << left  << setw(20) << v.name
        << right << setw(10) << v.umsatz << endl;
}
int main()
{
   Vertreter heidi, hannes;

   heidi.name    = "Sturm, Heidi";
   heidi.umsatz  = 37000.37;
   hannes.name   = "Forsch, Hannes";
   hannes.umsatz = 23001.23;

   heidi.umsatz += 1700.11;          // Mehr Umsatz

   cout << "  Vertreter            Umsatz\n"
        << "-------------------------------" << endl;
   print( heidi);
   print( hannes);
   cout << "\nSumme der Umsätze: "
        << heidi.umsatz + hannes.umsatz << endl;
   Vertreter *ptr = &hannes;         // Zeiger ptr.
                                     // Wer macht die
   if( hannes.umsatz < heidi.umsatz) // meisten Umsätze?
     ptr = &heidi;
   cout << "\nUnser bestes Pferd im Stall: "
        << ptr->name << endl;   // Name des Vertreters,
                                // auf den ptr zeigt.
   return 0;
}
```

Datensätze

In einer klassischen, prozeduralen Sprache wie C werden verschiedene Daten, die logisch zusammengehören, zu einem Datensatz (engl. *record*) zusammengefasst. Umfangreiche Daten, wie beispielsweise die Daten von Artikeln im Lager eines Autoherstellers, können so übersichtlich organisiert und in Dateien gespeichert werden.

Aus der Sicht einer objektorientierten Sprache ist ein Datensatz nichts anderes als eine Klasse, die nur öffentliche Datenelemente und keine Methoden enthält. Entsprechend kann in C++ auch mit dem Schlüsselwort `class` die Struktur eines Datensatzes festgelegt werden.

Beispiel:
```
class Datum
{ public:   short  tag, monat, jahr; };
```

Es ist jedoch üblich, reine Datensätze mit dem schon in C bekannten Schlüsselwort `struct` zu definieren. So ist die obenstehende Definition von `Datum` mit den Elementen `tag, monat` und `jahr` äquivalent zu:

Beispiel:
```
struct Datum { short  tag, monat, jahr; };
```

Die Schlüsselworte class und struct

Jede Klasse kann auch mit dem Schlüsselwort `struct` definiert werden, wie z.B. die Klasse `Konto`.

Beispiel:
```
struct Konto {
   private:     //  . . .   wie gehabt
   public:      //  . . .
};
```

Die Schlüsselworte `class` und `struct` unterscheiden sich nur in bezug auf die Datenkapselung:

- Die Voreinstellung für den Elementzugriff einer mit `struct` definierten Klasse ist `public`.

Im Gegensatz zu einer mit `class` definierten Klasse sind also alle Elemente `public`, wenn keine `private`-Marke angegeben ist. Durch diese Festlegung bleibt die Kompatibilität zu C gewahrt.

Beispiel:
```
Datum zukunft;
zukunft.jahr = 2100;     // ok! Daten public
```

Reine Datensätze, d.h. Objekte einer Klasse, die nur `public`-Datenelemente enthält, können bei der Definition durch eine Liste initialisiert werden.

Beispiel:
```
Datum birthday = { 29, 1, 1977};
```

Dabei initialisiert das erste Element der Liste das erste Element im Objekt usw.

Structured Bindings

Beispielprogramm

```
// strucbind.cpp
// Eine Struktur entpacken.
// ------------------------------------------------
#include <iostream>
#include <string>
using namespace std;

struct Vertreter { . . .} // wie zuvor

inline Vertreter& incUmsatz(Vertreter& v, double inc)
{
    auto&[nref, uref] = v;   // Struktur v entpacken.
    uref += inc;             // Umsatz erhöhen.
    return v;
}

int main()
{
    Vertreter v = {"Sturm, Heidi", 4950.99 };

    auto[nm, um] = v;          // Struktur  entpacken.
    cout << "\nDie entpackte Struktur: " << endl;
    cout << "Name: " << nm << "\nUmsatz: " << um << endl;

    // Referenzen auf die Datenelemente der Struktur:
    auto&[nref, uref] = v;

    nref = "Winter, Heidi";    // Name ändern

                               // Umsatz erhöhen:
    auto[r1, r2] = incUmsatz(v, 5000.0);

    cout << "\nAenderungen in der entpackten Struktur: ";
    cout << "\nName: " << r1 << "\nUmsatz: " << r2 << endl;

    cout << "\nDie geaenderte Struktur: " << endl;
    cout << "Name: " << v.name
         << "\nUmsatz: " << v.umsatz << endl;
    return 0;
}
```

Hinweis: In Visual C++ muss mindestens die Compileroption /std:c++17 aktiviert sein.

Gleichzeitige Initialisierung

C++ unterstützt die gleichzeitige Initialisierung mehrerer Variablen mithilfe einer Struktur. Sind die Struktur `Triple` und ein Objekt `ts` dieses Typs wie folgt definiert:

Beispiel:
```
struct Triple {char c, int n, double x };
Triple ts = {'a', 2, 3.1};
```

so bewirkt die Anweisung:

