Dynamische Speicherverwaltung

>> Bisher haben wir Zeiger nur auf *eine* Art verwendet: Wir haben mit ihnen auf Variablen gezeigt, die bereits in unserem System existieren.

Das bedeutet, wir müssen beim Kompilieren schon genau wissen, wie viel Speicher unser Programm benötigen wird.

Was aber, wenn wir die benötigte Speichermenge erst während der Programmausführung erfahren?

Wie bekommen wir dann Zugriff auf zusätzlichen Speicher?

Dynamische Speicherverwaltung

Wir können Zeiger verwenden, um zur Laufzeit auf *dynamisch allokierten Speicher* zuzugreifen.

Dieser dynamisch allozierte Speicher kommt aus einem Speicherbereich, den man **Heap** nennt.

Bisher haben wir ausschließlich mit Speicher aus einem Bereich gearbeitet, den man **Stack** nennt.

Speicher-Layout eines Programms

Dynamische Speicherverwaltung

Um dynamisch allokierten Speicher zu erhalten, verwenden wir die C-Standard-bibliotheksfunktion **malloc()**, der wir als Parameter die gewünschte Anzahl an Bytes übergeben.

Nach erfolgreicher Speicheranforderung gibt *malloc* einen Pointer auf diesen Speicherbereich zurück.

Falls *malloc* keinen Speicher bereit-stellen kann, gibt es NULL zurück.

Statische versus dynamische Allokation von Speicher

```c
// Integer statisch anlegen (auf dem Stack)
int x;


// Integer dynamisch anlegen (im Heap)
int *px = malloc(4);


// besser: sizeof verwenden
int *px = malloc(sizeof(int));
```

Statische und dynamische Allokation von Arrays variabler Größe

```c
// Integer vom User einlesen
int x = get_int();


// Array von Floats auf dem Stack
float stack_array[x];


// Array von Floats auf dem Heap
float* heap_array = malloc(x * sizeof(float));
```

Zu beachten bei dynamisch allokiertem Speicher

Wichtig: Dynamisch allokierter Speicher wird **nicht** automatisch freigegeben, wenn die Funktion endet, in der er angefordert wurde.

Wenn Sie den Speicher nicht explizit freigeben, führt das zu einem **Memory Leak**, was die Systemleistung beeinträchtigen kann.

Wenn Sie mit dynamisch alloziertem Speicher fertig sind, müssen Sie ihn mit **free()** freigeben.

Zu beachten bei dynamisch allokiertem Speicher

```c
char* word = malloc(50 * sizeof(char));


// mit word arbeiten


// fertig - free nicht vergessen!
free(word);
```

