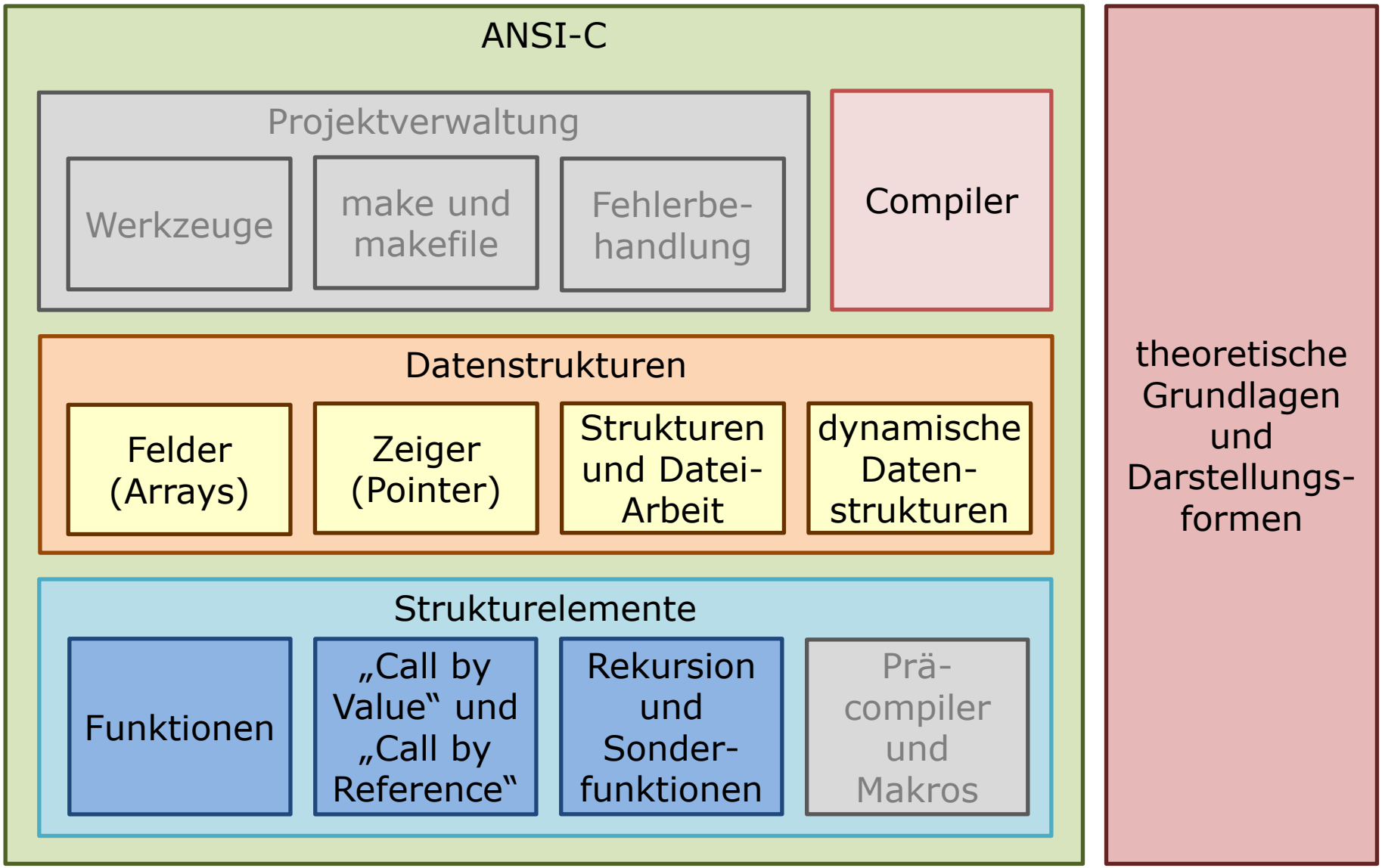
BERUFSAKADEMIE SACHSEN
STAATLICHE STUDIENAKADEMIE
DRESDEN
UNIVERSITY OF COOPERATIVE EDUCATION

