Dynamische Datenstrukturen in C

DI Thomas Helml

SEW

SJ 2017/18

Inhaltsverzeichnis

Motivation

- Nachteil statischer Speicher (Arrays)
  - Man weiß vorher nicht, wie viel Speicher benötigt wird
  - Speicherplatzverschwendung
  - Gültigkeit des Arrays: Anweisungsblock
- Lösung:
  - Dynamische Speicherverwaltung (zur Laufzeit)

Speicher reservieren

- Heap (Halde)
  - Speicher, der zur Laufzeit reserviert werden kann
  - man erhält immer zusammenhängenden Bereich
- Reservierung von Speicher mittels (`stdlib.h`):
  - `malloc()`
  - `calloc()`

Speicher reservieren

- Syntax:

```
void *malloc(size_t size);
```

- `malloc()` (memory allocation) liefert size Bytes zusammenhängenden Speicher
- Return-Wert:
  - `NULL` bei Fehler, sonst
  - Zeiger auf Anfangsadresse des res. Speicherblocks

Speicher reservieren

- Beispiel: Platz für 100 int reservieren

```
int *iptr;
iptr = malloc (400); // 400 Byte Speicher reservieren
...
```

- Vorsicht! int-kann unterschiedl. Größe haben
- Besser:

```
iptr = malloc (100*sizeof(int));
iptr = malloc (100*sizeof(*iptr));
```

Speicher reservieren

- Syntax `calloc` (core allocation):

```
void *calloc(size_t n, size_t size);
```

- Parameter:
  - `n`: Anzahl an zu reservierenden Objekte
  - `size`: Größe eines Objekts
- `calloc()` initialisiert den Speicher mit 0!
- Beispiel:

```
iptr = calloc (100, sizeof(int));
```

Speicher freigeben

- ACHTUNG:
  - Jeder Speicherblock der reserviert wurde, muss auch freigegeben werden!
  - Freigabe erfolgt anders wie in Java NICHT automatisch
  - Fehlende Freigabe führt zu Speicherlöchern – sogenannten Memory Leaks

Speicherblock freigeben

- Syntax:

```
#include <stdlib.h>
void free(void *ptr);
```

- der Speicherbereich, der mit `malloc/calloc` allokiert wurde, wird freigegben
- ist `ptr` ein `NULL`-Pointer, passiert nichts
- ist `ptr` kein/ein falscher Zeiger => undefiniertes Verhalten
- `ptr` sollte nach Freigabe wieder auf `NULL` gesetzt werden

Speicher freigeben / Beispiel

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

// reserviert einen Speicherbereich für n-INT Werte
// und gibt einen Zeiger darauf zurück
int *meinArray(unsigned int n) {
    int *iptr = NULL;
    int i = 0;

    iptr = malloc (n*(sizeof(int)));
    if (iptr!=NULL)
        for (i=0; i<n; i++)
            iptr[i] = i*i; // *(iptr+i) = ...;
    return iptr;
}
```

Speicher freigeben / Beispiel

```
int main () {
    int *arr = NULL;
    unsigned int val=0, i=0;

    printf ("Wie viele int-Elemente benötigen Sie? "); fflush(stdout);
    scanf("%u", &val);

    arr = meinArray(val);
    if (arr==NULL) {
        printf ("nicht genügend speicher");
        return -1;
    }
    printf ("Ausgabe der Elemente\n");
    for (i=0;i<val;i++) printf ("arr[%d] = %d\n", i, arr[i]);
    free(arr);
    arr=NULL;
    return 0;
}
```

Speicherblock vergrößern/verkleinern

- Größenänderung mit `realloc()` möglich
- Syntax

```
void *realloc (void *ptr, size_t size);
```

- der durch ptr adressierte Speicherbereich
  - wird freigeben
  - der ursprüngliche Block bleibt erhalten
  - falls möglich wird der neue Block hinten angehängt
  - sonst wird der gesamte Block umkopiert
- Rückgabewert:
  - Im Fehlerfall: NULL
  - Sonst wird ein Zeiger auf den Speicherblock mit size Byte Größe

Speicherblock vergrößern/verkleinern

- Wird für `ptr` ein `NULL` Zeiger verwendet, so funktioniert `realloc()` wie `malloc()`
- Folgende Aufrufe sind somit ident:

```
ptr = malloc(100*sizeof(int));
ptr = realloc(NULL, 100*sizeof(int));
```

Speicherblock vergrößern/verkleinern

- Verkleinern des Speichers:
  - für `size` kleinere Größe als ursprünglich angenommen:

```
// Speicher für 100 int-Elemente reservieren
ptr = malloc(100*sizeof(int));
...
// Speicher auf 50 int-Elemente verkleinern
ptr = realloc(ptr, 50*sizeof(int));
```

Speicherblock vergrößern/verkleinern

- Vergrößern des Speichers:
  - für `size` muss Größe angegeben werden, die das alte `size` beinhaltet
  - Folgendes Beispiel ist falsch!