Drei Goldene Regeln

1. Jeder Speicherblock, den Sie mit **malloc()** anfordern, muss später mit **free()** freigegeben werden.

2. Nur Speicher, den Sie mit **malloc()** angefordert haben, darf mit **free()** freigegeben werden.

3. Geben Sie einen Speicherblock nicht mehr als einmal mit **free()** frei.

m

```
int m;
```

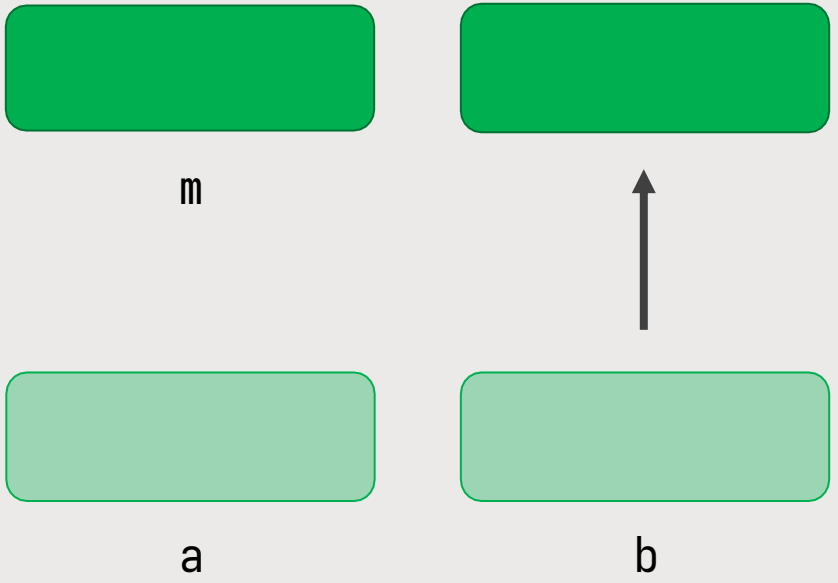
```
int m;
int* a;
```

m

a

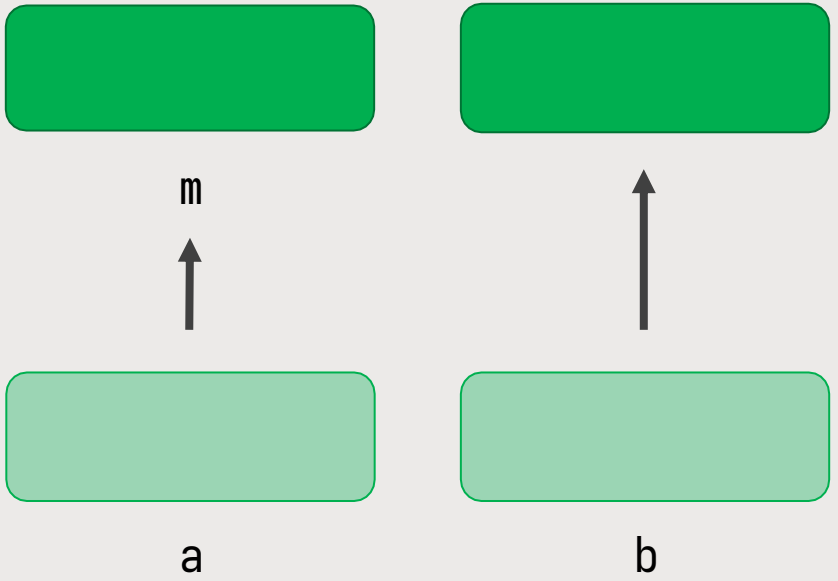
```c
int m;
int* a;

int* b = malloc(sizeof(int));
```



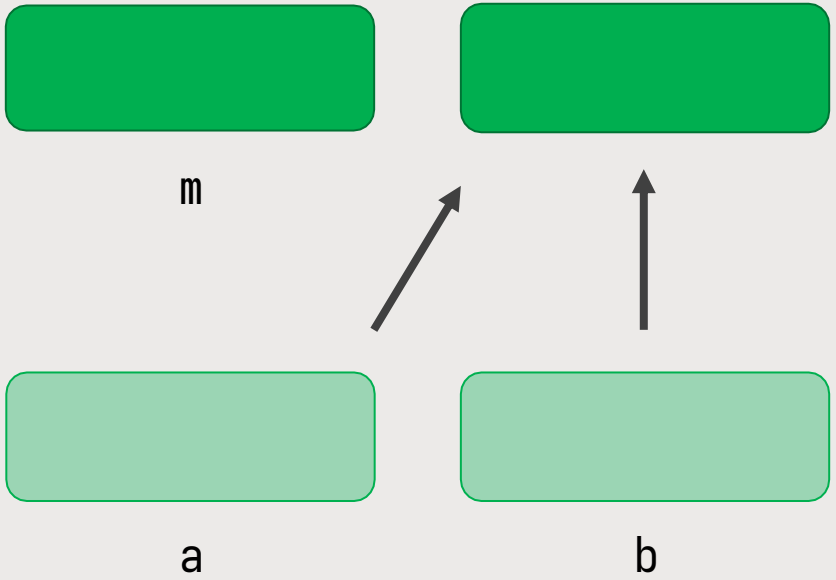
```c
int m;
int* a;

int* b = malloc(sizeof(int));
a = &m;
```



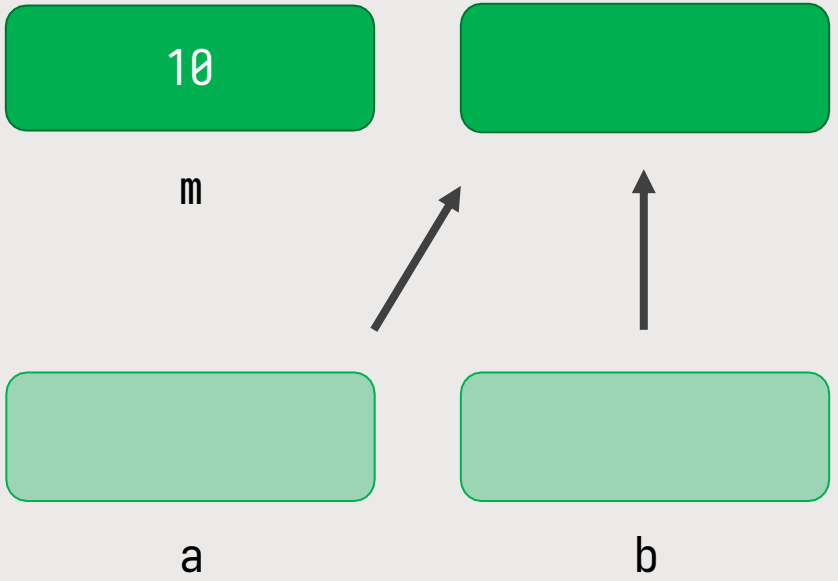
```c
int m;
int* a;

int* b = malloc(sizeof(int));
a = &m;
a = b;
```



