Chair of Embedded Systems
Department of Computer Science
University of Augsburg, Germany

Martin Frieb

Universität Augsburg University

Wintersemester 2020/2021

Informatik 1

Kapitel 14 – Komplexe Datenstrukturen

Inhaltsverzeichnis

Version 20210201-1059

14.1 Komplexe Datenstrukturen: Ein Fallbeispiel

Dieses Kapitel ist ein bisschen anders aufgebaut als die bisherigen. Wir wollen im Folgenden die *komplexen Datenstrukturen* anhand eines Anwendungsbeispieles einführen. Dafür wird das Beispiel anfangs kurz erklärt und anschließend der dazu benötigte Code vorgestellt sowie erklärt.

14.1.1 Ein Anwendungsproblem

Stellen wir uns als Erstes die Frage, wie man eine **Adressverwaltung** programmieren könnte, die folgende Aufgaben erfüllen kann:

- Eine Liste mit Adressen soll verwaltet werden
- Zu jeder Adresse sollen mehrere Eigenschaften / Datenwerte (Postleitzahl, Ort, Strasse, Hausnummer) gespeichert werden
- Folgende Verwaltungsoperationen sollen zur Verfügung stehen:
  - Neue Adresse anlegen
  - Adresse ausgeben
  - Adresse löschen
  - Adresse ändern
  - ...

Das Problem:
Es ist nicht sinnvoll, jede dieser Eigenschaften in einer eigenen Liste zu verwalten (z.B. `char *citylist[30]`, `char *plzlist[30]`, ... für bis zu `30` Adressen), da dann bei jeder Verwaltungsoperation alle Listen simultan manipuliert werden müssten.

Um das Problem zu umgehen, extrem viele Listen gleichzeitig bearbeiten zu müssen, werden im Folgenden *komplexe Datenstrukturen* eingeführt. Hierbei sollte der Name (besonders der *komplex*-Teil des Namens) einen nicht abschrecken. Ihre Verwendung ähnelt nämlich stark den bereits bekannten Datenstrukturen, wie dem `int` oder `char`.

Denn eine solche komplexe Datenstruktur ist, im Kern der Sache, nichts weiter als eine Sammlung von Variablen. Wir fassen mehrere Variablen zu einer Einheit zusammen, die gesammelt als ein Datentyp verwendet werden kann.

14.1.2 Eine neue komplexe Datenstruktur

Was wäre eine geeignete Datenstruktur für eine **Adressverwaltung**?

Hierbei muss zuerst entschieden werden, was wir alles speichern müssen. Lesen wir nochmal die Anforderung durch, stellt sich heraus: Wir benötigen eine Straße, Hausnummer, Ort sowie die Postleitzahl. Diese vier Variablen können wir in jeweils als Zeichenkette (`char`-Feld) speichern (aufgrund von führenden Nullen, benötigten Kombinationen von Zahlen und Buchstaben, etc. sollten auch Hausnummer und Postleitzahl aus `char`s bestehen).

Diese zusammengehörigen Datenwerte einer Adresse fassen wir nun zu einer neuen Datenstruktur namens `address` zusammen. Da es sich nicht um eine Funktionsimplementierung, sondern die Deklaration eines neuen Datentyps handelt, ist diese gut in einer Header-Datei aufgehoben (und lässt sich somit auch leicht z.B. in mehreren .c-Dateien verwenden).

```
typedef struct _address {
    char street[MAX_NAME + 1];
    char city[MAX_NAME + 1];
    char number[MAX_NUMBER + 1];
    char zip[SIZE_ZIP + 1];
} address;
```

Das obere Codestück stellt bereits die komplexe Datenstruktur dar. Sie lässt sich im Grunde in zwei Teile aufteilen:
Teil 1: Der eigentliche komplexe Datentyp:

```
struct _address {
    char street[MAX_NAME + 1];
    char city[MAX_NAME + 1];
    char number[MAX_NAME + 1];
    char zip[MAX_NAME + 1];
}
```

Das Schlüsselwort `struct` dient der Definition eines neuen, komplexen Datentyps. Dieser beinhaltet die benötigten vier `char`-Felder: Straße, Ort, Hausnummer und Postleitzahl. Jedes dieser Felder ist durch symbolische Konstanten auf eine maximale Länge von Zeichenketten begrenzt (jeweils +1 für die abschließende `\0`.