Imperative Programmierung

Strukturen und Dateiarbeit

Prof. Dr.-Ing. Tenshi Hara
tenshi.hara@ba-sachsen.de

Aufbau der Lehrveranstaltung



ZUSAMMENGESETZTE DATENTYPEN

- zwei Arten von zusammengesetzten Datentypen in C
  - Structure (Verbund)
    - Bestandteile stehen im Speicher hintereinander
    - Länge wird durch die Addition der einzelnen Längen bestimmt
  - Union (Variante)
    - Bestandteile beginnen alle an gleicher Adresse
    - Länge entspricht der Länge des längsten Bestandteils
- Längen sollten in beiden Fällen mit `sizeof` ermittelt werden
  → Structures oft an Maschinenwortgrenzen angelegt
  → Summe der einzelnen Bestandteillängen ist dann falscher Wert
- das deutsche Wort „Struktur" kann beides meinen
  → im Fall der Fälle lieber noch einmal fragen!

Aufbau einer Structure

```
struct M_Person {
      char name[80+1];
      char vorname[80+1];
      int  geburtsjahr;
   } person;
```

Aufbau der Structure;

die einzelnen Bestandteile (Strukturvariablen) werden hintereinander geschrieben

Variable dieser Struktur;

Name und Speicherplatzreservierung

Structure verhält sich wie jeder andere Datentyp

- es können auch mehrere Variablen deklariert werden
- die Bezeichnung (`M_Person`) definiert nur den Aufbau der Structure
- mit der Strukturdefinition wird kein Speicherplatz reserviert
  → erfolgt erst beim Anlegen einer Variablen (hier `person`)

Zugriff auf Bestandteile einer Structure (1/2)

- mit der einmal definierten Musterstruktur können weitere Strukturen angelegt werden

```
struct M_Person person1, person2;
```

- es können auch Felder von Strukturen angelegt werden

```
struct M_Person it1[44];
```

- Separator für den Zugriff auf die Strukturvariablen ist der Punkt (.)

```
strcpy(person.name, "Meier");
person.geburtsjahr = 1980;
```

Zugriff auf Bestandteile einer Structure (2/2)

- auch bei Strukturen muss der Adressoperator (wenn Strukturvariable kein Feld ist) in der Funktion `scanf` angegeben werden!
- Adressoperator bezieht sich auf die Strukturvariable (hier `geburtsjahr`)

```
scanf("%d", &person.geburtsjahr);
```

- Zugriff auf einzelne Elemente eines Feldes von Strukturen erfolgt über einen Index vor dem Separator für die Strukturvariablen

```
personen[i].geburtsjahr = 1980;
```

Zeiger auf Strukturen

```
struct M_Person {
      char name[80+1];
      char vorname[80+1];
      int  geburtsjahr;
   } studenten[20], *ptrStudent;
ptrStudent = studenten;

/* … */

gets((*ptrStudent).name); /* gültige Schreibweise wie bisher;
                           * wird aber so nicht gemacht!
                           *
                           * Sondern: */

gets(ptrStudent->name);   /* mit dem „Pfeil“ wird angezeigt,
                           * dass eine Zeigervariable vor der
                           * Strukturvariablen steht */
```

