„Systemnahe Programmierung“ (BA-INF 034) Wintersemester 2023/2024

Dr. Matthias Frank, Dr. Matthias Wübbeling

Institut für Informatik 4

Universität Bonn

E-Mail: *{matthew, matthias.wuebbeling}@cs.uni-bonn.de*

Sprechstunde: nach der Vorlesung bzw. nach Vereinbarung

UNIVERSITÄT BONN

C Speicherverwaltung (Folie 35 C-Crash-Kurs WS20/21)

(dort noch in altem Design)

- **Dynamische Objekte in Java:** `new` und Garbage Collector
- **In C:** Manuelles Speichermanagement durch `malloc()` und `free()`

```
char *buffer, *dest;                                    // Zeiger auf Zielpuffer
buffer = malloc(BUFFER_SIZE);                           // Speicher reservieren
dest = buffer;                                          // Zeiger merken
...
free(buffer);                                           // Speicher freigeben
```

- **`malloc()`** allokiert Speicher auf dem „Heap“
  - Mengenangabe in Byte
  - Rückgabe: Ein Zeiger, oder NULL falls Allokation fehlgeschlagen ist

- **`free()`** gibt vorher allokierten Speicher wieder frei
  - Nicht vergessen!
  - In backwards.c: „`dest = buffer;`“ wird benötigt, damit Zeiger auf allokierten Speicher für `free()` nicht verloren geht, da `dest` in der Schleife verändert wird!

Systemnahe Programmierung

(BA-INF 034)

Kapitel 2: C-Crashkurs für Java-Programmierer

Wintersemester 2011/12

Prof. Dr. Björn Scheuermann

Der Inhalt der nachfolgenden Folien zur Speicherverwaltung
entstamm der Vorlesung von
Björn Scheuermann,
WS 2011/2012.

Speicherverwaltung

- Häufig werden **`malloc()`** und **`free()`** deshalb **in Konstruktionen ähnlich der folgenden eingesetzt:**

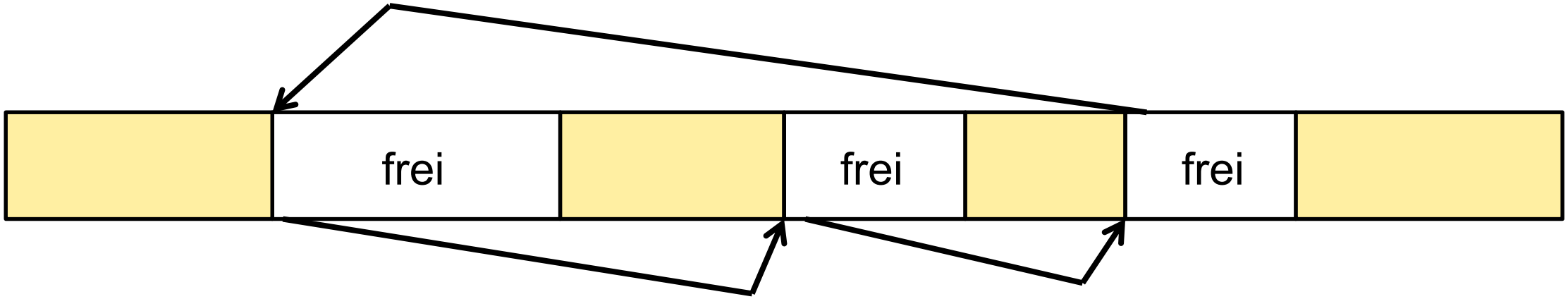
```
...
int *p;
p = malloc(1024 * sizeof(int));
if (p == NULL) {
    /* Fehlerbehandlung */
    ...
} else {
    /* irgendwas mit dem Speicher tun */
    ...
    free(p);
}
...
```

(Beispiel-) Umsetzung von `malloc/free`

- Wir schauen nun unter die Haube: **Wie arbeiten `malloc/free` intern?**
- Dafür gibt es keine festen Regeln, jedes System kann `malloc` auf beliebige Weise realisieren – jede funktionierende Implementation ist gleichermaßen „erlaubt“
- Wir schauen eine prototypische Beispielimplementation an (angelehnt an Kernighan & Ritchie: The C Programming Language, 2nd Edition, Abschnitt 8.7)
- **Zentrale Fragen:**
  - Wie findet `malloc` einen freien Speicherbereich passender Größe?
  - Woher weiß `free`, wie groß der freizugebende Speicherblock ist?