Wird nun eine Variable `a` vom Typ dieser Datenstruktur erstellt, besitzt `a` die *Komponenten* `street`, `city`, `number` und `zip`.

Um auf diese zugreifen zu können, stehen uns der *Punkt-Operator* `.` und der *Pfeil-Operator* `->` zur Verfügung. Abhängig von der Situation verwenden wir den einen oder den anderen:

- Der Punkt-Operator wird für Objekte verwendet. Das bedeutet: Für alles, was kein Zeiger ist, kann man den Punkt-Operator verwenden.
- Der Pfeil Operator wird für Referenzen verwendet. Will man einen Zeiger bzw. die Variable als Zeiger verwenden, muss der Pfeil-Operator verwendet werden.

Wenn wir eine eine Variable `a` anlegen wollen, die unser `struct` umsetzt, müssen wir folgendes schreiben: `struct _adress a`. Dies ist etwas umständlich, wir werden dies aber nach dem Beispiel vereinfachen.

> Beispiel:
>
> Ein kurzes Beispiel zum ausprobieren und nachvollziehen.
>
> ```
> #include <stdio.h>
> ```

```c
#define MAX_NAME 20

struct _address {
    char street[MAX_NAME + 1];
    char city[MAX_NAME + 1];
    char number[MAX_NAME + 1];
    char zip[MAX_NAME + 1];
};

int main(void)
{
    struct _address t;

    t.city[0] = 'c';
    t.city[1] = 'i';
    t.city[2] = 't';
    t.city[3] = 'y';
    t.city[4] = '\0';

    printf("%s", t.city);

    return 0;
}
```

Teil 2: Das typedef:

```c
typedef ... address;
```

Damit wir nicht bei jeder Deklaration einer Variable unseres komplexen Datentyps `struct _address` als Typ schreiben müssen, können wir einen Alias dazu deklarieren, über den die komplexe Datenstruktur ansprechbar ist. Wir können also unser `struct _adress` zu `address` umbenennen.

Beispiel:

Ein kurzes Beispiel zum ausprobieren und nachvollziehen.

```c
#include <stdio.h>

#define MAX_NAME 20

typedef struct _address {
    char street[MAX_NAME + 1];
    char city[MAX_NAME + 1];
    char number[MAX_NAME + 1];
    char zip[MAX_NAME + 1];
```

```
} address;

int main(void)
{
    address t;

    t.city[0] = 'c';
    t.city[1] = 'i';
    t.city[2] = 't';
    t.city[3] = 'y';
    t.city[4] = '\0';

    printf("%s", t.city);

    return 0;
}
```

14.1.3 Verwaltungsoperationen

Wir haben nun die eine Hälfte unserer Anforderungen gemeistert: Das Erstellen einer praktischen Datenstruktur. Nun müssen wir dafür noch einige Verwaltungsoperatoren zu Verfügung stellen. Sie lassen sich in drei Lager aufteilen: die `get-`, `set-` und `check`-Funktionen.

`set`-Funktionen behandeln das Setzen eines Wertes. Oftmals müssen bestimmte Invarianten, Einschränkungen, etc. beachtet werden. Diese werden, falls vorhanden, in einer `check`-Funktion implementiert. In der `set`-Funktion kann man nun diese `check`-Funktion aufrufen und anschließend entweder den Wert setzen oder einen Fehlerwert zurück geben. Die `get`-Funktionen behandeln zu guter Letzt das Zurückgeben einer Variable.

get-Funktionen

Eine **get-Funktion** gibt den Wert einer Komponente einer Variable vom Typ `address` zurück:

```
char *address_get_street(address *a);
char *address_get_city(address *a);
char *address_get_number(address *a);
char *address_get_zip(address *a);
```

Für jede Komponente gibt es eine zugehörige get-Funktion. An die get-Funktion wird die Adresse einer Variable vom Typ `address` übergeben (würde man eine Variable vom Typ `address` übergeben, würde mehr Speicherplatz für deren Kopie gebraucht; außerdem folgt der Aufruf von `get`-Funktionen dann dem gleichen Prinzip wie der von `set`-Funktionen).

