Modellierung und Programmierung

Dr. Martin Riplinger

12.12.2012

Strukturen: Motivation

Situation: Mit Funktionen verfügen wir über ein wirksames Mittel, um **Programmcode** sinnvoll zu gliedern und Komplexität zu verbergen. Für **Daten** haben wir bisher kein solches Konstrukt!

Beispielproblem: Adressdatenbank

Eine Adresse (in Deutschland) besteht aus

Straße	Zeichenkette
Hausnummer	positive Ganzzahl (Vorsicht: „21a“ nicht darstellbar!)
Stockwerk	positive Ganzzahl (optional)
Postleitzahl	positive Ganzzahl
Ort	Zeichenkette

Umsetzung: jede Informationseinheit wird in einem entsprechenden Feld gespeichert, und die Einträge mit gleichen Feldindices gehören zueinander.

```
char **strassen, unsigned *hausnummern, unsigned *stockwerke, unsigned
*postleitzahlen, char **orte
```

Nachteile:

- Unlogische Gruppierung von Daten, die miteinander nichts zu tun haben
- Zuordnung über den Feldindex ist fehleranfällig

Strukturen

Bessere Lösung: jede Adresse ist wie eine Variable ansprechbar, deren Elemente die einzelnen Informationseinheiten sind. Dazu gibt es in C das Konzept der **Strukturen**.

Definition

Eine Struktur ist die Zusammenfassung einer bestimmten Anzahl von Daten (möglicherweise) **verschiedenen Typs** zu einer Einheit, die mit einem festgelegten Namen angesprochen werden kann.

Deklaration

- eines Struktur**typs**: `struct Etikett;`

  Teilt dem Compiler mit, dass es einen Strukturtyp mit einem bestimmten Etikett (engl. *structure tag*) gibt. Muss **außerhalb von** `main` stehen.

- einer Struktur**variablen**: `struct Etikett Variablenname;`

  Teilt dem Compiler mit, dass eine Variable eines bestimmten Namens gibt, die vom Typ `struct Etikett` ist.

Definition eines Strukturtyps: `struct Etikett {Codeblock};`

Legt fest, welche Komponenten (engl. *members*) zu einer Struktur dieses Typs gehören. Muss **außerhalb von** `main` stehen (meistens direkt nach oder sogar statt der Deklaration).

Strukturen

Anwendung auf das Beispielproblem:

```
1  struct Adresse;       // Deklaration des Strukturtyps "Adresse"
2
3  struct Adresse        // Definition der Struktur
4  {
5    char *strasse;
6    unsigned hausnr;
7    int stock;          // 0 = "keine Angabe"
8    unsigned plz;
9    char *ort;
10 };
11
12 int main(void)
13 {
14   // Deklaration einer Variablen vom Typ "struct Adresse"
15   struct Adresse meine_adr;
16   printf("sizeof(struct Adresse) = %lu\n", sizeof(struct Adresse));
17   printf("alternativ: sizeof(meine_adr) = %lu\n", sizeof(meine_adr));
18   return 0;
19 }
```

- Die Definition macht in diesem Beispiel die Deklaration überflüssig.
- Anzahl und Namen der Komponenten der Struktur sind mit der Definition festgelegt und im Nachhinein nicht mehr veränderbar.
- Hinter der schließenden geschweiften Klammer muss ein Semikolon stehen!
- „`struct Adresse`“ kann wie ein gewöhnlicher Datentyp behandelt werden.

Strukturen: Zugriff auf Komponenten

Die Komponenten einer Struktur werden mit ihrem **Namen** angesprochen, im Gegensatz zum Zugriff über Indices bei Feldern. Das Pendant zum Index-Operator „[]“ ist der Strukturkomponenten-Operator „.“.

- **Verwendung**: `Strukturvariable.Komponentenname`

  Greift auf die Komponente des entsprechenden Namens einer zuvor deklarierten Strukturvariable zu.
- Der .-Operator hat wie [] höchste Priorität, insbesondere höher als *, &, ++, Cast und arithmetische Operatoren.

