Vorlesung Informatik 1 (Wintersemester 2020/2021)

Kapitel 14: Komplexe Datenstrukturen

Martin Frieb
Johannes Metzger

Universität Augsburg
Fakultät für Angewandte Informatik

01. Februar 2021

Ein Anwendungsproblem

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

- 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
  - ...

Problem

Es ist nicht sinnvoll, jede der 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.

Eine neue komplexe Datenstruktur

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

Fasse zusammengehörige Datenwerte einer Adresse zu einer neuen Datenstruktur `address` zusammen (Vereinbarung in separater Header-Datei `address.h`):

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

- Hier wird ein **neuer komplexer Datentyp** `struct _adress` definiert und mit `typedef` in `address` umbenannt (`struct` ist ein Schlüsselwort zur Definition komplexer Datentypen)
- Eine Variable `a` vom Typ `address` repräsentiert eine Adresse und hat die Komponenten `street`, `city`, `number`, `zip`
- Zugriff auf Komponenten einer Adresse `a` mit Punkt-Operator: `a.street`, ...
- Symbolische Konstanten definieren die (maximale) Länge von Zeichenketten

Verwaltungsoperationen

get-Funktionen – Deklaration in `address.h`

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)

Verwaltungsoperationen

set-Funktionen - Deklaration in `address.h`

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);
```

- Für jede Komponente gibt es eine zugehörige set-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 unverändert)

Verwaltungsoperationen

get-Funktionen – Implementierung in `address.c`

Zugriff mit Zeiger `a` auf die Komponente `street` mit dem `->` - Operator.

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

set-Funktionen – Implementierung in `address.c`

Die check-Funktion überprüft die Gültigkeit von `street`

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

Verwaltungsoperationen

check-Funktionen – Deklaration in `address.h`

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 Komponente gibt es eine zugehörige check-Funktion
- Falls der übergebene Wert für die Komponente **gültig ist**, wird `1` zurückgegeben, sonst `0`

Datenstrukturinvariante

- Eine **Datenstrukturinvariante** ist eine Eigenschaft, die die Werte der Komponenten einer Datenstruktur erfüllen müssen
- Ein Wert heißt **gültig**, falls er alle Datenstrukturinvarianten erfüllt

Verwaltungsoperationen

check-Funktionen – Implementierung in `address.c`

Die check-Funktion überprüft `street` auf Erfüllung der vorhandenen Datenstrukturinvarianten:

- `street` darf höchstens aus `MAX_NAME` Zeichen bestehen.

```
int address_check_street(char *street)
{
    if (strlen(street) > MAX_NAME) return 0;
    return 1;
}
```

Weitere denkbare Datenstrukturinvarianten:

- `street` beginnt mit einem Großbuchstaben
- `street` besteht nur aus Klein- und Großbuchstaben und dem Bindestrich-Zeichen

Verwaltungsoperationen

init-Funktion – Implementierung in `address.c`

Die **init-Funktion** initialisiert eine `address`-Datenstruktur mit übergebenen Werten.

```
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;
}
```

- Es 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

Benutzung der neuen Datenstruktur

Es wird statisch eine Variable `a` vom Typ `address` angelegt. An die Verwaltungsoperationen wird deren Adresse übergeben.

```
#include "address.h"
#include <stdio.h>

int main(void)
{
  address a;
  if (address_init(&a, "Universitaetsstrasse", "Augsburg","6
      a", "86159") == 0) {
    printf("Adresse konnte nicht initialisiert werden\n");
    return 1;
  } else
    address_print(&a);
  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 14.1 (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

Verschachtelte Strukturen

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

Beispiel 14.2 (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 strukturierten Datentyp `name`
- Die Komponente für die Adressen `paddress` ist ein Feld der Länge `2` mit dem strukturierten 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 *`)

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 systemabhängig 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 14.4

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

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: Repräsentation von Dateien in `stdio.h`

Beispiel 14.6

```
typedef struct _iobuf {
            char* _ptr;
            int _cnt;
            char* _base;
            int _flag;
            int _file;
            int _charbuf;
            int _bufsiz;
            char* _tmpfname;
} FILE;
```

- Variablen vom Typ `FILE` repräsentieren einen Datenstrom von oder zu einer Datei
- Die `FILE`-Komponenten beinhalten u.a. einen Puffer zum Zwischenspeichern von Informationen aus der Datei im Arbeitsspeicher
- Der Zugriff erfolgt ausschließlich über Verwaltungsoperationen aus `stdio.h`
- Zum Beispiel ist der Standardeingabestrom `stdin`, auf den man mit `scanf` und `getchar` zugreift, vom Typ `FILE`

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.