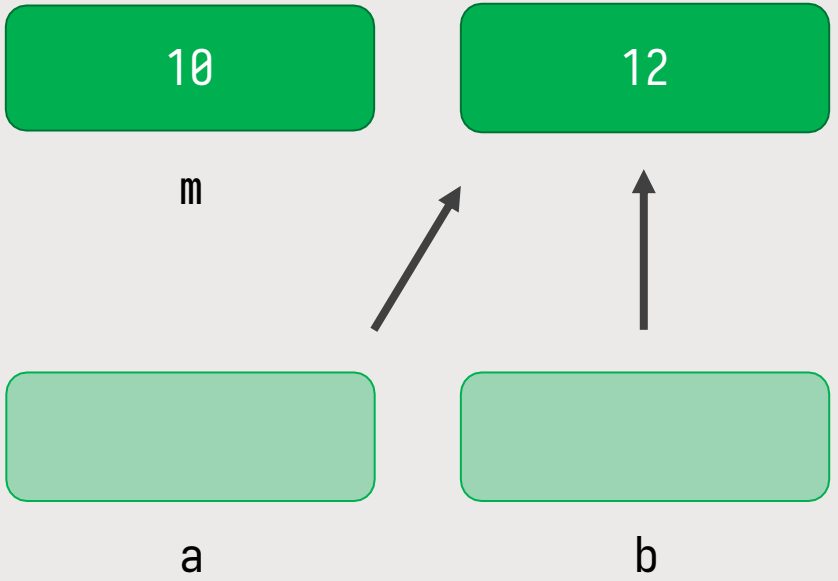
```c
int m;
int* a;

int* b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
```



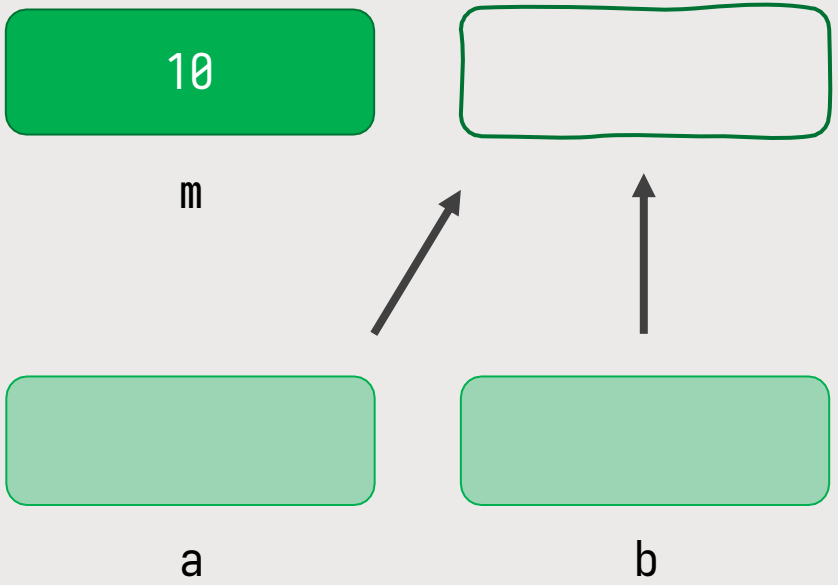
```c
int m;
int* a;

int* b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
*b = m + 2;
```