Beispiel:

Im Grunde sehen die meisten `get`-Funktionen gleich aus. Oftmals bestehen sie nur aus einer `return` Anweisung.

```
char *address_get_street(address *a)
{
    return a->street;
}
```

Hier gilt es zu beachten: Es wird ein Zeiger übergeben. Somit können wir **nicht** den Punkt-Operator verwenden, um auf die `street` zugreifen zu können. Stattdessen müssen wir den Pfeil-Operator verwenden.

set-Funktionen

Eine **set-Funktion** setzt den Wert einer Komponente einer Variable vom Typ `address` neu:

```
int address_set_street(address *a, char *street);
int address_set_city(address *a, char *city);
int address_set_number(address *a, char *number);
int address_set_zip(address *a, char *zip);
```

Auch bei set-Funktionen gibt es für jede Komponente eine zugehörige Funktion. An die set-Funktion wird die Adresse einer Variable vom Typ `address` und der neue Wert der Komponente übergeben. Falls der übergebene neue Wert **gültig ist**, wird er der zugehörigen Komponente zugewiesen und `1` zurückgegeben. Falls der übergebene neue Wert **ungültig ist**, wird `0` zurückgegeben (der Wert der Komponente bleibt dann unverändert).

Beispiel:

Auch die meisten `set`-Funktionen sind ähnlich aufgebaut. Sie bestehen aus einem `check`, womit die Gültigkeit der Eingabe überprüft wird. Ist sie gültig, wird der Wert gesetzt.

```
int address_set_street(address *a, char *street)
{
    if (address_check_street(street) == 0) {
       return 0;
    }
    strcpy(a->street, street);
    return 1;
}
```

check-Funktionen

Eine **check-Funktion** überprüft, ob ein Wert gültig für eine Komponente einer Variable vom Typ `address` ist:

```
int address_check_street(char *street);
int address_check_city(char *city);
int address_check_number(char *number);
int address_check_zip(char *zip);
```

Für jede Komponenten wo dies sinnvoll ist, gibt es eine zugehörige `check`-Funktion. Falls der übergebene Wert für die Komponente **gültig ist**, wird `1` zurückgegeben, sonst `0`. Hierbei gilt: Ein Wert ist gültig, falls alle **Datenstrukturinvarianten** erfüllt sind. Dabei sind Datenstrukturinvarianten Eigenschaften, die die Werte der Komponenten einer Datenstruktur erfüllen müssen. Wenn ein beliebiger Wert zulässig ist und auch sonst nichts zu überprüfen ist, kann man die `check`-Funktion auch weglassen.

> Beispiel:
>
> Eine mögliche Datenstrukturinvariante für `street` könnte beispielsweise folgendermaßen aussehen:
>
> - `street` darf höchstens aus `MAX_NAME` Zeichen bestehen.
>
> Somit ergibt sich folgende `check`–Funktion:
>
> ```
> int address_check_street(char *street)
> {
>     if (strlen(street) > MAX_NAME) {
>         return 0;
>     }
>     return 1;
> }
> ```
>
> Weitere denkbare Datenstrukturinvarianten könnten beinhalten:
>
> - `street` beginnt mit einem Großbuchstaben
> - `street` besteht nur aus Klein- und Großbuchstaben und dem Bindestrich-Zeichen

init-Funktion

Zusätzlich zu den `set`-Funktionen gibt es eine `init`-Funktion. Ihre Aufgabe ist das Initialisieren einer neuen Adresse mit übergebenen Werten für ihre Komponenten. Hierfür ruft sie die `set`-Funktionen auf. Für die komplexe Datenstruktur `address` sieht die `init`–Funktion folgendermaßen aus:

```
int address_init(address *a, char *street, char *city, char *
    number, char *zip)
```

```
{
    if (address_set_street(a, street) == 0) return 0;
    if (address_set_city(a, city) == 0) return 0;
    if (address_set_number(a, number) == 0) return 0;
    if (address_set_zip(a, zip) == 0) return 0;
    return 1;
}
```