Zugriff auf Komponenten

Ist `k` eine Komponente einer Datenstruktur `struct E`, und ist `x` eine Variable vom Typ `struct E`, so kann man wie folgt mit dem `.` - Operator

`x.k`

auf die Komponente `k` zugreifen.

Ist die Komponente `k` vom Typ `T`, so ist `x.k` eine Variable vom Typ `T`.

Eigenschaften von `struct`-Datentypen

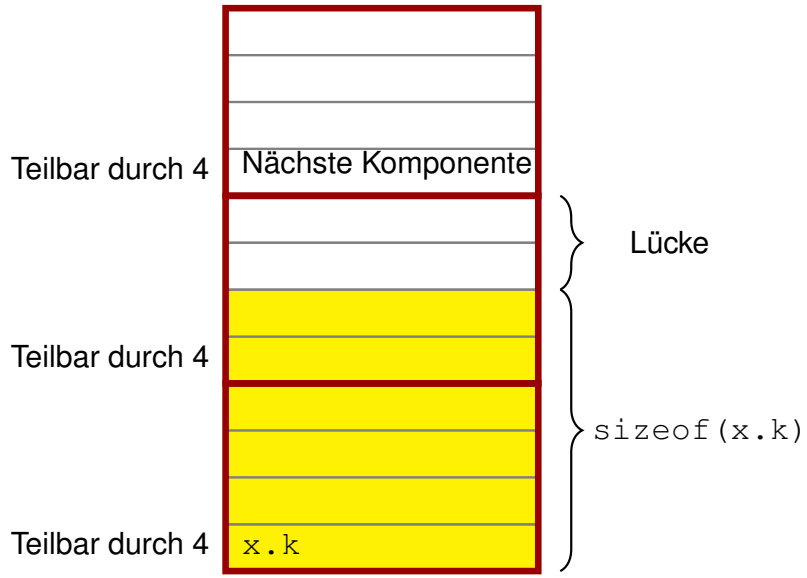
Der Speicherbedarf eines `struct`-Datentyps ist systemabhängig und kann wie üblich mit `sizeof` abgefragt werden. Er entspricht wegen dem sog. **Alignment** i.d.R. **nicht** der Summe der Speicherbedarfe seiner Komponenten.

Speicherbedarf abfragen

Der Speicherbedarf einer Variable `x` vom Typ `struct E` kann wie üblich mit **`sizeof`**`(x)` oder **`sizeof`**`(`**`struct`**` E)` abgefragt werden.

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.



Eigenschaften von `struct`-Datentypen

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)
- 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)
- 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>};`
  (ähnlich wie bei Feldern)

Eigenschaften von `struct`-Datentypen

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`)

Eigenschaften von `struct`-Datentypen

Strukturen als Eingabeparameter von Funktionen

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

Strukturen 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 zu Komponenten eines `struct`-Datentyps, haben.

Eigenschaften von `struct`-Datentypen

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 existiert folgende abkürzende Schreibweise:
  `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.

Verwaltungsoperationen für dynamische `structs`

Kapselung von `calloc` und `free` in eigenen Verwaltungsoperationen

new, z.B. `address_new`

- Anlegen und Initialisieren einer neuen Instanz der komplexen Datenstruktur auf dem Heap via `calloc`
- Initialisieren mit Nullwerten (`calloc`)
- Zurückliefern der Adresse der neuen Instanz

destroy, z.B. `address_destroy`

Zwei Varianten möglich:

1. Gegenstück zu `new`:
   - Aufruf von `free` für komplexe Datenstruktur und ggf. dynamisch reservierte Komponenten
   - Darf nur aufgerufen werden, wenn komplexe Datenstruktur via `new` auf dem Heap angelegt wurde!
2. Freigabe von Speicher, der in `set`-Funktionen dynamisch reserviert wurde, Rest der komplexen Datenstruktur bleibt erhalten