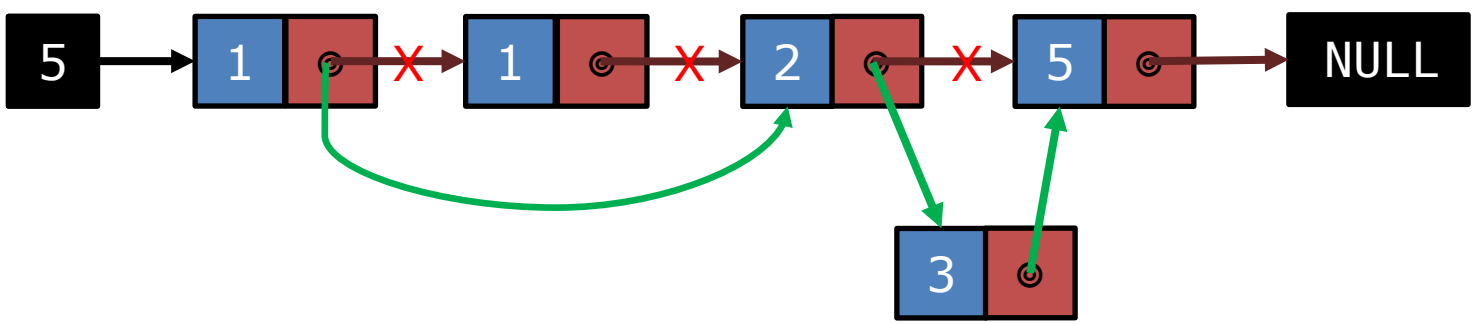
dynamische Datenstrukturen

DYNAMISCHE DATENSTRUKTUREN

- dynamische Datenstrukturen werden benötigt, um bedarfsgetrieben Speicher für Daten zu zuweisen
- Felder (Arrays) sind in C statisch, bzw. ab C99 semi-dynamisch.
- keine Standardfunktionen für dynamische Datenstrukturen (Stack, Baumstruktur, verkettete Liste, Hash-Tabelle, …) in C
- Programmpakete und Bibliotheken für viele solche Datenstrukturen
- Beispiel: doppelt verkettete Liste

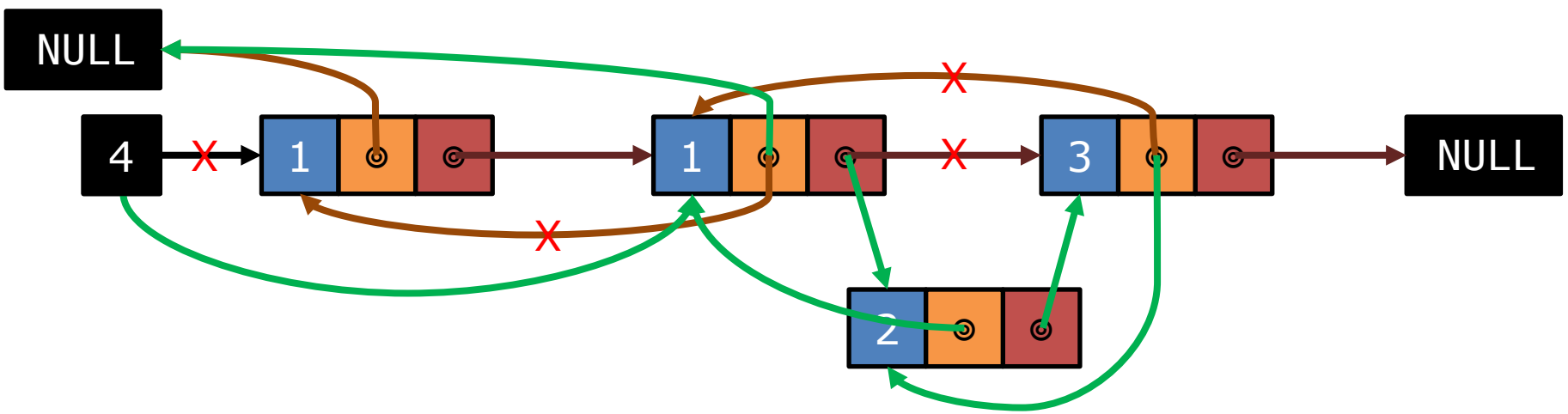
EINFACH VERKETTETE LISTE (EVL)

- dynamische Struktur mit Nachfolgerregelung
- Vorteile
  - Einfüge-Operationen haben Aufwand $\mathcal{O}(1)$
    (beim Array $\mathcal{O}(n)$, da Datensätze umkopiert werden müssen)
  - geringer zusätzlicher Speicherbedarf (1 Zeiger)
- Nachteile
  - Such-Kosten haben Aufwand $\mathcal{O}(n)$, da im WCS über jeden Knoten iteriert werden muss
- Listenanker kann Zusatzinformationen enthalten (Knotenanzahl, …); zeigt auf das erste Element