Beispiel:

```
meine_adr.strasse = "Stuhlsatzenhausweg";
meine_adr.hausnr = 103;
meine_adr.stock = 0;
meine_adr.plz = 66123;
meine_adr.ort = "Saarbruecken";

printf("%s %u\n", meine_adr.strasse,
                  meine_adr.hausnr);
printf("%u %s\n", meine_adr.plz,
                  meine_adr.ort);
```

Ausgabe:

```
Stuhlsatzenhausweg 103
66123 Saarbruecken
```

Strukturen: Weitere Eigenschaften

- **Initialisierung bei Deklaration**: wie bei statischen Feldern in geschweiften Klammern

```
struct Adresse meine_adr = {"Stuhlsatzenhausweg", 103, 0, 66123,
                            "Saarbruecken"};
```

- Strukturen können andere Strukturen enthalten (**Schachtelung**)

```
struct Anschrift
{
  char *nachname;
  char *vorname;
  struct Adresse adresse;
};
```

Initialisierung:

```
struct Anschrift meine_anschrift = {"mit Biergarten",
  "Restaurant",
  {"Stuhlsatzenhausweg", 103, 0, 66123, "Saarbruecken"} };
```

Zugriff auf geschachtelte Strukturen: z. B. `meine_anschrift.adresse.hausnr`

Achtung: Zu tiefe Schachtelung macht Code unlesbar!

Strukturen: Weitere Eigenschaften

- **Zeiger auf Strukturen**: `struct Etikett *pointer`

  Deklariert einen Zeiger auf den Datentyp `struct Etikett`

  Für den Zugriff `(*pointer).Komponente` auf Komponenten mit Hilfe des Pointers gibt es die vereinfachte Schreibweise `pointer->Komponente`

- Strukturen können einen **Zeiger auf den eigenen Typ** enthalten.Dazu müssen jedoch Deklaration und Definition getrennt werden:

```
struct Anschrift;

struct Anschrift
{
  char *nachname;
  char *vorname;
  struct Adresse adresse;
  struct Anschrift *li_nachbar;
  struct Anschrift *re_nachbar;
} meine_anschrift_;
```

Bemerkung: In diesem Programmcode wird in Verbindung mit der Definition der Struktur `Anschrift` eine **globale Instanz** `meine_anschrift_` deklariert.

Zugriff auf Nachbarelement: z. B. `meine_anschrift_.li_nachbar->hausnr`

Anwendung: Verkettung von Daten, z. B. Listen oder Bäume

Strukturen: Weitere Eigenschaften

- **Zeiger** auf Strukturen können an Funktionen als Parameter übergeben und als Rückgabewert zurückgegeben werden.

- Strukturen, die andere Strukturen oder dynamische Speicherobjekte enthalten, sollten nicht mehr „von Hand“, sondern mit speziell dafür vorgesehenen Funktionen (Erstellung, Manipulation von Komponenten, Kopie, Löschung, . . .) verwaltet werden
  $\rightsquigarrow$ Robuste Programme, Anfänge der „Objektorientierten Programmierung“

- **Vorsicht** Strukturen können nur komponentenweise verglichen werden. Der Vergleich

```
struct Vektor u,v;
if (u==v)
  printf("Vektoren stimmen überein");
```

führt zu der Fehlermeldung: `invalid operands to binary ==`

Strukturen: Weitere Eigenschaften

Vorsicht: Bei Zuweisung wird **nur die oberste Ebene** betrachtet.

```
1  struct Vektor
2  {
3    int dim;
4    char *label;
5    int *koord;
6  };
7
8  int main(void)
9  {
10   struct Vektor u = {2, "Ursprung", NULL}, e1;
11   u.koord = (int *) malloc(u.dim * sizeof(int));
12
13   e1 = u;       // Shallow copy
14   e1.label = "1. Einheitsvektor";
15   //umbiegen des Zeigers e1.label
16   e1.koord[0] = 1;
17
18   printf("%s u = (%d,%d), ", u.label, u.koord[0], u.koord[1]);
19   printf("%s e1 = (%d,%d)\n", e1.label, e1.koord[0], e1.koord[1]);
20
21   free(u.koord);
22   return 0;
23 }
```

Ausgabe: `Ursprung u = (1,0), 1. Einheitsvektor e1 = (1,0)`

⇝ Es wurden sämtliche **Variablenwerte** (= Adressen bei Zeigern!) kopiert, nicht jedoch das dynamische Feld! (Woher soll der Compiler auch davon wissen?)