Dabei werden die Werte der Komponenten und die Adresse einer Variable vom Typ `address` (die vorher statisch oder dynamisch angelegt werden muss) übergeben. Die Werte der Komponenten werden mit den set-Funktionen gesetzt. Falls alle übergebenen Werte gültig sind, wird 1 zurückgegeben, sonst 0.

Hauptprogramm

Im Folgenden sehen wir ein kleines Hauptprogramm, das eine statische Variable `a` vom Typ `address` angelegt und diese an die Verwaltungsoperationen übergibt. Am Ende wird `a` noch ausgegeben.

```
#include <stdio.h>

int address_init(address *a, char *street, char *city, char *
   number, char *zip);
void address_print(address *a);

int address_check_street(char *street);
int address_check_city(char *city);
int address_check_number(char *number);
int address_check_zip(char *zip);

int address_set_street(address *a, char *street);
int address_set_city(address *a, char *city);
int address_set_number(address *a, char *number);
int address_set_zip(address *a, char *zip);

char *address_get_street(address *a);
char *address_get_city(address *a);
char *address_get_number(address *a);
char *address_get_zip(address *a);

int main(void)
{
    int status;
    address a;
```

```
    status = address_init(&a, "Universitaetsstrasse", "Augsburg
        ","6a", "86159");

    if (status == 0) {
        printf("Adresse konnte nicht angelegt werden\n");
        return 1;
    } else {
        address_print(&a);
    }
    return 0;
}

void address_print(address *a)
{
    printf("%s %s\n", address_get_street(a), address_get_number
        (a));
    printf("%s %s\n", address_get_zip(a), address_get_city(a));
}

...
```

14.2 Neue Namen für Datentypen mit `typedef`

Nachdem wir die Anwendung von `typedef` bereits im Fallbeispiel betrachtet haben, werfen wir nun noch einen allgemeinen Blick darauf.

Neue Namen für Datentypen mit `typedef` vereinbaren

Definition: 14.3 Vereinbarung von Datentyp-Namen

Nach einer Vereinbarung der Form
`typedef <Datentyp> <Name>;`
ist `<Name>` wie der Datentyp `<Datentyp>` verwendbar.

- Wird benutzt um in Header-Dateien Namen von Datentypen system-abhängig zu definieren und so system-unabhängig verwenden zu können.
- Wird benutzt um Datentypen intuitive sprechende Namen gemäß ihrer Benutzung zu geben und so Programme deutlich lesbarer zu gestalten.

Feld-Datentypen kann wie folgt ein neuer Name gegeben werden:
`typedef <Datentyp> <Name>[N];`
Die Anzahl `N` der Komponenten wird also hinten angestellt.

Beispiel:

`typedef unsigned long size_t;`
Neuer Name `size_t` für den Datentyp `unsigned long` zur Angabe von Speichergrößen bei Benutzung des gcc-Compilers.

Beispiel:

Ein anderes (sinnloseres) Beispiel:

```
#include <stdio.h>

int main(void)
{
    typedef int cool_int;

    int y = 1;

    cool_int x = 5;

    printf("%i\n", x);
    printf("%i", y);

    return 0;
}
```

`int` wird hier umbenannt zu `cool_int`. Dieses lässt sich analog zum normalen `int` verwenden. Außerdem kann man weiterhin das normale `int` verwenden.

14.3 Neue Datentypen mit `struct`

Wie wir im Fallbeispiel gesehen haben, lassen sich mit `struct` neue Datentypen definieren. Betrachten wir `struct` und seine Eigenschaften noch einmal genauer.

