System-Level Programming

33 Dynamic Allocation of Memory

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Summer Term 2024

http://sys.cs.fau.de/lehre/ss24

Size of Types and Objects

- Size of standard types is known; e.g.:
  - `char`: 1 byte
  - `uint16_t`: 2 bytes
  - `uint32_t`: 4 bytes
  - ...
- Size of data structures:
  - **Arrays**: An array of N elements needs N times the space of one element
  - **Structs**: Structs need (at least) the space of all elements combined

Their size can be determined:

```
sizeof type
```

or

```
sizeof var
```

`sizeof`-operator returns a value of type `size_t`.

Dynamic Allocation of Memory: Heap

- **Heap** := RAM memory that is explicitly used by the program
  - lifespan independent of program structure
- Allocation and freeing via two basic operations
  - `void *malloc(size_t n)` — allocates a memory area of size n; returns `NULL` pointer on error
  - `void free(void *pmem)` — frees a previously with `malloc()` allocated memory area

Dynamic Allocation of Memory: Heap

- **Heap** := RAM memory that is explicitly used by the program
  - lifespan independent of program structure
- Allocation and freeing via two basic operations
  - `void *malloc(size_t n)` allocates a memory area of size n; returns `NULL` pointer on error
  - `void free(void *pmem)` frees a previously with `malloc()` allocated memory area
- Example

```
#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! **) */
}
```


Dynamic Allocation of Memory: Linked Lists

Example: Allocation and insertion of a list element into a list:

```
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;
}
```

Dynamic Allocation of Memory: Linked Lists (continued)

Example: Removing and freeing of a list element:

```
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;
}
```


Dynamic Allocation of Memory: Strings

Example: Duplicating a string:

```
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;
}
```