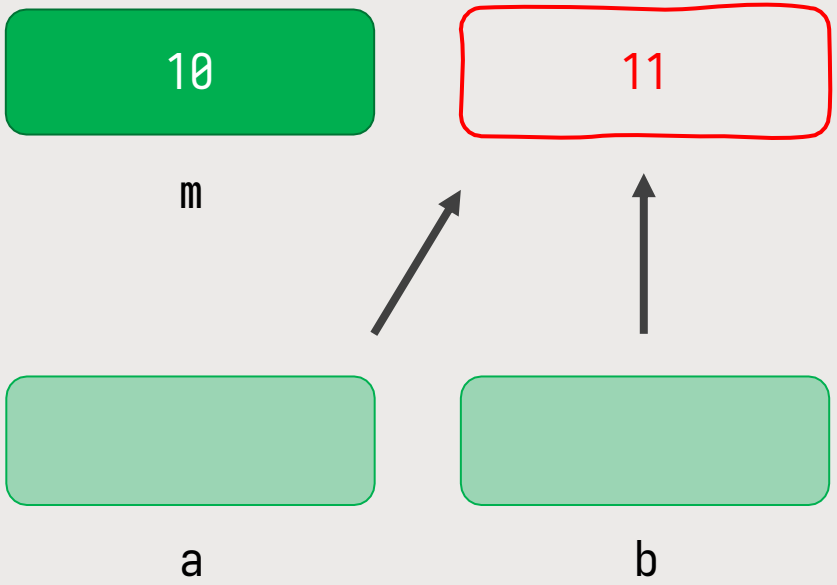
```c
int m;
int* a;

int* b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
*b = m + 2;
free(b);
```



```c
int m;
int* a;

int* b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
*b = m + 2;
free(b);
*a = 11;  // Gefahr!
```



EXTRAS IN 3 MINUTEN

FRAGEN – ANTWORTEN – RÄTSEL
UND KURZE ZUSAMMENFASSUNG

Was sind die drei goldenen Regeln der dynamischen Speicherverwaltung?

1. Jeder malloc()-Aufruf braucht ein korrespondierendes free().
2. Nur mit malloc() allozierter Speicher darf mit free() freigegeben werden.
3. Speicher darf nur einmal freigegeben werden.

```c
char* format_log_entry(int timestamp, const char* message) {
    char* entry = malloc(1024);  // Platz für Zeitstempel + Nachricht
    snprintf(entry, 1024, "[%d] %s", timestamp, message);
    return entry;
}

void process_log_entry(char* entry) {
    // Verarbeite Log-Eintrag...
    if(strstr(entry, "ERROR") != NULL) {
        store_error_entry(entry);
    }
}

int main() {
    const char* messages[] = {
      "System started", "ERROR: Invalid input", /* ... */};

    for(int i = 0; i < 1000000; i++) {
        char* entry = format_log_entry(i, messages[i % 4]);
        process_log_entry(entry);
    }
}
```

```c
void format_log_entry(int timestamp, const char* message,
                      char* buffer, size_t bufsize) {
    snprintf(buffer, bufsize, "[%d] %s", timestamp, message);
}


int main() {
    const char* messages[] = {
      "System started", "ERROR: Invalid input", /* ... */};

    char* entry = malloc(1024);

    for(int i = 0; i < 1000000; i++) {
        format_log_entry(i, messages[i % 4], entry, 1024);
        process_log_entry(entry);
    }
    free(entry);
}
```

Parameter von malloc: size_t

```c
// Meistens kein Problem:
int count = 1000;
int* data = malloc(count * sizeof(int)); // Kleine Allokation

// Könnte problematisch werden:
int count = user_input(); // Was, wenn count negativ ist?
int* data = malloc(count * sizeof(int)); // Negativer Wert wird zu riesigem unsigned!

// Besser:
size_t count = validate_user_input(); // Stellt sicher: count ist positiv
int* data = malloc(count * sizeof(int));
```

malloc erwartet `size_t` als Parameter. `size_t` ist der korrekte Typ für Speichergrößen: garantiert groß genug für max. Objektgröße auf der Plattform, für die das Programm kompiliert wird.

`size_t` ist *unsigned*, d.h. keine negativen Größen möglich.

Achtung: −1 wird zu 4 GB wenn als unsigned interpretiert.

Dynamischer Speicher kommt vom Heap (malloc)

Pointer auf freige-gebenen Speicher nicht mehr verwenden

Stack-Speicher verschwindet am Funktionsende automatisch

Speicherallokation sollte transparent sein (Verantwortlichkeiten klar)

Heap-Speicher muss explizit freigegeben werden (free)

Vorsicht bei Speichergrößen: size_t verwenden