Neue Datentypen mit `struct` vereinbaren

Definition: 14.5 Vereinbarung von `struct`-Datentypen

Mit einer Vereinbarung der Form

```
struct <Etikett> {
    <Komponenten>
};
```

definiert man einen **neuen Datentyp** `struct <Etikett>`, der Variablen unterschiedlichen Typs, seine **Komponenten**, unter einem Namen zusammenfasst.

Jede Komponente wird wie ein Variable in der Form

```
T <Komponente>;
```

vereinbart.

Diese Vereinbarung reserviert **keinen** Speicherplatz, sondern definiert einen neuen Datentyp, den man zur Deklaration von Variablen verwenden kann.

Beispiel:

Ein (weniger abstraktes) Beispiel wäre die Implementierung einer Kühlschrank Datenstruktur. Dieser Kühlschrank ist jedoch sehr speziell, denn es dürfen nur bestimmte Lebensmittel hineingelegt werden: Birnen, Käse, Ketchup und Joghurt.

Diese Datenstruktur beschreibt nun für diesen Kühlschrank, wie viele Lebensmittel sich darin befinden.

```
struct kuehlschrank {
    int birnen;
    int kaese;
    int ketchup;
    int joghurt;
};
```

Hier gilt zu beachten: Den Strichpunkt am Ende nicht vergessen!

14.3.1 Eigenschaften von `struct`-Datentypen

Variablen deklarieren:
Die Deklaration einer Variable `x` vom Typ `struct E` erfolgt wie bei den bisherigen Datentypen:

```
struct E x;
```

Die Deklaration bewirkt wie üblich die statische Reservierung von Speicherplatz. Die Komponenten von `x` werden (ähnlich wie bei Feldern) in aufeinanderfolgenden Speicherzellen abgespeichert. Wichtig ist, dass das Schlüsselwort `struct` mit zum Datentyp gehört und daher immer dazu geschrieben werden muss (außer man nutzt `typedef` zur Umbenennung).

Speicherbedarf abfragen:
Der Speicherbedarf einer Variable `x` vom Typ `struct E` kann wie üblich mit `sizeof(x)` oder `sizeof(struct E)` abgefragt werden. Er entspricht wegen dem sog. **Alignment** i.d.R. **nicht** der Summe der Speicherbedarfe seiner Komponenten.

Alignment:
Für einen schnellen Zugriff ist auf jedem System die Speicheradresse einer Komponente `k` eines `struct`-Datentyps durch eine feste Zahl `b` teilbar, wobei `b` üblicherweise 2, 4 oder 8 ist. Ist der Speicherbedarf des Datentyps von `k` nicht durch `b` teilbar, so entstehen im Speicher Lücken zwischen den Komponenten.

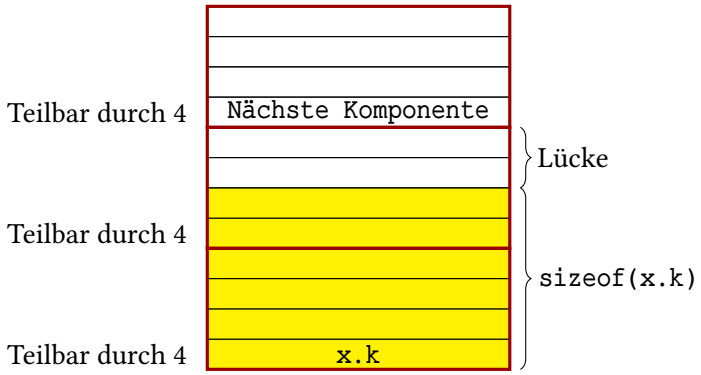

Abbildung 1: Beispielhafte Darstellung im Speicher. `x.k` benötigt einen Speicherbedarf von 6 Speicherzellen. Dadurch entsteht eine Lücke von 2 Speicherzellen.

Ein `struct` aus jeweils einem `double`, `int` und `char` kann damit z.B. auf 16 statt 8+4+1=13 Byte kommen (bei einem Alignment von 4). Bei einem Alignment von 8 käme man sogar auf 24 Byte.