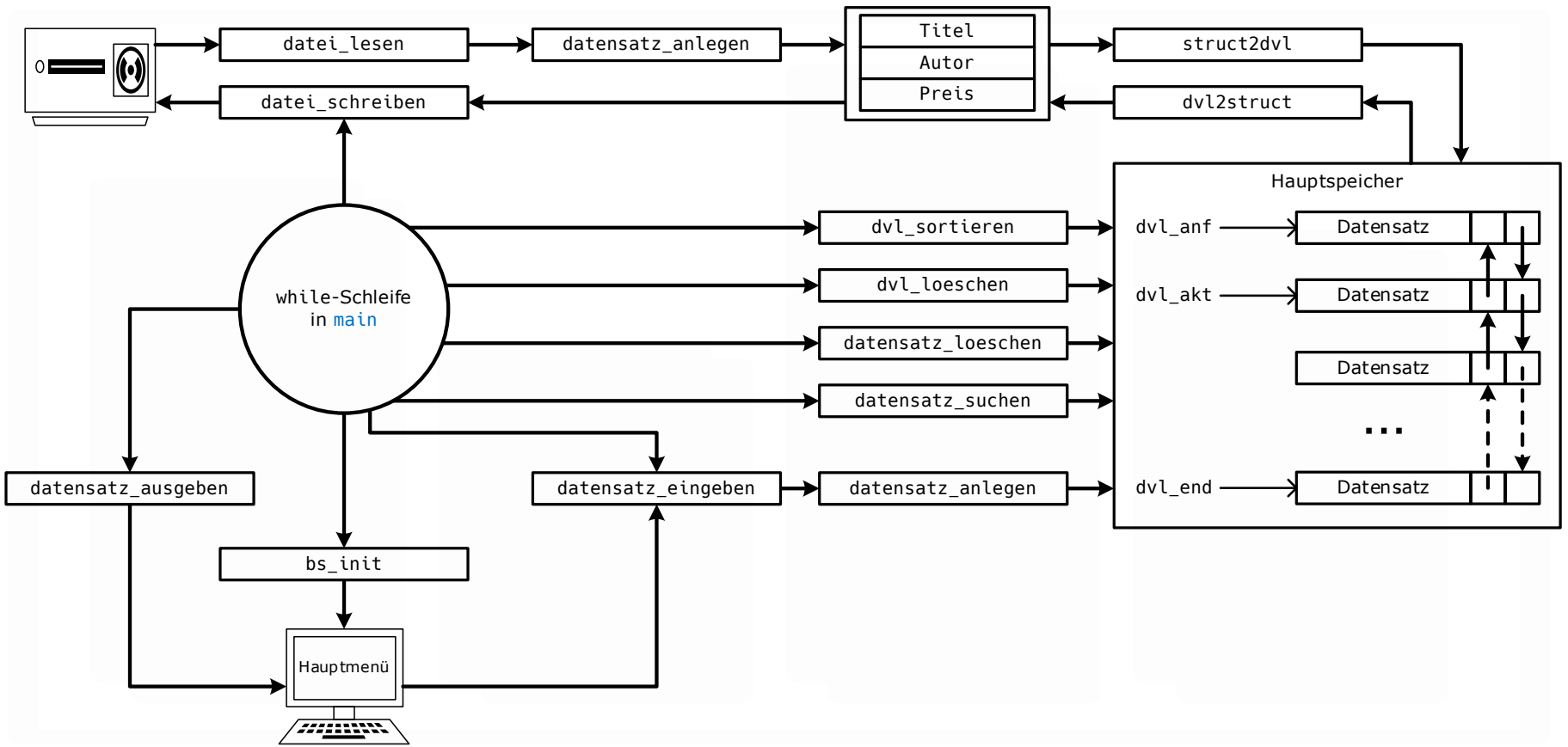


DOPPELT VERKETTETE LISTE (DVL)