```
int block = 256;
ptr = malloc(block  * sizeof(int));
...
// Hier wird kein neuer Speicher reserviert
// es wird nur Speicher für 256 int-Element reserviert
ptr = realloc(ptr, block*sizeof(int));
```

Speicherblock vergrößern/verkleinern

- Vergrößern des Speichers:
  - Korrektes Beispiel:

```
int block = 256;
ptr = malloc(block  * sizeof(int));
...
block += block;
// Speicher für 512 int-Elemente reservieren
ptr = realloc(ptr, block*sizeof(int));
```

Einfach Verkettete Listen

- Beispiel: Struktur für „Namensliste“ (WH)

```
#define MAX_LEN 255

typedef struct data {
   char name[MAX_LEN];
   char vorname[MAX_LEN];
}DATA;
```

- damit lässt sich 1 Datensatz speichern
- Wie kann ich mehrere speichern?

Einfach Verkettete Listen

- Möglichkeit 1:

```
DATA dataArr[MAX];
```

- Nachteil:
  - Limitierung!

Einfach Verkettete Listen

- Möglichkeit 2:

```
DATA *d = NULL;
d = malloc (sizeof(DATA));
```

  - Speicherplatz wird dynamisch reserviert
  - ABER nur für 1 Datensatz
  - Zeiger muss man sich merken!

```
DATA *d[MAX];
d[i] = malloc (sizeof(DATA));
```

  - Nachteil: kompliziert + limitiert!

Einfach Verkettete Listen

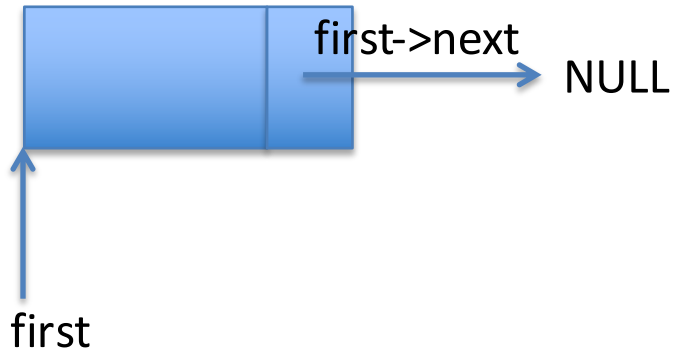
- Annahme:
  - Reihe von Strukturvariablen dynamisch erzeugen
  - Wie können wir uns alle Zeiger merken?
- Lösung: Verketten
- In der Struktur Zeiger auf nächste Struktur

```
typedef struct data {
   char name[MAX];
   char vorname[MAX];
   struct data *next;
}DATA;
```

Einfach Verkettete Listen

- Anlegen eines neuen Elements