Typendefinition

Mit Hilfe des Schlüsselworts `typedef` lassen sich in C eigene Datentypen definieren.

- **Syntax**: `typedef AlterDatentyp NeuerDatentyp;`
- **Beispiel 1**: primitive Datentypen

  ```
  typedef unsigned long int size_t;
  ```

  Anwendung: architekturunabhängige Programmierung
- **Beispiel 2**: in Verbindung mit Strukturen

  ```
  typedef struct Adresse Adresse;
  ```

  Im folgenden Code kann statt „`struct Adresse`“ einfach „`Adresse`“ geschrieben werden. Folge: erhöhte Lesbarkeit
- Die Typendefinition lässt sich mit der Strukturdefinition verbinden:

```
typedef struct _Adresse_ Adresse;

struct _Adresse_
{
  char *strasse;
  unsigned hausnr;
  int stock;
  unsigned plz;
  char *ort;
};
```

oder äquivalent:

```
typedef struct
{
  char *strasse;
  unsigned hausnr;
  int stock;
  unsigned plz;
  char *ort;
} Adresse;
```

Beispiel: Erstellen/Kopieren eines Vektors

```
1  Vektor *neuerVektor(int dim, char *label, int *koord)
2  { Vektor *vec=(Vektor *) malloc(sizeof(Vektor));int i;
3    if(vec==NULL)
4         printf("Zu wenig speicher");
5    vec->dim = dim;
6    vec->koord = (int *) malloc(dim*sizeof(int));
7    for (i=0;i<dim;i++)
8                vec->koord[i] = koord[i];
9    vec->label = dupliziereString(label);
10   return vec;}
11
12 char *dupliziereString(char *string)
13 {char *clone = NULL;
14  if (string)
15  {
16    if ( !(clone = (char *) malloc(sizeof(char)*(strlen(string)+1))))
17      printf("Zu wenig Speicher!");
18    strcpy(clone,string);
19  }
20  return clone;}
```

Ausschnitt aus dem Hauptprogramm:

```
int dimension=2;
koord=(int*) malloc(dimension*sizeof(int));
char ulabel[]="Ursprung";
struct Vektor *u=neuerVektor(dimension,ulabel,koord); //neuen Vektor erzeugen
struct Vektor *e1=neuerVektor(u->dim,u->label,u->koord); //Vektor kopieren
```

Beispiel: komplexe Zahlen

Deklarationen und Definitionen

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  typedef struct
6  {
7    double re;
8    double im;
9  } Complex;
10
11 Complex *newComplexPolar(double radius, double angle);
12 Complex *complexProduct(Complex *z1, Complex *z2);
13 //-------------------------------------------------------------------------
14
15 Complex *newComplexPolar(double radius, double winkel)
16 {
17   Complex *z = (Complex *) malloc(sizeof(Complex));
18   z->re = radius * cos(winkel);
19   z->im = radius * sin(winkel);
20   return z;
21 }
22 //-------------------------------------------------------------------------
23 Complex *complexProduct(Complex *z1, Complex *z2)
24 {
25   Complex *z = (Complex *) malloc(sizeof(Complex));
26   z->re = z1->re*z2->re - z1->im*z2->im;
27   z->im = z1->re*z2->im + z1->im*z2->re;
28   return z;
29 }
```

Beispiel: komplexe Zahlen

Hauptprogramm

```
1  int main(void)
2  {
3    Complex a = {1.0, 1.0};
4    Complex *b = newComplexPolar(2.0, 3*M_PI/4.0);
5    Complex *c = complexProduct(&a, b);
6
7    printf("a  = % lf%+lfi\n", a.re, a.im);
8    printf("b  = % lf%+lfi\n", b->re, b->im);
9    printf("a*b = % lf%+lfi\n", c->re, c->im);
10   c->re = 0;
11   c->im = 2.5;
12   b = complexProduct(&a, c);
13   printf("a*c = % lf%+lfi\n", b->re, b->im);
14
15   return 0;
16 }
```

Ausgabe:

```
a   =  1.000000+1.000000i
b   = -1.414214+1.414214i
a*b = -2.828427+0.000000i
a*c = -2.500000+2.500000i
```

Felder von Strukturen

