Einführung in die Programmierung

Konstanten, dynamische Datenstrukturen

Arvid Terzibaschian

Konstanten

Motivation

- Unveränderliche, wichtige Werte
  - mathematische Konstanten z.B. PI
  - String-Konstanten wie z.B. häufige statische Meldungen
  - mögliche Werte von kategorischen Attributen
- Ablegen in Variablen:
  - Variablen könnten verändert werden

```
double pi = 3.14159265358979323846264338327950;
pi = 3; // hahahah!!!
```

  - zusammengehörige kategorische Attribute einzeln definieren

```
int ohne_ausbildung = 0;
int hauptschule = 1;
int realschule = 2;
int abitur = 3;
int diplom = 1; // oops
```

Konstanten 1: Zahlen und Strings

- konstante Variablen mit Schlüsselwort „const [typ]“ deklarieren:

```
const double pi = 3.1415926535897932384626433832795;
const char* err214 = “Fehler 214 ist aufgetreten!“;
```

- möglich für Variablen, Zeiger und Parameter
  - Wert kann initialisiert und danach nur noch gelesen werden

```
pi = 3; // Compilerfehler/-warnung

// Compilerfehler/-warnung
sprintf(err214,“Mal was reinschreiben“);
```

Konstanten 2: Kategorie-Attribute

- „enum“ definiert Kategorien- und Aufzählungstypen

```
int ohne_ausbildung = 0;
int hauptschule = 1;
int realschule = 2;
int abitur = 3;
int diplom = 1; // oops
```

```
enum ausbildung_t {
        AUSBLDNG_OHNE,
        AUSBLDNG_HS,
        AUSBLDNG_RS,
        AUSBLDNG_ABITUR,
        AUSBLDNG_DIPLOM,
};
```

- `enum ausbildung_t` ist neuer Datentyp (vgl. struct)
  - kann für Variablen und Parameter genutzt werden

```
int hat_abitur(enum ausbildung_t a) {
    if(a == AUSBLDNG_ABITUR
    || a == AUSBLDNG_DIPLOM)
        return 1;
    else
        return 0;
}
```

```
int main() {
    enum ausbildung_t peter = AUSBLDNG_HS;
    if(hat_abitur(peter))
        printf(“hat Abitur!“);
    else
        printf(„kein Abitur!“);
}
```

Konstanten 3: #define

- #define ist Präprozessor-Anweisung
  - zu erkennen an # (bisher z.B. #include)
  - Präprozessor dient „Vorverarbeitung“ von Quellcode-Dateien
    - z.B. #include <stdio.h> wird vor Kompilierung mit Text aus Datei „stdio.h“ ersetzt
- #define für Konstanten:

```
#define PI 3.14159265358979323846264338327950

double kreis_umfang(double radius) {
        return 2*PI*radius;
}
```

  - jedes Vorkommen von „PI“ wird Textuell durch „3.14159…“ ersetzt.
    - erst danach kompiliert
- Nachteile:
  - keine Typsicherheit (welche Typ ist PI?)
  - bei Schreibfehlern schwer verständliche Fehlermeldungen
    - kompiliert teilweise sogar ohne Fehler
  - alles in Großbuchstaben ist übliche Schreibweise (vgl. NULL)

Zusammenfassung Konstanten

- Vorteile:
  - „Dingen einen Namen geben …“
    (PI, Error, ROT, GRUEN …)
    - erhöhte Lesbarkeit & Wartbarkeit
  - können nicht verändert werden
- 3 Möglichkeiten
  - **const Variablen**
    - Vorteil: Typprüfung durch den Compiler
    - Nachteil: nicht für Aufzählungstypen geeignet
  - **enum**
    - Vorteil: eigener Datentyp für kategoriale Variablen
      - erhöht Lesbarkeit und Wartbarkeit von Code
    - Nachteil: nicht für Strings oder reelle Zahlen geeignet
  - #define
    - Vorteil: Durch Namenskonvention gut zu erkennen
    - Nachteil: Datentypfrei und Fehleranfällig durch textuelle Ersetzung

Dynamische Datenstrukturen

Problem

- Sie sollen eine Kundenverwaltung schreiben
  - Für jeden Kunden sollen die üblichen Kenndaten gespeichert werden
  - Anzahl der Datensätze ist dynamisch
  - Datensätze können jederzeit hinzugefügt oder gelöscht werden
- Wie kann man das Modellieren?
  - `struct kunde_t` für einzelnen Datensatz
  - ein großes Array für alle Kunden
- Probleme
  - hinzufügen oder löschen einzelner Elemente mit Arrays oder dynamischer Speicherverwaltung schwer möglich