```
DATA *first = NULL;
first = malloc (sizeof(DATA));
if (first!=NULL)
   first->next = NULL;
```

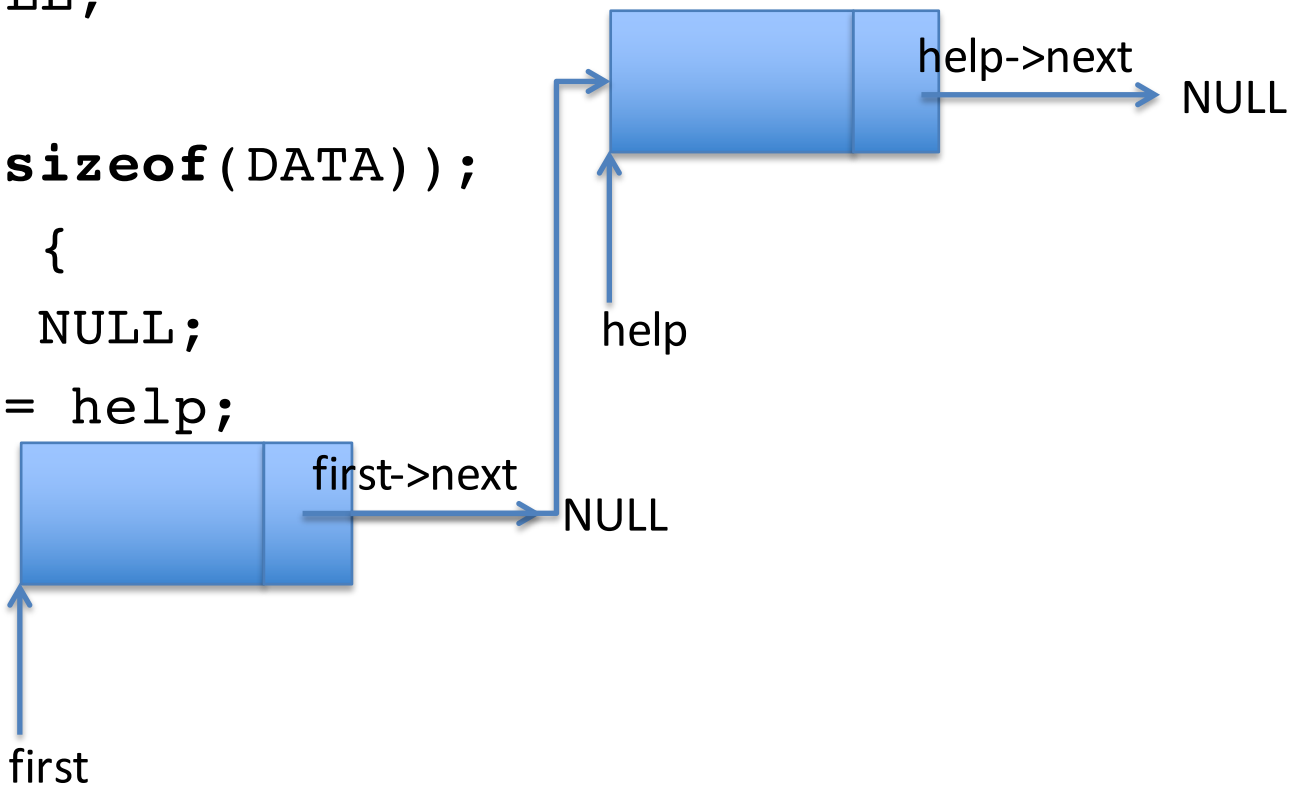


Verkettete Listen

- Anlegen 2. Elements + Anhängen an 1. Element

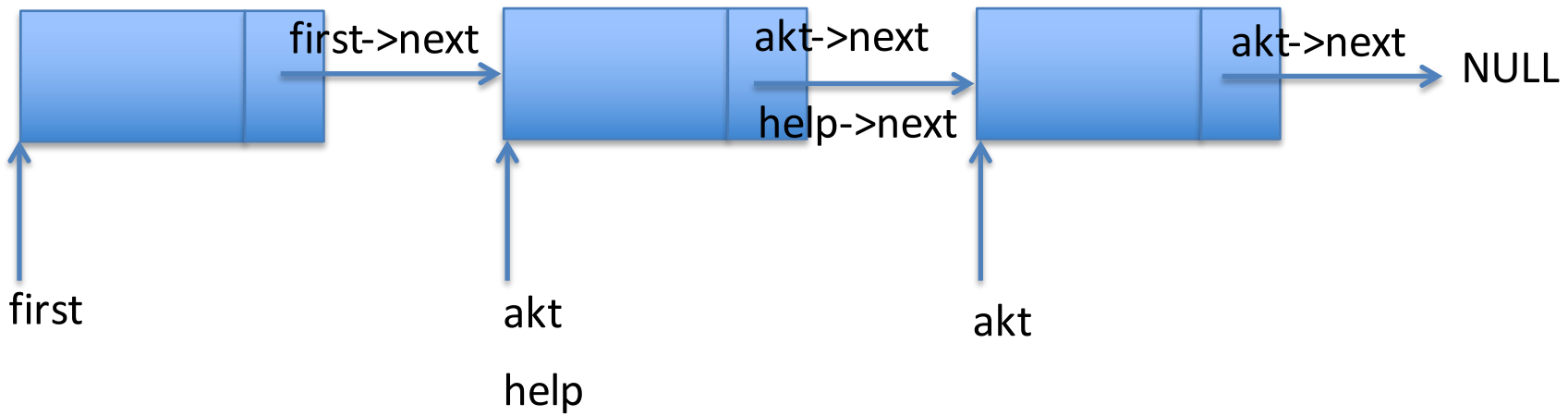
```
DATA *help = NULL;