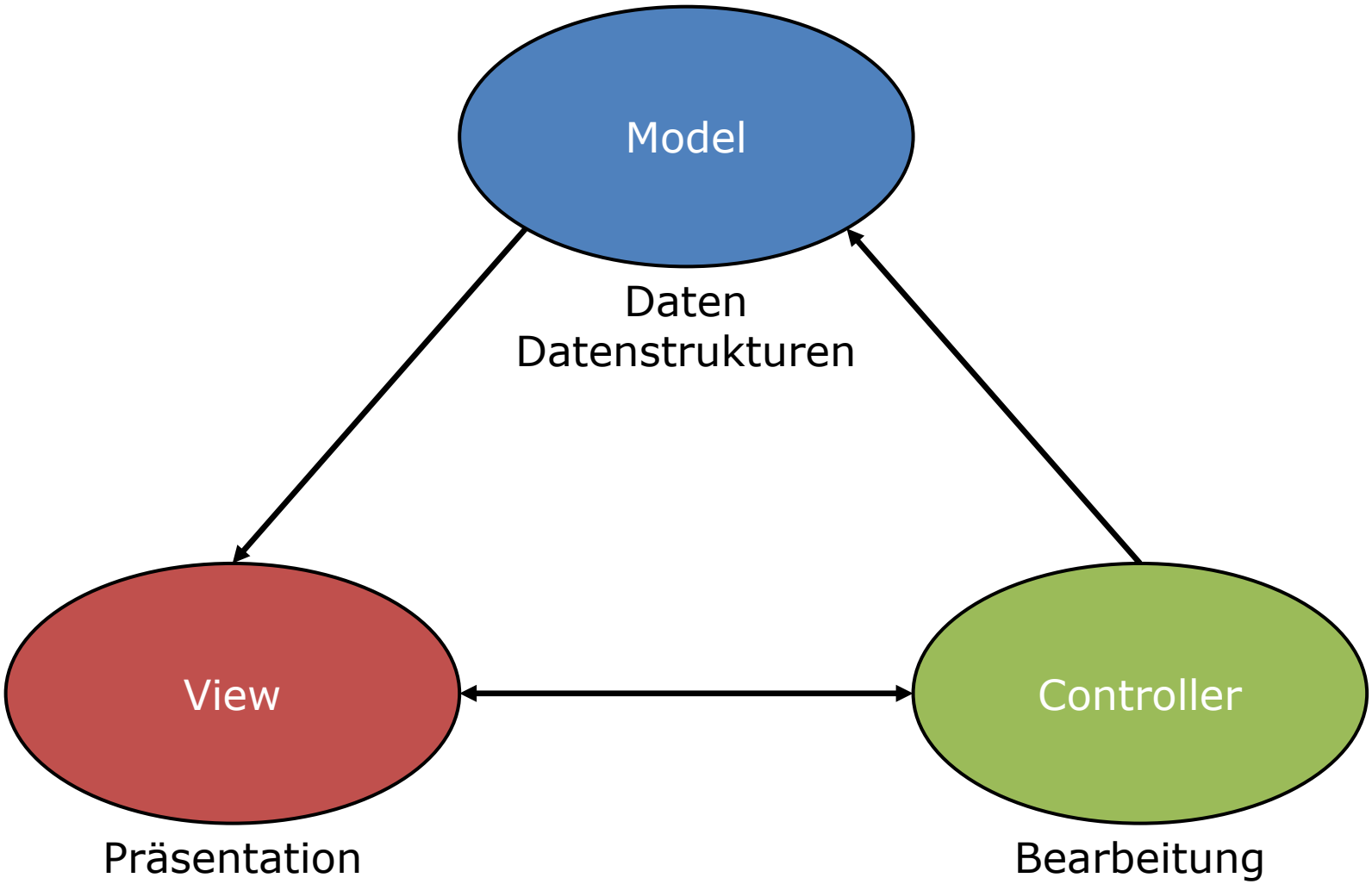
- dynamische Struktur mit Nachfolger- *und* Vorgängerregelung
- Vorteile
  - Entfernen eines Elements aus der Liste kann in $\mathcal{O}(1)$ geschehen
  - über die Liste kann von hinten nach vorne iteriert werden
- Nachteile
  - höherer Speicherbedarf für zusätzliche Zeiger
  - beim Löschen und Einfügen müssen auch die Vorgänger-Zeiger der nachfolgenden Listenelemente angepasst werden



