Systemnahe Programmierung in C (SPiC)

33 Dynamische Speicherallokation

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2020

http://www4.cs.fau.de/Lehre/SS20/V_SPIC

Größe von Typen/Objekten

- Größe elementarer Typen bekannt; z.B.:
  - `char`: 1 Byte
  - `uint16_t`: 2 Byte
  - `uint32_t`: 4 Byte
  - ...
- Größe von Datenstrukturen:
  - **Felder**: *N*-elementiges Array braucht *N*-mal den Platz eines Elements
  - **Strukturen**: Struktur braucht (mindestens) den Platz aller Elemente
  - **Unions**: Union braucht den Platz des größten Elements

Größe ermittelbar mit:

```
sizeof type
```

bzw.

```
sizeof var
```

`sizeof`-Operator liefert Wert vom Typ `size_t`.

Dynamische Speicherallokation: Heap

- **Heap** := Vom Programm explizit verwalteter RAM-Speicher
  - Lebensdauer ist unabhängig von der Programmstruktur
- Anforderung und Wiederfreigabe über zwei Basisoperationen
  - `void *malloc(size_t n)` fordert einen Speicherblock der Größe n an; Rückgabe bei Fehler: `NULL`-Zeiger
  - `void free(void *pmem)` gibt einen zuvor mit `malloc()` angeforderten Speicherblock vollständig wieder frei

Dynamische Speicherallokation: Heap

- **Heap** := Vom Programm explizit verwalteter RAM-Speicher
  - Lebensdauer ist unabhängig von der Programmstruktur
- Anforderung und Wiederfreigabe über zwei Basisoperationen
  - `void *malloc(size_t n)` fordert einen Speicherblock der Größe n an; Rückgabe bei Fehler: `NULL`-Zeiger
  - `void free(void *pmem)` gibt einen zuvor mit `malloc()` angeforderten Speicherblock vollständig wieder frei
- Beispiel

```c
#include <stdlib.h>

int *intArray(size_t n) { /* alloc int[n] array */
    return (int *) malloc(n * sizeof int);
}

void main(void) {
    int *array = intArray(100); /* alloc memory for 100 ints */
    if (array == NULL) { /* error handling... */ }
    ...
    array[99] = 4711; /* use array */
    ...
    free(array); /* free allocated block (** IMPORTANT! **) */
}
```


Dynamische Speicherallokation: Verkette Liste

Beispiel: Allozieren eines Listenelementes und Einfügen in Liste:

```
struct list_elem {
    struct list_elem *next;
    int num;
}
struct list_elem *head = NULL;

void add_to_list(int num) {
    struct list_elem *elem;

    /* Allocate memory for element. */
    elem = (struct list_elem *) malloc(sizeof(*elem));
    if (elem == NULL) { /* Error handling... */ }

    /* Fill object. */
    elem->num = num;

    /* Add element to list. */
    elem->next = head;
    head = elem;
}
```

Dynamische Speicherallokation: Verkette Liste (Forts.)

Beispiel: Herausnehmen eines Listenelementes aus Liste und Freigeben:

```
int remove_from_list(void) {
    /* Get element. */
    struct list_elem *elem = head;

    if (elem == NULL) {
        return -1; /* List empty. */
    }

    /* Remove element from list. */
    head = elem->next;

    /* Get info from element. */
    int num = elem->num;

    /* Free memory of element. */
    free(elem);

    return num;
}
```


Dynamische Speicherallokation: Strings

Beispiel: Duplizieren eines Strings:

```
char *strdup(const char *s) {
    /* Calculate size of string. */
    /* ** IMPORTANT **: "+ 1" for '\0' at end! */
    size_t size = strlen(s) + 1;

    /* Allocate memory. */
    char *p = (char *) malloc(size * sizeof(char));
    if (p == NULL) {
        return NULL; /* Out of memory. */
    }

    /* Copy string. */
    strcpy(p, s);

    return p;
}
```