Die Free-Liste

- `malloc/free` verwalten (in unserer Implementation) freien Speicher in einer zyklisch verketteten Liste
- Jedem freien Speicherblock geht ein Header voraus
- **Der Header enthält:**
  - die Größe des freien Speicherblocks (ohne Header)
  - einen Zeiger auf den nächsten freien Speicherblock
- Der Zeiger im Header des letzten freien Blocks zeigt wieder auf den ersten freien Block



`malloc`

- Ein Aufruf von `malloc` durchläuft die **Liste freier Blöcke** und sucht nach einem, **der groß genug für die angeforderte Menge Speicher** ist

- Sobald einer gefunden ist **(„first fit“)**, wird darin die angeforderte Menge **Speicher + Platz für einen Header belegt**
  - wenn es **genau passt**, dann wird der **Block aus der Free-Liste genommen** (durch entsprechendes Anpassen des Zeigers des vorangegangenen Blocks)
  - **wenn der freie Block größer** als die angeforderte Speichermenge ist, dann wird die **benötigte Menge Speicher *hinten abgeschnitten***
    - warum hinten? Dann müssen die Zeiger in der Free-Liste nicht geändert werden, sondern nur das Größen-Feld im Header des betroffenen freien Blocks

`malloc`

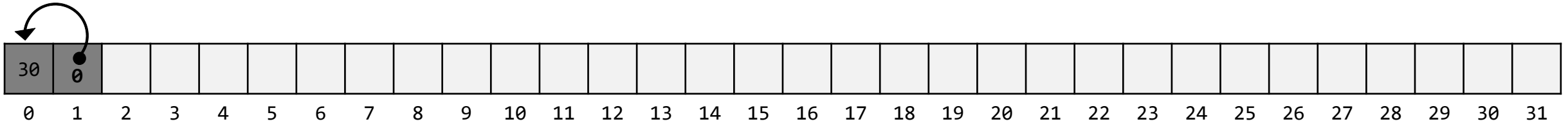
- Auch am Anfang jedes mit **`malloc`** reservierten Blocks steht ein **Header**
- Sieht genauso aus wie der Header freier Blöcke, aber das Zeigerfeld wird nicht verwendet
- Der **Rückgabewert** von `malloc` ist ein Zeiger auf das erste Byte *hinter dem Header*
  - so kann das aufrufende Programm ab der zurückgegebenen Adresse beliebig arbeiten, es „sieht“ den `malloc`-Header nicht
- Wenn `malloc` keinen freien Block ausreichender Größe findet, dann bittet es das Betriebssystem um **Zuteilung weiteren Speichers** (eine teure und aufwändige Operation!)
  - daraus wird dann ein neuer, großer freier Block erzeugt und in die Free-Liste eingefügt
  - danach kann `malloc` normal arbeiten und die Anforderung durch Reservierung eines Teils des neuen freien Blocks bedienen

`free`

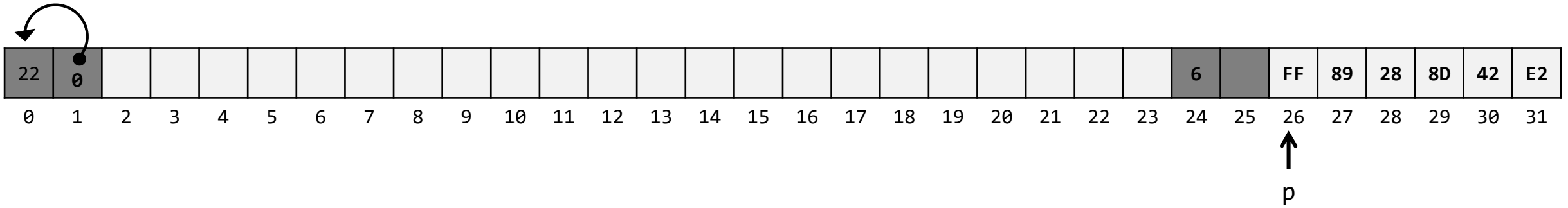
- `free` erhält als Parameter einen Zeiger, den ein früherer Aufruf von `malloc` zurückgegeben hat
- **„Vor“ der Adresse**, auf die dieser Zeiger zeigt, liegt also der `malloc`-Header mit der Größenangabe
- **`free` verwendet diese Information**, um einen freien Block zu generieren
  - zunächst sucht es in der Free-Liste nach den nächstgelegenen freien Blöcken davor und dahinter
  - wenn (min.) einer direkt angrenzt: Blöcke zusammenfassen
  - andernfalls an der entsprechenden Stelle in die Liste einfügen
- **Ohne den `malloc`-Header wäre nicht klar**, wie groß der freizugebende Speicherbereich überhaupt ist

`malloc/free` – Beispiel

- 32 Byte großer Speicher, 1 Byte lange Adressen (0-31)
- Zu Beginn ein einziger, großer, freier Block:

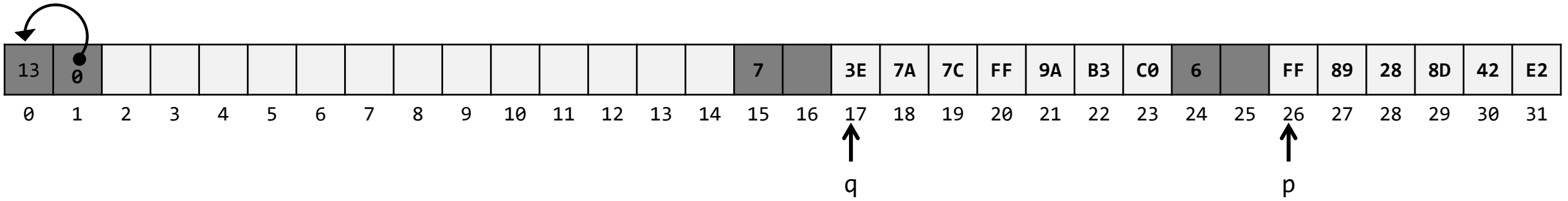


- Nach Aufruf von `malloc(6)` (Rückgabewert: p) und schreiben von `FF 89 28 8D 42 E2` an die Stelle, auf die p zeigt:

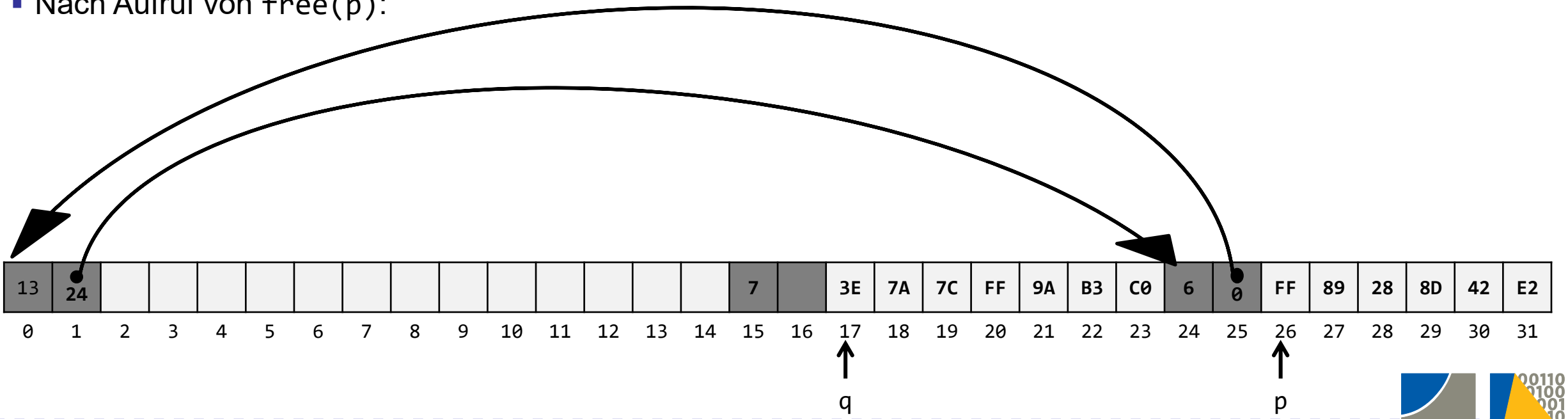


`malloc/free` – Beispiel (Forts.)

- Nach Aufruf von `malloc(7)` (Rückgabewert: q) und schreiben von `3E 7A 7C FF 9A B3 C0` an die Stelle, auf die q zeigt:



- Nach Aufruf von `free(p)`:



Typische Programmierfehler mit `malloc/free`

- **Keine Abfrage**, ob das Ergebnis von `malloc` ein Nullzeiger ist
  - `malloc` gibt `NULL` zurück, wenn die Anforderung fehlschlägt
  - da das nicht vorhersehbar ist, muss dies **überprüft** werden!
  - wie im Beispiel weiter oben:

```
...
int *p;
p = malloc(1024 * sizeof(int));
if (p == NULL) {
   /* Fehlerbehandlung */
   ...
```

Typische Programmierfehler

- Nicht freigegebener Speicher / Speicherlecks:
  - es passiert sehr leicht, dass `malloc` aufgerufen wird, aber das „zugehörige“ `free` fehlt
  - in diesem Fall bleibt der Speicher reserviert
  - solche Programme „fressen“ mehr und mehr Speicher, je länger sie laufen
  - das nennt man „Speicherleck“ (engl. memory leak)

```
int *p;
for (int i=0; i<1000; i++) {
    p = malloc(sizeof(int));
    if (p != NULL) { *p = i; }
}

/* Hier existiert nur noch ein Zeiger auf den zuletzt angeforderten Speicherblock! Alle vorigen
   können also gar nicht mehr freigegeben werden! */
```