help = malloc (sizeof(DATA));
if (help!=NULL) {
   help->next = NULL;
   first->next = help;
}
```

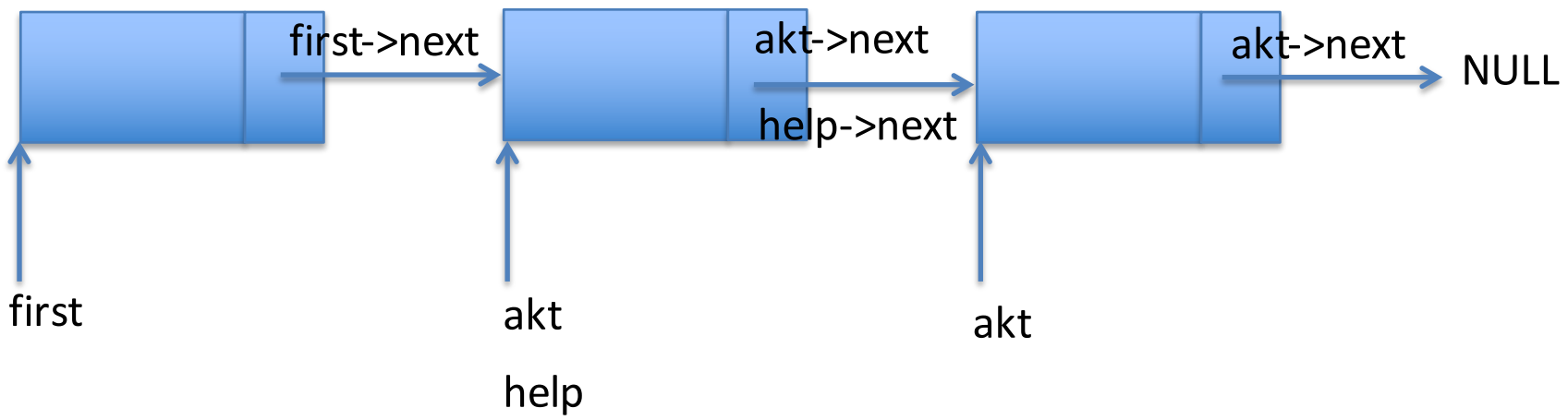


Verkettete Listen

- Für jedes weitere Element:
  - Zeiger auf aktuelles (letztes) Element merken



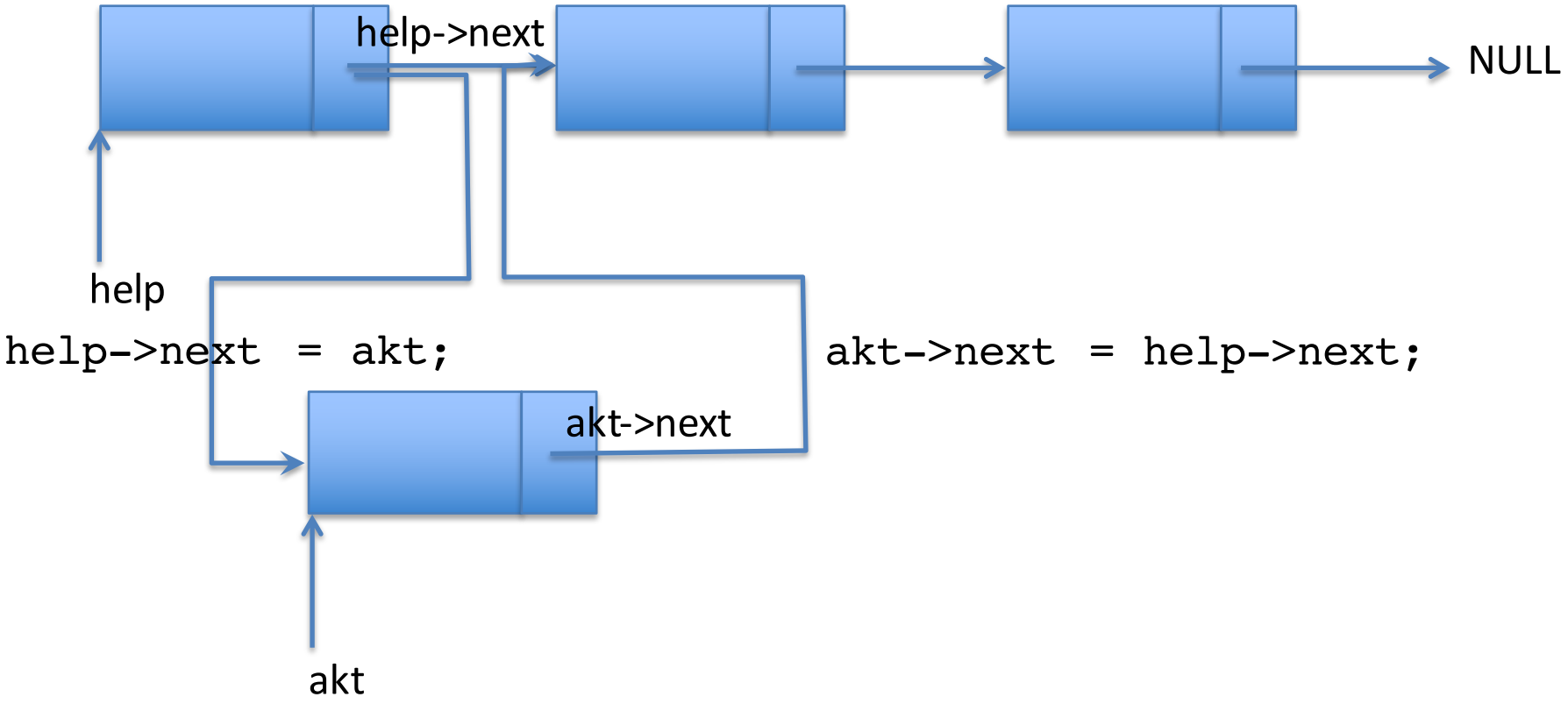
Einfach Verkettete Listen



- first zeigt immer auf erstes Element
  - Verlust von first -> Kette verloren
- help zeigt auf vorletztes Element
- akt zeigt auf aktuelles/zuletzt hinzugefügtes Element

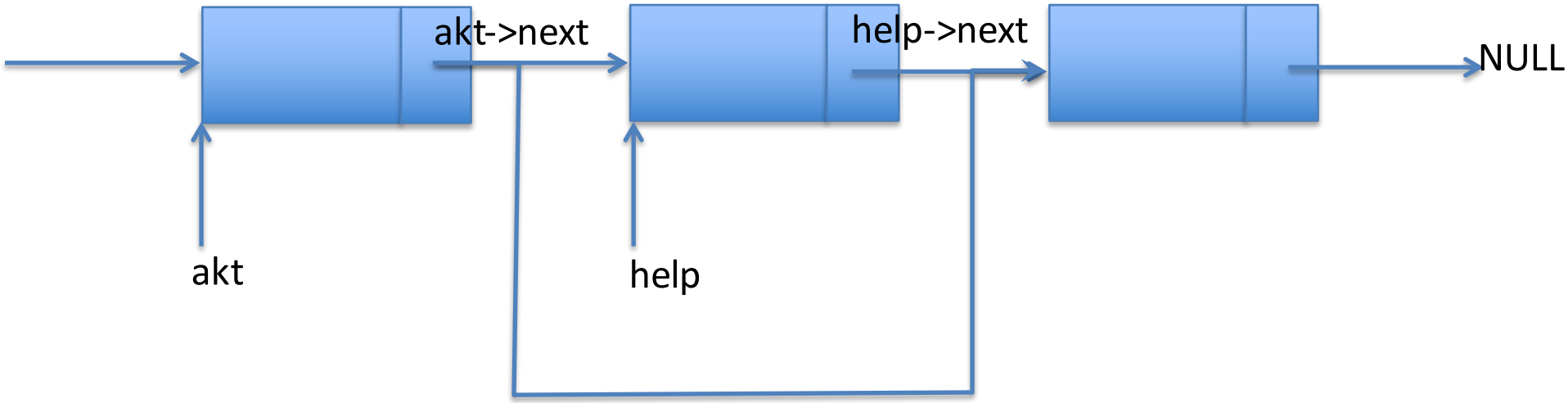
Einfach Verkettete Listen

- Hinzufügen eines Elements



Einfach Verkettete Listen

- Löschen eines Elements




```
akt->next = help->next;
free (help);
```

- VORSICHT: Löschen des ersten Elements!

Einfach Verkettete Listen

- Beispiel

```
typedef struct data
{
    char name[MAX_LEN];
    char vorname[MAX_LEN];
    struct data *next;
}DATA;
```


```
DATA *AddElementEnd(char *name, char *vorname, DATA *last);
DATA *AddElementBegin(char *name, char *vorname, DATA *first);
DATA *DeleteElement(char *name, DATA *first);
void PrintAll (DATA *elem);
void FreeAll (DATA *elem);
```