Übersicht zur Mini-Datenbank



DAS KLASSISCHE DESIGN-PATTERN: MODEL-VIEW-CONTROLLER (MVC)



Anwendung des MVC

strenge Anwendung des MVC-Konzeptes bewirkt

- modularer Aufbau von Programmen
- bessere Aufteilung der Aufgaben (mehrere Programmierer)
- bessere Pflegbarkeit der Programme
- Austauschbarkeit von Funktionen

→ siehe später: Programmierprojekte

Typedefinition für DVL

```
struct M_Buch { // Muster/Modell
   char   titel[80+1];
   char   autor[80+1];
   float  preis;
   struct M_Buch *davor;
   struct M_Buch *danach;
};

typedef struct M_Buch T_Buch; // Name des Typs

// globale Daten (in allen Funktionen bekannt)
T_Buch *dvl_anf = NULL; // Zeiger auf DVL-Anfang
T_Buch *dvl_end = NULL; // Zeiger auf DVL-Ende
T_Buch *dvl_akt = NULL; // Zeiger auf aktuellen DS
```

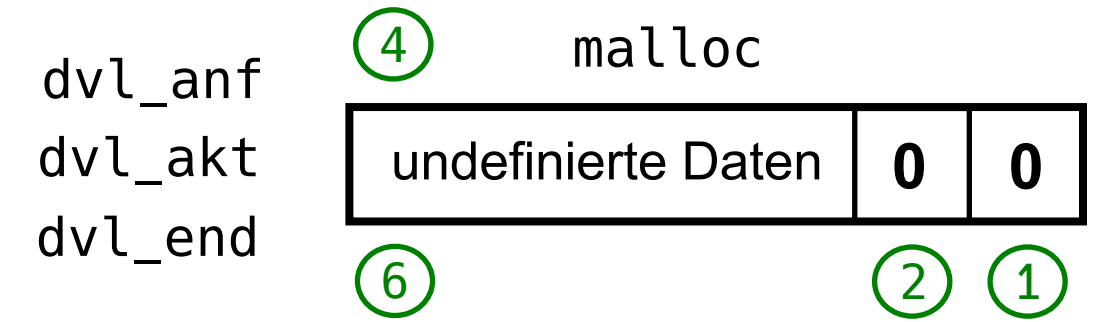
Anlegen eines Datensatzes (DS) (1/2)

```
int ds_anlegen(void) { // Datensatz wird an das Ende angehängt
    if (!(dvl_akt = (T_Buch *) malloc(sizeof(T_Buch)))) {
        fehlerAusgabe("ds_anlegen", -2);
        return -2;
    }

    dvl_akt->danach = NULL;          // 1
    dvl_akt->davor  = dvl_end;       // 2

    if (dvl_anf == NULL)             // 3
        dvl_anf = dvl_akt ;          // 4
    else
        dvl_end->danach = dvl_akt; // 5
    dvl_end = dvl_akt;               // 6

    return 0;
}
```

Anlegen eines Datensatzes (DS) (2/2)