Wertzuweisungen:
Man kann direkt der Komponente `k` einer Variable `x` vom Typ `struct E` einen neuen Wert `w` zuweisen mit:
`x.k = w;`
(da `x.k` eine einfache Variable ist)

Beispiel:

```
#include <stdio.h>

typedef struct _bsp {
    int n;
    char c;
    double x;
} bsp;

int main(void)
{
    bsp b;

    b.n = 1;
    b.c = 'a';
    b.x = 4.2;

    printf("%i, %c, %f", b.n, b.c, b.x);

    return 0;
}
```

Ist stattdessen ein Zeiger `p` auf ein `struct E` gegeben, müsste man diesen zuerst dereferenzieren und kann erst dann auf die Komponente zugreifen:

```
(*p).k = w;
```

Mit dem Pfeil-Operator wird die Dereferenzierung automatisch durchgeführt. Folgende Schreibweise ist daher äquivalent zur obigen:

```
p->k = w;
```

Beispiel:

Für das Beispiel muss ein bisschen vorgegriffen werden. Einzelheiten zur Erzeugung eines Zeigers auf ein `struct` werden weiter unten behandelt.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct _bsp {
    int n;
    char c;
```

```
    double x;
} bsp;

int main(void)
{
    bsp *b = malloc(sizeof(bsp));

    b->n = 1;
    b->c = 'a';
    (*b).x = 4.3;  /* Alternativ */

    printf("%i, %c, %f", b->n, b->c, b->x);

    free(b);

    return 0;
}
```

Sind `x,y` Variablen vom Typ `struct E`, so kann man die in den Komponenten von `y` gespeicherten Werte direkt nach `x` kopieren:
`x = y;`
(im Gegensatz zu Feldern)

Beispiel:

```
#include <stdio.h>

typedef struct _bsp {
    int n;
    char c;
    double x;
} bsp;

int main(void)
{
    bsp b;
    bsp b2;

    b.n = 1;
    b.c = 'a';
    b.x = 4.2;
```

```
    b2 = b;

    printf("%i, %c, %f", b2.n, b2.c, b2.x);

    return 0;
}
```

Man kann den Komponenten einer Variable `x` vom Typ `struct E` direkt in der Deklaration Werte über eine Liste von Konstanten zuweisen:
`struct E x = {<Konstantenliste>};`

Hinweis: Dies ist nur bei der Deklaration eines `structs` möglich (ähnlich wie bei Feldern)!

Beispiel:

```
#include <stdio.h>

typedef struct _bsp {
    int n;
    char c;
    double x;
} bsp;

int main(void)
{
    bsp b = {1, 'a', 4.3};

    printf("%i, %c, %f", b.n, b.c, b.x);

    return 0;
}
```

Beispiel:

Nicht möglich:

```
#include <stdio.h>

typedef struct _bsp {
    int n;
    char c;
```

```
    double x;
} bsp;

int main(void)
{
    bsp b = {1, 'a', 4.3};

    printf("%i, %c, %f", b.n, b.c, b.x);

    /* Diese Zeile verursacht einen Fehler */
    b = {2, 'b', 5.3};

    return 0;
}
```

Felder vom Typ `struct E`:
Deklaration eines Feldes `v` vom Typ `struct E` mit `N` Feldkomponenten:
`struct E v[N];`

Jede Feldkomponente `v[i]` ist vom Typ `struct E`.
Zugriff auf eine Komponente `k` von `v[i]`:
`v[i].k`
(Ist `k` vom Typ `T`, so ist `v[i].k` eine Variable vom Typ `T`)

Beispiel:

```
#include <stdio.h>

typedef struct _bsp {
    int n;
    char c;
    double x;
} bsp;

int main(void)
{
    bsp b[2];

    b[0].n = 1;
    b[0].c = 'a';
    b[0].x = 4.3;

    b[1].n = 100;
    b[1].c = 'b';
```

```
    b[1].x = 400.333;

    printf("%i, %c, %f\n", b[0].n, b[0].c, b[0].
       x);
    printf("%i, %c, %f", b[1].n, b[1].c, b[1].x)
       ;

    return 0;
}
```

Komplexe Datenstrukturen als Eingabeparameter von Funktionen:
Für einen Eingabeparameter des Typs `struct E` erwartet eine Funktion bei Aufruf die Übergabe einer Variable vom Typ `struct E`. In der Funktion wird dann nach dem Call-by-Value-Prinzip mit einer Kopie der Variable gearbeitet. Da komplexe Datenstrukturen oft einen großen Speicherbedarf haben, ist es effizienter, wenn man an Funktionen grundsätzlich die Adressen von komplexen Datenstrukturen übergibt und in der Funktion mit Zeigern arbeitet. Möchte man die Werte einer komplexen Datenstruktur in einer Funktion ändern, so kann man dies **nur** nach dem Call-By-Reference-Prinzip tun und **muss** die Adresse der komplexen Datenstruktur übergeben.

Beispiel:

```
#include <stdio.h>

typedef struct _bsp {
    int n;
    char c;
    double x;
} bsp;

void foo(bsp *b);

int main(void)
{
    bsp b = {1, 'a', 4.3};

    foo(&b);

    return 0;
}

void foo(bsp *b)
{
    printf("%i, %c, %f\n", b->n, b->c, b->x);
```

```
}
```

Komplexe Datenstrukturen als Rückgabetyp von Funktionen:
Ist `struct E` der Rückgabetyp einer Funktion, so gibt sie eine Variable vom Typ `struct E` zurück. Auf diese Weise kann eine Funktion mehrere Ausgabewerte, zusammengefasst als Komponenten eines `struct`-Datentyps, haben.

Beispiel:

```
#include <stdio.h>

typedef struct _bsp {
    int n;
    char c;
    double x;
} bsp;

bsp foo(void);

int main(void)
{
    bsp b = foo();

    printf("%i, %c, %f\n", b.n, b.c, b.x);

    return 0;
}

bsp foo(void)
{
    bsp temp = {1, 'a', 4.3};

    return temp;
}
```

Zeiger auf `struct E`:
Deklaration eines Zeigers p auf `struct E`:
`struct E *p;`

Dereferenzierung:
`*p` ist eine Variable vom Typ `struct E`.

Zugriff auf eine Komponente k von `*p`:
`(*p).k`
(die Klammerung ist wegen der Auswertungsreihenfolge notwendig!)

Hierfür haben wir bereits folgende abkürzende Schreibweise kennengelernt:
`p->k`
(Ist `k` vom Typ `T`, so ist `p->k` eine Variable vom Typ `T`)

Adressverschiebung:
`p + n` verschiebt die Adresse um `n * sizeof(struct E)` Byte.
Der Ausdruck `p[n].k` entspricht `(p + n)->k`

Dynamische Speicherreservierung:
Wie für jeden anderen Datentyp.

Beispiel:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct _bsp {
    int n;
    char c;
    double x;
} bsp;

int main(void)
{
    bsp *b = malloc(sizeof(bsp));

    b->n = 1;
    b->c = 'a';
    (*b).x = 4.3;  /* Alternativ */

    printf("%i, %c, %f", b->n, b->c, b->x);

    free(b);

    return 0;
}
```

Mehrwertige Komponenten
Komponenten einer `struct`-Definition können auch statische oder dynamische Felder sein. Ist eine Komponente ein Feld, die **keine** Zeichenkette ist, so nennt man diese Komponente **mehrwertig**.