Problem

```
typedef
struct kunde_t {
     int kdnr;
     char name[64];
     char adresse[64];
} cust;
```

- Kundenverwaltung
  - Datensätze sollen jederzeit hinzugefügt oder gelöscht werden
- Beispiel: „Löschen von Kunden“

```
void delete_cust(cust db[],int n,int kdnr) {
     int i = 0;
     for(i = 0;i<n;++i) {
          if(db[i].kdnr == kdnr)
               db[i] = ???;
     }
}
```

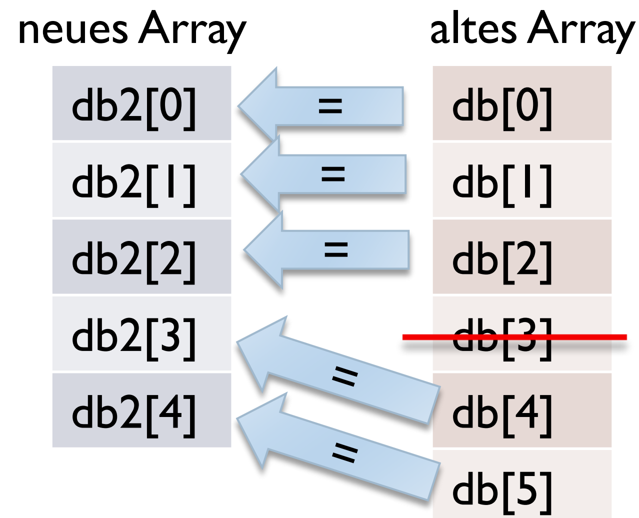
  - eine Möglichkeit:
    - neues Array auf dem Heap anlegen „db2“, mit 1 Element weniger
    - alles kopieren, außer zu löschendem Eintrag

Problem

```c
typedef
struct kunde_t {
	int kdnr;
	char name[64];
	char adresse[64];
} cust;
```

- Kundenverwaltung
  - Datensätze sollen jederzeit hinzugefügt oder gelöscht werden
- Beispiel: „Löschen von Kunden“

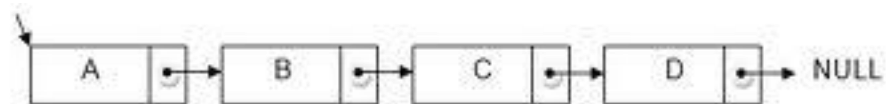
„Löschen von Element x“
==
„alle kopieren, außer x“
???



  - eine Möglichkeit:
    - neues Array auf dem Heap anlegen „db2“, mit 1 Element weniger
    - alles kopieren, außer zu löschendem Eintrag
  - unintuitiv & ineffektiv

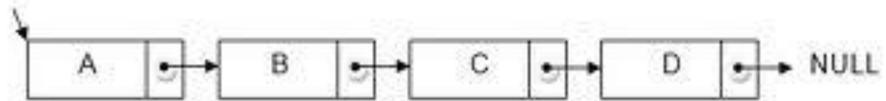
Problem

- Kundenverwaltung
  - Datensätze sollen jederzeit hinzugefügt oder gelöscht werden
- Lösung: verkettete Liste
  - „linked list“ ist die einfachste dynamische Datenstruktur
  - gehört zur Standardbibliothek aller modernen Sprachen
    - leider: C besitzt keine eingebaute linked-list



- Idee:
  - Jeder Datensatz besitzt neben seinen Kerndaten einen Zeiger auf einen nächsten Datensatz („next“)
    - Kette von Datensätzen ergibt eine Liste

Verkettete Liste



- Kundenverwaltung
  - Datensätze sollen jederzeit hinzugefügt oder gelöscht werden
- Lösung: verkettete Liste
- Idee:
  - Jeder Datensatz besitzt neben den eigentlichen Daten einen Zeiger auf den nächsten Datensatz („next“)