Typische Programmierfehler

- Wenn ein **Zeiger** verwendet wird, der in **einen bereits freigegebenen Speicherbereich zeigt**, ist das **Ergebnis unvorhersehbar:**
  - falls der entsprechende Bereich schon **neu vergeben** wurde, kann man mitten in einer **„fremden“ Datenstruktur** landen
  - falls das Programm auf die **entsprechenden Adressen mittlerweile nicht mehr zugreifen darf** (z.B. weil der Speicherbereich ans Betriebssystem zurückgegeben wurde), ist das Ergebnis ein ***Segmentation Fault***

```
int *p;
p = malloc(sizeof(int));
...
free(p);
...
*p = 17;
```

Bei unserem Beispiel:
- Landen in „fremder“ Datenstruktur
- Oder sogar Überschreiben eines Headers von später angeforderten Speicherblöcken

Typische Programmierfehler

- Aufrufe von `free` mit einem Zeiger, der nicht das Ergebnis eines `malloc`-Aufrufs war, haben ebenfalls unvorhersehbare Folgen
  - `free` (zumindest in unserer Implementierung) wird den Speicherbereich vor der übergebenen Adresse als Header interpretieren
  - es wird den betreffenden Block in die Free-Liste aufnehmen, dabei „Zeiger“ aktualisieren
  - wenn vor dem übergebenen Zeiger kein sinnvoller Header steht, sondern irgendwelche anderen Daten, dann kann alles mögliche passieren

Bei unserem Beispiel:
- Längenfeld wird gelesen & interpretiert (Wert 0 … 255)
- Freier Block könnte mit anderen Speicherzellen überlappen
- Ggf. sogar Speichergrenze des Prozesses überschreiten

```
int *p;
int *q;
p = malloc(20 * sizeof(int));
q = &(p[10]);              /* hier steht kein malloc-Header! */
free(q);
```

Preisfrage

- Was gibt „backwards“ bei folgender Eingabe aus?

```
$ ./backwards Zweite Vorlesung Systemnahe Programmierung
```

Preisfrage

- Warum bricht „backwards“ mit einer Fehlermeldung ab?

32 Byte!

```
buffer = malloc(BUFFER_SIZE);                    // Speicher reservieren
dest = buffer;                                   // Zeiger merken
for (i=argc-1; i>0; i--) {                       // Rückwärts über alle Argumente
    for (j = strlen(argv[i])-1; j>=0; i--) {
        *dest++ = argv[i][j];                    // char in Puffer schreiben
```

- Eingabe „Zweite Vorlesung Systemnahe Programmierung“
  - 43 Byte lang und damit länger als der reservierte Puffer => Speicherüberlauf!
- In diesem Fall: Glück, dass nicht andere Variablen überschrieben wurden
  - Seiteneffekte. Solche Fehler sind sehr schwer zu finden!
- Solch ein „Buffer Overflow“ kann Angriff auf Rechner ermöglichen!
- Lektion: Im Umgang mit Zeigern und Speicher vorsichtig sein!

Finden von Speicherfehlern

- Es ist sehr schwierig, Fehler bei der Verwendung von `malloc/free` zu finden
- Es kann sein, dass sich ein fehlerhaftes Programm je nach System oder sogar bei jedem Aufruf anders verhält
- Es kann sogar sein, dass es völlig fehlerfrei arbeitet – aber eben nur manchmal!
- Am besten: Gut aufpassen und keine Fehler machen! ;-)
- Da das nicht immer klappt, helfen sogenannte Speicher-Debugger
- **Unter Linux gibt es `valgrind` (bisherige SysProg-Empfehlung)**
  - `valgrind` kann Speicherlecks, ungültige Zeiger und viele weitere Fehlertypen ziemlich zuverlässig erkennen
  - gibt wertvolle Hinweise zum Eingrenzen der Fehler
  - **Fazit: Verwenden!**
- Vgl. WS 2023/2024 Übungsblatt 1: **clang und ASAN (AddressSanitizer)**

Literatur

- Das war **nur eine Auswahl von Grundlagen zu C**.
  Vieles wurde nicht erwähnt:
  - **`typedef`, `enum`**, Pointer auf Funktionen, verschiedene Sprachstandards (C89, C99, …), Parameter in Makros, Debugger (gdb, …), Bibliotheken, …

- **Bei Fragen:**
  - „man 3“ und Google
  - „The C Book“ (online)
    - http://publications.gbdirect.co.uk/c_book/
  - „C for Java Programmers“ (online)
    - https://www.cs.utexas.edu/~ans/classes/cs439/docs/dictaat.pdf
  - Brian W. Kernighan und Dennis Ritchie
    - “The C Programming Language”
  - Mailingliste

(Links getestet 23.10.2023)