Beispiel: Namen von Personen

```
typedef struct _name {
    char nachname[MAX_NAME + 1];
```

```
    char vorname[3][MAX_NAME + 1];/*mehrwertige
       Komponente*/
} name;
int name_set_vorname(name *n, char *vorname, int
    index);
char *name_get_vorname(name *n, int index);
void name_delete_vorname(name *n, int index);
```

- Eine Person kann **mehrere** (bis zu drei) Vornamen haben, die in einem Feld verwaltet werden
- Der erste, zweite und dritte Vorname kann jeweils hinzugefügt (`set`), gelesen (`get`) und gelöscht (`delete`) werden
- Mit `index` übergibt man, um welchen Vornamen es geht

Beispiel:

```
#include <stdio.h>

typedef struct _bsp {
    int n[2];  /* Mehrwertige Komponente */
    char c;
    double x;
} bsp;

int main(void)
{
    bsp b = {{1, 2}, 'a', 4.3};

    printf("%i, %i, %c, %f\n", b.n[0], b.n[1], b
       .c, b.x);

    return 0;
}
```

Verschachtelte komplexe Datenstrukturen

Komponenten einer `struct`-Definition können selbst wieder einen `struct`-Datentyp haben

Beispiel: Personen mit Namen und Adressen

```
typedef struct _person {
    name pname;
    address paddress[2];
} person;
```

- Eine Person hat einen Namen und **mehrere** (bis zu zwei) Adressen
- Die Komponente für den Namen `pname` hat den komplexen Datentyp `name`
- Die Komponente für die Adressen `paddress` ist ein Feld der Länge 2 mit dem komplexen Datentyp `address`

Zugriff auf Komponenten einer Variable `p` vom Typ `person`:

- `p.pname`: Name (vom Typ `name`)
- `p.pname.nachname`: Nachname des Namens (vom Typ `char *`)
- `p.pname.vorname[0]`: Erster Vorname des Namens (vom Typ `char *`)
- `p.paddress[0]`: Erste Adresse (vom Typ `address`)
- `p.paddress[0].zip`: Postleitzahl der ersten Adresse (vom Typ `char *`)

Beispiel:

```c
#include <stdio.h>

typedef struct _bsp {
    char c;
    double x;
} bsp;

typedef struct _bsp2 {
    int n;
    bsp verschachtelt;
} bsp2;

int main(void)
{
    bsp2 b;

    b.n = 1;
    b.verschachtelt.c = 'a';
    b.verschachtelt.x = 4.3;

    printf("%i, %c, %f\n", b.n, b.verschachtelt.
       c, b.verschachtelt.x);

    return 0;
}
```

14.3.2 Verwaltungsoperationen für dynamische komplexe Datenstrukturen

Wenn wir es dem Anwender einfach machen wollen, Variablen vom Typ unserer komplexen Datenstruktur auf dem Heap zu speichern, stellen wir ihm zwei weitere Verwaltungsoperationen zur Verfügung: `new` und `destroy` bzw. im Sinne unseres Adressverwaltungsbeispiels `address_new` und `address_destroy`.

new
`new` legt mit Hilfe von `calloc` eine neue Variable vom Typ unserer komplexen Datenstruktur im Heap an, initialisiert diese mit Nullwerten (0 bzw. `NULL` bei Zeigern) und liefert die Adresse auf die neue Instanz an den Aufrufer zurück.

Beispiel:

```
address *address_new(void)
{
    address *a = calloc(1, sizeof(address));
    /* Initialisierung bereits via calloc
       erledigt */
    return a;
}
```

destroy
`destroy` kann es in zwei verschiedenen Varianten geben. Die eine ist das Gegenstück zu `new` – dann ruft `destroy` `free` für die Datenstruktur und ggf. dynamisch reservierte Komponenten auf. Es ist zu beachten, dass diese Variante nur für solche Variablen aufrufbar ist, die vorher via `new` auf dem Heap angelegt wurden!

Beispiel:

```
void address_destroy(address *a)
{
    free(address);
}
```

Die andere Variante implementiert man für komplexe Datenstrukturen, für die kein `new` vorgesehen ist, aber die Elemente enthalten, die dynamisch reserviert werden. Das ist beispielsweise dann der Fall, wenn die komplexe Datenstruktur nur einen Zeiger auf etwas vorsieht und dieser in der `set`-Funktion via `malloc` o.ä. reserviert wird. In diesem Fall wird der in der `set`-Funktion reservierte Speicher wieder freigegeben. Mit dieser Variante befassen Sie sich auf dem Übungsblatt.