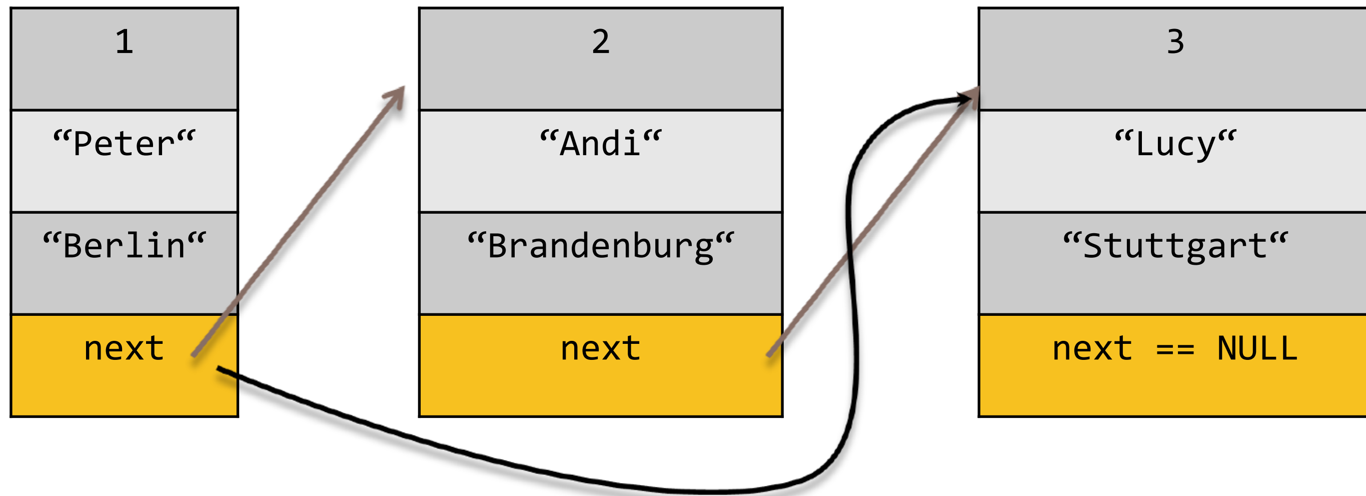
```
typedef
struct kunde_t {
     int kdnr;
     char name[64];
     char adresse[64];
     struct kunde_t* next;
} cust;
```

  - durch Verändern der „next“-Zeiger können Elemente hinzugefügt, gelöscht oder umgeordnet werden
  - Elemente liegen ***nicht*** nacheinander im Speicher wie bei Arrays

Verkettete Liste

```
typedef
struct kunde_t {
      int kdnr;
      char name[64];
      char adresse[64];
      struct kunde_t* next;
} cust;
```

- Verkettete Liste für Kundenstamm
  - Löschen von “Andi“



  - Hinzufügen und Umordnen nach demselben Prinzip
    - keine Daten werden kopiert
    - intuitivere Implementierung der Operationen

Implementierung: Kundenliste

- übliche Herangehensweise
  - Funktion für jede einzelne Anforderung implementieren:
    - Erstellen einer neuen Liste mit einem Element
      ```
      cust* list_create();
      ```
    - Hinzufügen eines neuen Elementes
      ```
      cust* list_new_element(cust* first);
      ```
    - Finden eines Elementes
      ```
      cust* list_find(cust* first,int kdnr);
      ```
    - Löschen eines Elements
      ```
      cust* list_delete_element(cust* first,cust *e);
      ```
    - Löschen der gesamten Liste
      ```
      void list_delete(cust* first);
      ```

mögliche Implementierung: Kundenliste

```
typedef
struct kunde_t {
        int kdnr;
        char name[64];
        char adresse[64];
        struct kunde_t* next;
} cust;
```

- neue Liste

```
/* Create new list with one element and
return by Pointer */
cust* list_create() {
     cust* c;
     c = malloc(sizeof(cust));
     c->next = NULL;
     return c;
}
```

- neues Element

```
/* Create new element and insert after
element „e“.
Return new element by pointer */
cust* list_new_element(cust* first)
     cust* new;
     new = malloc(sizeof(cust));
     new->next = first->next;
     return (first->next = new);
}
```

- finden per Kundennummer

```
cust* list_find(cust* e,int kdnr) {
     if(e != NULL) {
          if(e->kdnr == kdnr) // Gefunden!
            return e;
          else // weitersuchen …
            list_find (e->next,kdnr);
     }
     else // Ende der Liste erreicht
          return NULL;
}
```

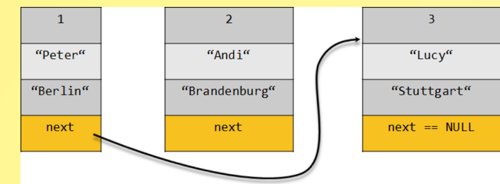
- Liste komplett löschen

```
void list_delete(cust* first) {
     if(first == NULL) return;
     cust* next = first->next;
     free(first);
     list_delete(next);
}
```

mögliche Implementierung: Kundenliste

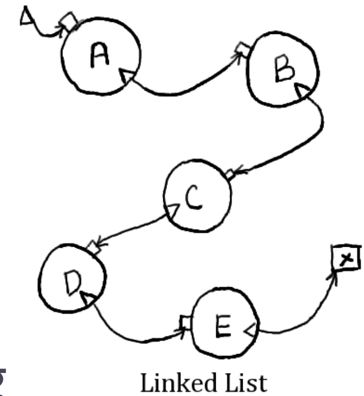
```
typedef
struct kunde_t {
        int kdnr;
        char name[64];
        char adresse[64];
        struct kunde_t* next;
} cust;
```