```
1  int main(void)
2  {
3    int j, n = 10;
4    Complex *dyn_vek = (Complex *) calloc(n, sizeof(Complex));
5    Complex stat_vek[10];
6
7    for (j=0; j<n; j++)
8    {
9      dyn_vek[j].re = 2.0 * j;
10     dyn_vek[j].im = 0.5 * j;
11     stat_vek[j].re = 2.0 * j;
12     stat_vek[j].im = 0.5 * j;
13   }
14   printf(" dyn_vek[7] = % lf%+lfi\n", dyn_vek[7].re, dyn_vek[7].im);
15   printf("stat_vek[7] = % lf%+lfi\n", stat_vek[7].re, stat_vek[7].im);
16
17   free(dyn_vek);
18   return 0;
19 }
```

Ausgabe:

```
 dyn_vek[7] =  14.000000+3.500000i
stat_vek[7] =  14.000000+3.500000i
```

⇝ Statische und dynamische Felder von Strukturen können wie gewohnt verwaltet werden.

Allgemeine Merkregeln für Strukturen

- Strukturen dienen dazu, Einzelinformationen zu einer **sinnvollen Einheit** zusammenzufassen.
- Häufig sind Strukturen Abbilder von Konzepten aus dem Alltag oder von mathematischen Objekten (siehe Beispiele zu Adressen und komplexen Zahlen).
- Geschachtelte Strukturen sind möglich und oft sinnvoll, bergen aber die Gefahr, zu große Komplexität zu erzeugen und das Ziel größerer Ordnung zu verfehlen. Zudem steigt das Risiko von Programmierfehlern (s. *shallow-copy*-Problem).
- Strukturen, die andere Strukturen oder dynamische Speicherobjekte enthalten, sollten nicht mehr „von Hand“, sondern mit speziell dafür vorgesehenen Funktionen verwaltet werden (Erstellung, Manipulation von Komponenten, Kopie, Löschung, ...) $\rightsquigarrow$ Anfänge der „Objektorientierten Programmierung“
- Zur Erhöhung der Lesbarkeit sollten Strukturen **immer** mit einem `typedef` als neuer Datentyp definiert werden.
- Zur Namensgebung hat sich folgende Konvention durchgesetzt:

  `ErsterBuchstabeGrossOhneUnterstriche` für eigene `typedef`-Datentypen

  `_VorneUndHintenMitUnterstrichen_` für Strukturen, die nicht direkt, sondern nur in Verbindung mit `typedef` verwended werden.

  Beispiel:
  ```
  typedef struct _Adresse_ Adresse;
  ```
  Verwendung nur über `Adresse`, niemals über `struct _Adresse_`

Funktionszeiger: Vorüberlegungen

- Funktionsaufrufe sind bis jetzt im Code **mit Name explizit** angegeben („hartcodiert“)
- **Folge**: Zur Compilierzeit muss bekannt sein, **welche Funktion** eine bestimmte Aufgabe erfüllen soll.

- **Beispielszenario**: Wir haben einen Sortieralgorithmus geschrieben, der ein Feld von Zahlen sortiert. Dabei wollen wir flexibel bestimmen können, nach welchem **Vergleichskriterium** sortiert werden soll (größer, kleiner, 5. Ziffer in Dezimaldarstellung größer, Quersumme kleiner, ...).
- Mit den bisherigen Mitteln müsste im Algorithmus nach Vergleichskriterien unterschieden werden, z. B. mittels `if-else` oder `switch`.
- Alternativ müsste für jedes Vergleichskriterium eine separate Funktion geschrieben werden.
- **Einleuchtendere Herangehensweise**:
  - Es gibt einen **allgemeinen** Sortieralgorithmus, der das Feld nach einer bestimmten Methode durchsucht und beim Vergleich zweier Elemente **irgendein** (variables!) Vergleichskriterium heranzieht.
  - Zum Vergleich gibt es eine Reihe von **Vergleichsfunktionen**, die getrennt vom Sortierverfahren deklariert und definiert sind.
  - Beim Aufruf des Sortieralgorithmus wird eine der Vergleichsfunktionen **als Parameter** mit angegeben.

  Genau dieses Verhalten lässt sich mit Funktionszeigern erzeugen!