

## Programmieren in C für Elektrotechniker

# Kapitel 8: Dynamischer Speicher

## Dynamische Speicherzuweisung

- **Statische Variablen** (→ nicht mit *static* in C verwechseln)
  - Variablen, die in einem Programm vereinbart werden.
  - Sind über einen Bezeichner ansprechbar.
  - **Lebensdauer ist durch die statische Programmstruktur bestimmt**
    - **global**: während des gesamten Programmlaufs (→ *globaler Datenbereich*),
    - **lokal**: während der Ausführung eines Blocks (→ *Stack*).
- **Dynamische Variablen**
  - **Während des Programmlaufs allokiert/angelegt.**
  - Sind nicht über einen Namen ansprechbar (nur über einen Pointer).
  - Lebensdauer ist nicht durch die Programmstruktur bestimmt.
  - Liegen auf dem Heap.
- **Heap: Dynamischer Speicherbereich**
  - Variablen werden durch Funktionsaufrufe angelegt und gelöscht (in C: `malloc()`, `free()`).
  - Zugriff nur über Pointer möglich.
  - **Probleme**: - Überlauf, falls zu viel Speicher angefordert.  
- Zerstückelung bei wechselndem Anlegen und Löschen.

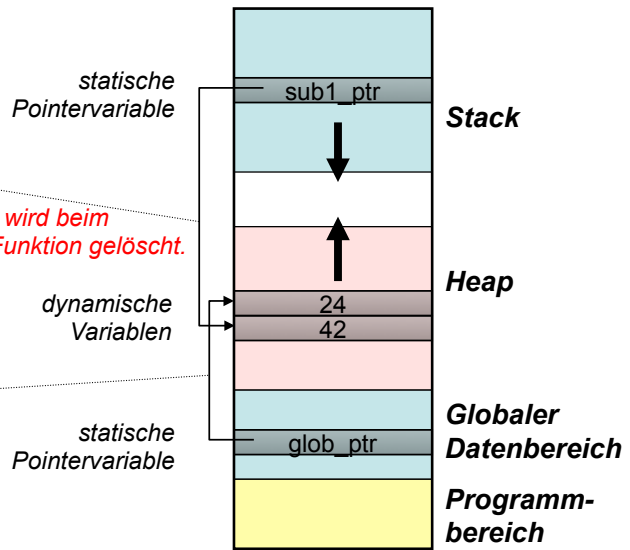
# Dynamische Speicherzuweisung

```
int * sub1 (void) {
    int * sub1_ptr;
    sub1_ptr = malloc (...);
    *sub1_ptr = 42;
    return sub1_ptr;
}

int * glob_ptr;

int main (void) {
    sub1();
    glob_ptr = malloc (...);
    *glob_ptr = 24;
    free(glob_ptr);
}
```

*Nur der Pointer wird beim Verlassen der Funktion gelöscht.*



## ▪ Reservierung von dynamischem Speicher

```
#include <stdlib.h>
void * malloc (size_t size);
```

- malloc → **m**emory **a**llocation
- Reserviert einen Speicherblock der Größe `size` auf dem Heap (bis er über `free ()` wieder frei gegeben wird).
- Rückgabe:
  - Pointer auf den angelegten Speicherblock (wird bei Zuweisung in einen Pointer auf einen Datentyp gewandelt)
  - **NULL**, falls Allokation nicht erfolgreich.
- Beispiel:

```
int main (void) {
    int * pointer;
    if ((pointer = malloc (sizeof (int))) != NULL) {
        *pointer = 3;
        printf ("pointer zeigt auf Wert %d\n", *pointer);
        free (pointer); /* s.u. */
    }
    else
        printf ("Nicht genug Speicher verfuegbar\n");
}
```

▪ **Änderung der Größe eines dynamischen Speicherblocks**

```
#include <stdlib.h>
void * realloc (void * memblock, size_t size);
```

- **memblock**: Pointer auf bisher reservierten Speicherblock.
- **size**: neue Speicherblockgröße.
- **Rückgabe**:
  - Pointer auf den Block geänderter Größe.
  - **NULL**, falls Größenänderung nicht erfolgreich.

• **Beispiel:**

```
int main (void) {
    int * pointer;
    if ((pointer = malloc (sizeof (int))) != NULL) {
        *pointer = 3;
    }
    else {
        printf ("Nicht genug Speicher verfuegbar\n");
        return (-1);
    }
    pointer = realloc ((void *)pointer, 2 * (sizeof(int)));
    if (pointer == NULL) {
        printf ("Nicht genug Speicher verfuegbar\n");
        return (-1);
    }
    pointer[1] = 6;
    free (pointer);
}
```

▪ **Freigabe von dynamischem Speicher**

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

- **pointer**: Referenz auf den freizugebenden Speicherblock.  
Der aktuelle Parameter kann von beliebigem Pointertyp sein.  
→ von `malloc()` bzw. `realloc()` gelieferter Pointer  
(sonst ist die Funktion undefiniert).
- **Achtung**: Wird ein Pointer auf ein gültiges Objekt im Heap überschrieben  
(→ keine Referenz mehr auf dieses Objekt),  
kann auf das Objekt nicht mehr zugegriffen werden.  
→ Speicher wird erst zum Programmende durch Betriebssystem frei.  
→ C besitzt keinen „Garbage Collector“.

• **Beispiel:**

```
int main (void) {
    char * ptr;
    char string[] = "Ab in den Heap";
    if ((ptr = malloc(strlen(string)+1)) != NULL) {
        strcpy (ptr, string);
        printf ("String im Heap: %s\n", ptr);
        free (ptr);
    }
    else
        printf ("Nicht genug Speicher verfuegbar\n");
}
```

▪ **Tipp zu Speicherlecks**

- Häufig wird vergessen, Speicher wieder freizugeben.  
→ Speicherlecks (memory leaks)
- Desktop-Betriebssysteme  
→ nicht ganz so tragisch, da Speicher zu Programmende durch Betriebssystem wieder freigegeben wird.
- Eingebettete Systeme  
→ besitzen oft eingeschränktes Betriebssystem, so dass der Speicher erst beim „Kaltstart“ wieder frei wird.
- **Speicheranforderung und Freigabe immer paarweise programmieren.**  
→ mit jedem `malloc()` auch gleich `free()` ins Programm einsetzen (zumindest die Notwendigkeit notieren).

▪ **Beispiel 1: Array beliebiger Größe anlegen**

- Größe eines Array zur Übersetzungszeit meist noch nicht bekannt.  
→ meist werden zur Vorsicht viel zu große Arrays angelegt.
- **Sinnvoller:** Array zur Laufzeit auf dem Heap anlegen, wenn dessen Größe bekannt ist.
- **Beispiel:**

```
int main (void) {
    int * ptrArray = NULL;
    int anzahl = 0, i;
    printf ("Anzahl der Array-Elemente: ");
    scanf ("%d",&anzahl);
    ptrArray = malloc (sizeof (int) * anzahl);
    for (i = 0; i < anzahl; i++) {
        ptrArray[i] = i;
        printf ("Array[%d]: %d\n", i, ptrArray[index]);
    }
    free (ptrArray);
    return 0;
}
```

▪ **Beispiel 2: Verkettete Listen**

- Lineare Liste beliebiger Größe.
- Elemente liegen nicht hintereinander im Speicher (*nicht wie beim Array*).
- Es können jederzeit beliebig viele neue Elemente angelegt werden.