```
auto[u, v, w] = ts;
```

dass die Variablen `u`, `v` und `w` erzeugt und initialisiert werden. Die Variable `u` erhält das Zeichen `'a'` und die Variablen `v`, `w` erhalten die Werte `2` bzw. `3.1`.

Der Compiler nimmt dabei eine automatische Typableitung vor. Die Variable `u` hat dann den Typ `char` und die Variablen `v` bzw. `w` haben den Typ `int` bzw. `double`. Dieser Vorgang heißt auch *Structured Bindings*.

Werden die zu initialisierenden Variablen als Referenzen deklariert, so können die Datenelemente einer Struktur auch verändert werden, wie das folgende Beispiel zeigt.

Beispiel:
```
auto&[uref, vref, wref] = ts;
uref = 'b';
```

Hier wird das erste Datenelement der Struktur `Triple` mit dem Zeichen `'b'` überschrieben.

Strukturen entpacken

Die Technik des Structured Bindings wird normalerweise eingesetzt, um mehrere Return-Werte einfach zu handhaben. Die verschiedenen Werte werden in eine Struktur „gepackt“, die von einer Funktion bearbeitet und als Return-Wert geliefert wird. Anschließend kann die Struktur wieder „entpackt“ werden.

Dies demonstriert auch das nebenstehende Beispiel. Die Funktion `incUmsatz()` erhöht den Umsatz eines als Argument übergebenen Vertreters um einen bestimmten Betrag, der mit dem zweiten Argument übergeben wird. In der Definition der Funktion werden mit der Anweisung

Beispiel:
```
auto&[nref, uref] = v;
```

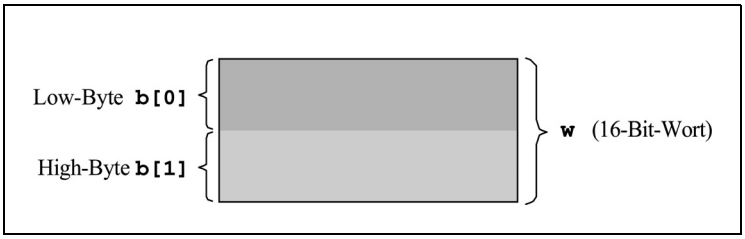
zwei Referenzen auf die Datenelemente der Struktur `Vertreter` gesetzt (`nref` und `uref`). Dann wird der Umsatz aktualisiert. Mit dem Aufruf

Beispiel:
```
auto[r1, r2] = incUmsatz(v, 5000.0);
```

wird die von `incUmsatz()` zurückgelieferte Struktur wieder entpackt.

Unions

Ein Objekt der Union WordByte im Speicher



Definition und Verwendung der Union WordByte

```
// unions.cpp
// Eine Union definieren und verwenden.
// ----------------------------------------------------
#include <iostream>
using namespace std;

union WordByte
{
  private:
    unsigned short w;     // 16-Bit
    unsigned char b[2];   // Zwei Byte: b[0], b[1]
  public:                 // Wort und Byte-Zugriff:
    unsigned short& word()    { return w; }
    unsigned char&  lowByte() { return b[0]; }
    unsigned char&  highByte(){ return b[1]; }
};

int main()
{
    WordByte wb;
    wb.word() = 256;
    cout << "\nWord:      " << (int)wb.word();
    cout << "\nLow-Byte:  " << (int)wb.lowByte()
         << "\nHigh-Byte: " << (int)wb.highByte()
         << endl;
    return 0;
}
```

Ausgabe des Programms:

```
Word:      256
Low-Byte:  0
High-Byte: 1
```

Nutzung von Speicherplatz

Bei einer gewöhnlichen Klasse belegt jedes Datenelement eines Objekts seinen eigenen, separaten Speicherplatz. Eine *Union* ist dagegen eine Klasse, deren Datenelemente auf demselben Speicherplatz abgelegt werden. Jedes Datenelement hat also dieselbe Anfangsadresse im Speicher. Natürlich kann eine Union nicht *gleichzeitig* verschiedene Daten auf demselben Platz speichern. Aber eine Union erlaubt, dieselbe Speicherstelle verschiedenartig zu nutzen.

Definition

Syntaktisch unterscheidet sich eine Union von einer mit `class` oder `struct` definierten Klasse nur durch das Schlüsselwort `union`.

Beispiel:
```
union Zahl
{
  long   n;
  double x;
};
Zahl zahl1, zahl2;
```

Hier werden eine Union `Zahl` und zwei Objekte dieses Typs definiert. In der Union `Zahl` kann entweder eine ganze Zahl oder eine Gleitpunktzahl gespeichert werden.

Ohne die Angabe einer `private`-Marke sind alle Elemente einer Union automatisch `public`. Dies entspricht der Voreinstellung für Strukturen. Daher kann im Fall der Union `Zahl` direkt mit den Elementen `n` und `x` gearbeitet werden.

Beispiel:
```
zahl1.n = 12345;   // Ganze Zahl speichern,
zahl1.n *= 3;      // mit 3 multiplizieren.
zahl2.x = 2.77;    // Gleitpunktzahl!
```

Der Programmierer ist selbst dafür verantwortlich, den aktuellen Inhalt richtig zu interpretieren. Meist wird dafür ein zusätzliches „Typfeld“ angelegt, das den gespeicherten Inhalt kennzeichnet.

Die Größe eines Objekts vom Typ einer Union wird durch das längste Datenelement bestimmt, da ja alle Datenelemente bei derselben Anfangsadresse beginnen. Im Beispiel der Union `Zahl` ist dies die Größe des `double`-Elements, also `8 == sizeof(double)` Byte.

Das nebenstehende Beispiel definiert eine Union `WordByte`, die es erlaubt, eine 16-Bit-Speicherstelle als ganzes oder byteweise zu lesen bzw. zu beschreiben.