- einzelnes Element „e“ Löschen




```
/* Idee: Setze Next-Zeiger vom Vorgänger von „e“ auf
         Nachfolger von „e“
   Rückgabewert: erstes Element der Liste (Warum?)
*/
cust* list_delete_element(cust* first,cust *e) {
     /* bestimme Zeiger auf Vorgänger- und Nachfolgerelemente */
     cust* e_prev = find_previous(first,e);    /* finde Vorgänger von e (?)*/
     cust* e_next = e->next;                   /* Nachfolger von e*/
     free(e);                                  /* lösche speicher */
     if(e_prev != NULL) {                      // Verkette Vorgänger mit Nachfolger
          e_prev->next = e_next;
          return first;                        // 1. Element der Liste zurückgeben
     }
     else                                      // e ist 1. Element (kein Vorgänger)
          return e_next;                       // neues 1. Element zurückgeben

}
```

Implementierung: Kundenliste



Linked List

- Grundsätzlich gilt:
  - viele andere Implementierungen möglich
  - optimale Implementierung ist Anwendungsabhängig
- Eigenschaften dieser Implementierung
  - einfach verkettete Liste
    - vor Löschen eines Elementes muss Vorgänger gesucht werden
      - alternative: doppelt verkettete Liste
  - keine Funktion zum Vertauschen
  - kein spezielles Anfangselement „Liste“
    - jedes Element könnte Anfang einer Liste sein

Kundenliste einsetzen

Funktionen:
list_create()
list_new_element()
list_find()
list_delete_element()
list_delete()

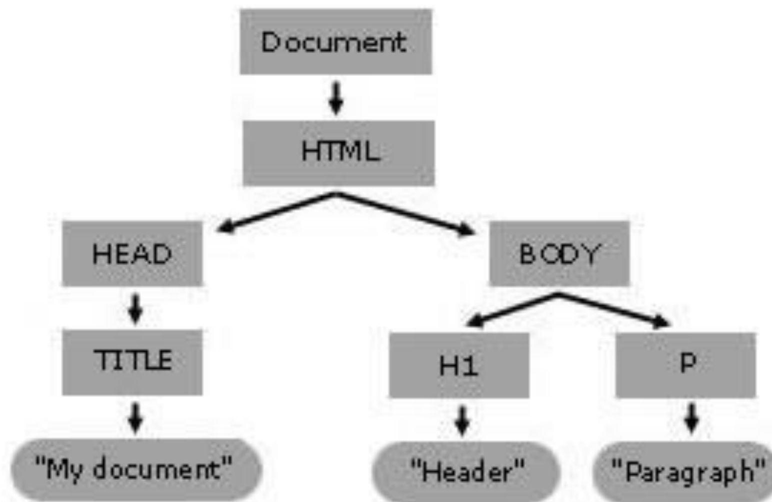
```
Datentyp:
typedef
struct kunde_t {
        int kdnr;
        char name[64];
        char adresse[64];
        struct kunde_t*
        next;
} cust;
```

- Arbeiten mit „Kundenliste“
  - Funktionen & Datentyp bestimmen „Interface“
- Beispiel:

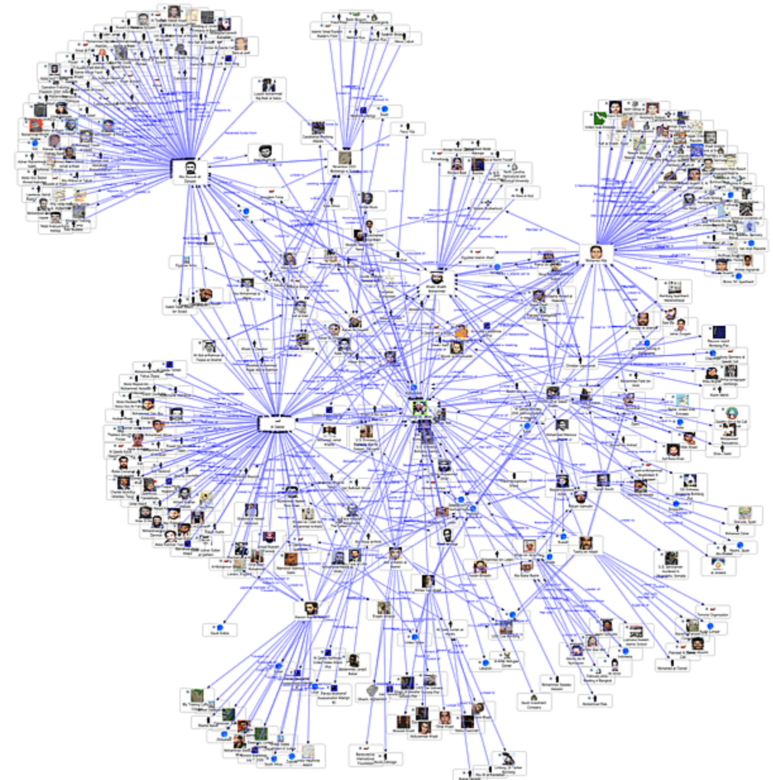
```
int main() {
     cust* e1 = list_create();
     strcpy(e1->name,“Peter“); strcpy(e1->adresse,“Berlin“),e1->kdnr = 1;

     cust* e2 = list_new_element(e1);
     strcpy(e2->name,“Frank“); strcpy(e2->adresse,“Bonn“),e2->kdnr = 2;

     cust* e3 = list_new_element(e1);
     strcpy(e3->name,“Jana“); strcpy(e3->adresse,“Prag“),e3->kdnr = 3;

     cust* e = list_find(e1,3);        // finde Eintrag mit Kundennummer 1
     if(e == NULL)
          printf(“Kundennummer 3 nicht gefunden\n“);
     else
          printf(“Kundennummer 3 ist %s\n“,e->name);
     list_delete(e1);
};
```

Ausblick: Dynamische Datenstrukturen

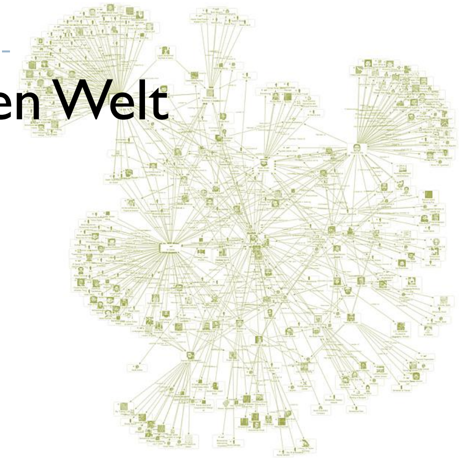
- Bäume
  - z.B. zum Verarbeiten von HTML-Dokumenten



- allgemeine Graphen
  - z.B. zum Speichern sozialer Netzwerke



Zusammenfassung: Dynamische Datenstrukturen



- können komplexe, dynamische Strukturen der realen Welt modellieren
  - Listen, Hierarchien, Netzwerke, …
  - Modellierung von Daten und deren Relationen mächtiges Entwicklungswerkzeug
    - Basis für z.B. Objektorientierung oder Datenbanksysteme
- elementare Idee ist die Verknüpfung von Daten durch Referenzen (Zeiger)
  - (unidirectional) linked list einfachste dynamische Datenstruktur
- Implementierung vieler Datenstrukturen und passenden Algorithmen in modernen Sprachbibliotheken vorhanden
  - In C Standardbibliothek gibt es (leider) keine dynamischen Datenstrukturen
    - Abhilfe z.B. durch Bibliotheken

Vielen Dank!

- Bei Fragen einfach eine Mail an:
  - arvid@cs.uni-potsdam.de

Hinweise zur Übung

Hinweise zur Übung

- **Übungsaufgabe dieses Mal:**
  - Schreiben eines komplexen Programmes
  - gerne in Teamarbeit
  - Übung geht über 2 Wochen
    - diese Woche (13.1.-17.1.): Tutoren geben Hilfestellung in Übung
    - nächste Woche (20.1.-24.1.): Tutoren stellen Musterlösung vor

- **Vorlesung nächste Woche 20.1.:**
  - Praktische Vorlesung:
    - Besprechen von schwierigen/offenen Fragen
    - „Vorprogrammieren“
    - Anwesenheit wird empfohlen

Kurzvorstellung

- Ziel ist es, eine kleine Lagerverwaltungssoftware fertig zu stellen

1. Selbständiges Anwenden des erlernten Wissens
   - Lösungen entwickeln mit C
   - Nutzen der Standardbibliothek

2. Implementieren nach Spezifikationen
   - Was genau soll das Programm tun?
   - Was sollen einzelne Programmteile tun?

1. Testen der Implementierung
   - Ist meine Implementierung „korrekt“?