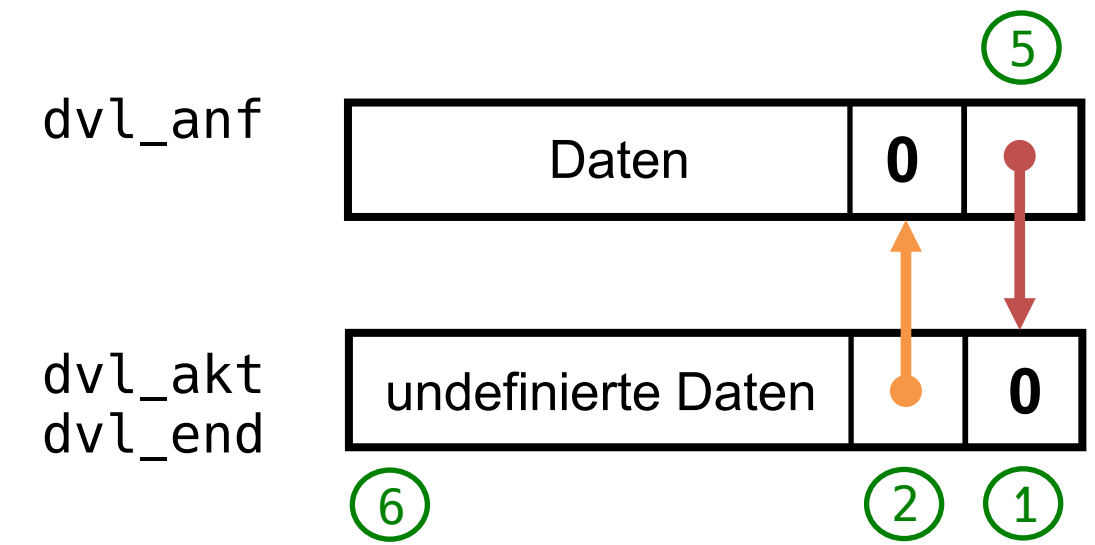
1. Fall: noch keine DVL vorhanden

```
dvl_anf = NULL
dvl_end = NULL
dvl_akt = NULL
```



2. Fall: ein Datensatz in der DVL vorhanden

```
dvl_anf = dvl_end
dvl_akt = dvl_anf
```



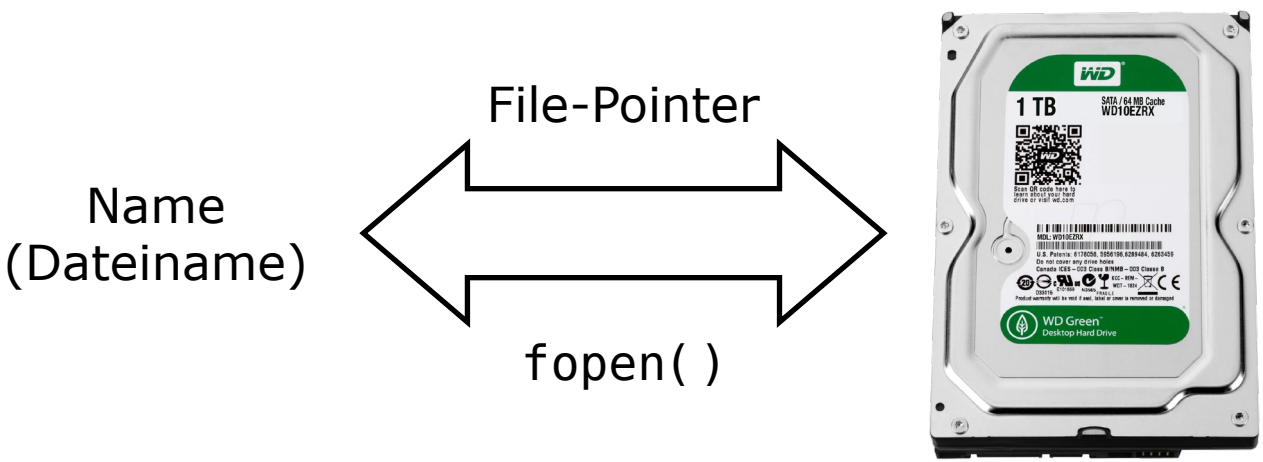
3. Fall: mehrere Datensätze in DVL vorhanden

⇒ wie 2. Fall!

Dateiarbeit

DATEIARBEIT

- jede Ein- und Ausgabe (Dateisystem, Sockets, Peripherie, …) ist in C Dateiarbeit (genauer: Arbeit an der Datei-Schnittstelle)
- keinerlei Datensatzstrukturen (Records)
  → alles muss durch den Programmierer selbst angelegt werden
- File-Pointer stellt die Beziehung zwischen Namen und Speicherort



FILE-POINTER

Typdefinition in `stdio.h` (in etwa so oder so ähnlich):

```
typedef struct {
    int level;                  /* Datei besetzt      */
    unsigned flags;             /* Datei-Status       */
    char fd;                    /* Dateibeschreibung
                                   (File Descriptor) */
    unsigned char hold;         /* Zeichenpuffer      */
    int bsize;                  /* Puffergröße        */
    unsigned char *buffer;      /* Puffer             */
    unsigned char *curp;        /* aktiver Zeiger     */
    unsigned istemp;            /* temporäre Datei    */
    short token;                /* Gültigkeitsdauer   */
} FILE;                         /* Typname            */
```

DATEI UNFORMATIERT LESEN

```c
int datei_lesen(char *datei) {
   FILE *ein_fp; // File-Pointer

   if ((ein_fp = fopen(datei, "rb")) == NULL) return -3;

   while (!feof(ein_fp)) { // Test auf Dateiende
      fread((char *) &buch, sizeof(buch), 1, ein_fp);
      ds_anlegen(); // Datenstruktur für DVL anlegen
      temp2dvl();   // Datenstruktur mit Daten füllen
   }
   dvl_aktuell = dvl_anfang;

   fclose(ein_fp); // Datei schließen
   return 0;
}
```

DATEI UNFORMATIERT SCHREIBEN

```c
int datei_schreiben(char *datei) {
   FILE *aus_fp; // File-Pointer

   if ((aus_fp = fopen(datei, "wb")) == NULL) return -5;

   dvl_aktuell = dvl_anfang;

   while (dvl_aktuell) { // solange nicht Ende der DVL-Datei
      dvl2temp(); // Datenstruktur aus DVL ohne Zeiger
      if (!fwrite((char *) &buch, sizeof(buch), 1, aus_fp))
         return -6;
      dvl_aktuell = dvl_aktuell->danach;
   }
   dvl_aktuell = dvl_anfang;

   fclose(aus_fp); // Datei schließen
   return 0;
}
```

BEFEHLSÜBERSICHT

	Op.-Typ	Bildschirm E/A	Datei E/A	System E/A	String E/A
Eingabe	*formatiert*	scanf	**f**scanf	read	**s**scanf
		gets	**f**gets		
		getchar	**f**getchar		
		getc	**f**getc		
	unformatiert		fread		
			fopen	open/create	
			freopen		
Ausgabe	*formatiert*	printf	**f**printf	write	**s**printf
		puts	**f**puts		
		putchar	**f**putchar		
		putc	**f**putc		
	unformatiert		fwrite		
		perror	clearerr		
			fclose	close	
			ferror	remove	
			feof	rewind	
Positionierung			ftell/fseek	rename	
			fgetpos		
			fsetpos		

Zeitfunktionen

ZEITFUNKTIONEN

Grundlage der Datumsformate ist die Structure `tm` aus `<time.h>`

```
struct tm {
   int tm_sec;   // Sekunden
   int tm_min;   // Minuten
   int tm_hour;  // Stunde (0-23)
   int tm_mday;  // Monatstag (1-31)
   int tm_mon;   // Monat (0-11; 0..Januar)
   int tm_year;  // Jahr (Kalenderjahr minus 1900)
   int tm_wday;  // Wochentag (0-6; 0..Sonntag)
   int tm_yday;  // Jahrestag (0-365)
   int tm_isdst; // 0..Normalzeit, 1..Sommerzeit
};
```

Funktion für deutsches Datumformat

```
char * zeit(char *datum) {
   time_t sekunden;
   struct tm *zeit; // tm aus time.h
   const char tag[7][2+1] = {"So","Mo","Di","Mi","Do","Fr","Sa"};
   const char monat[12][3+1] = {"Jan","Feb","Mar","Apr",
      "Mai","Jun","Jul","Aug","Sep","Okt","Nov","Dez"};
   time(&sekunden); // Sekunden seit 01.01.1970 (Unix-Timestamp)

   zeit = localtime(&sekunden); // Füllen der Struktur tm

   sprintf(datum, "%s,%2d. %s %4d %02d:%02d",
      tag[zeit->tm_wday],
      zeit->tm_mday, monat[zeit->tm_mon],
      zeit->tm_year + 1900, zeit->tm_hour,
      zeit->tm_min);
   return datum;
}
```