Programmieren in C

SS 2021

Vorlesung 6, Dienstag 1. Juni 2021

(nochmal Felder, Speicher, Debugger)

Prof. Dr. Peter Thiemann
Professur für Programmiersprachen
Institut für Informatik
Universität Freiburg
Folienvorlage von Prof. Dr. Hannah Bast

UNI FREIBURG

Die heutige Vorlesung

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem Ü5
  - Ankündigungen
- Inhalt
  - 2D dynamische Felder
  - Debugging, Sanitizer — gdb
  - Speicherorganisation — Operatoren [] und *
  - **Übungsblatt 6:**

Erfahrungen mit dem Ü5 1/2

UNI FREIBURG

- 277 Abgaben, 198 Erfahrungen, 194 mit lesbarer Zeit
- Zeitstatistik (in Stunden)

Min	Q1	Med	Q3	Max	Avg	MD	Var
0.5	5.5	7.5	9.0	29.0	7.79	2.61	14.79

- Regexp-Stats

#	Schlüsselworte
39	schwer|schwierig|anspruch|aufwendig|fordernd|hart|viel zeit|lange gedauert
1	nicht schwer|nicht schwierig|nicht anspruch|nicht aufwendig|unaufwendig|nicht fordernd|nicht hart|nicht viel zeit|nicht lange gedauert
94	cool|nett|spaß|gut|schön|toll|super
2	nicht cool|uncool|nicht nett|keinen spaß|nicht gut|nicht schön|unschön|nicht toll|nicht super
9	unklar|verwirrend

Erfahrungen mit dem Ü5 2/2

UNI FREIBURG

- Probleme mit Speicher

  schwer, zu lange, nicht so kreativ, gab schon bessere Übungsblätter -> kommt jetzt

- Weitere Probleme

  Die Tatsache, dass wir jetzt eine Idee haben, was hinten diesen Funktionen (wie z.B. Python oder C++) steht, finde ich echt cool.

  Das heisst aber nicht, dass es Spass gemacht hat. Auch mit den address sanitizer als Begleiter auf meine C Abenteuer, sind alle Errors extrem nervig und nehmen eine grosse Menge Zeit.

Ankündigungen 1/1

UNI FREIBURG

- Treffen mit den Tutoren zum Zweiten
  - Teil der Studienleistung: 10-15 minütiges Gespräch
  - Begann letzte Runde (Blatt 5)
  - Melden Sie sich beim Tutor wegen eines Termins dafür!

Felder 1/6

UNI FREIBURG

- Zwei-dimensionale dynamische Felder
  - Anforderung: Schreiben und Lesen von dia[i][j] mit beliebigem i, j >= 0
  - Angenommen aktuell gilt die Dimensionierung

    int dia[d_1][d_2];
  - Beim Speichern gemäß row-major Verfahren müssten d_1 und d_2 so geändert werden, dass d_1>i und d_2>j
  - Problem: wenn sich d_2 ändert, ändert sich die Schrittweite für d_1

Felder 2/6

UNI FREIBURG

- Zweidimensionale dynamische Felder (Beispiel)
  - Vorher: int dia[2][2] = {0, 1, 2, 3};
  - assert(dia[1][1] == 3);

0	1
2	3

  - Schreiben auf dia[1][3] = 99 erfordert Änderung nach int dia[2][4] (mindestens)
  - Naives Anpassen der Adressberechnung zerstört den alten Inhalt:
  - assert(! (dia[1][1] == 3));

0	1	2	3
0	0	0	99

Felder 3/6

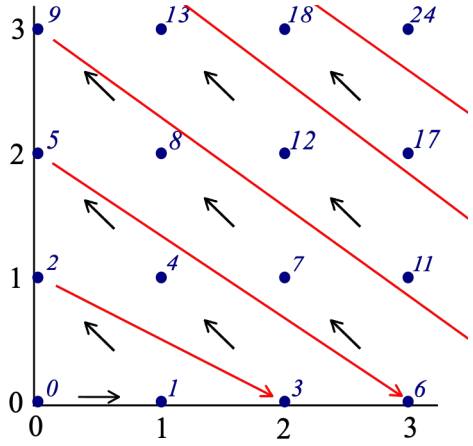
UNI FREIBURG

- Zweidimensionale dynamische Felder
  - Gesucht ist Abbildung von beliebigem i, j >= 0 auf lineare Adressen, sodass
    - Vergrößerung möglich ist und
    - Alle vorhandenen Inhalte unverändert bleiben und
    - Keine Umspeicherung notwendig ist

Felder 4/6

- Zweidimensionale dynamische Felder
  - Die Diagonalenmethode berechnet

    address(i, j) = (i + j) * (i + j + 1) / 2 + j
  - Auch bekannt als Cantors Paarfunktion
  - Die Funktion ist umkehrbar, d.h. aus dem Wert von address(i, j) können i und j eindeutig ermittelt werden

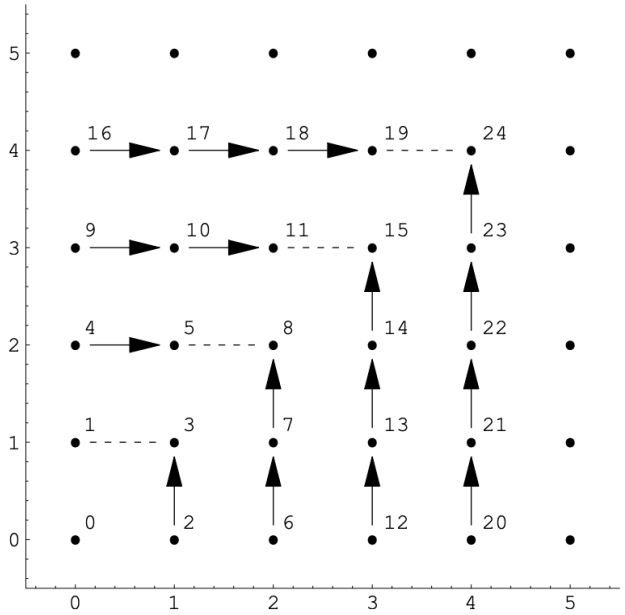


- Illustration: By I, Cronholm144, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2316840

Felder 5/6

- Zweidimensionale dynamische Felder
  - Die Quadratschalenmethode bildet (i, j) ab auf

    address(i, j) = i >= j ? i * i + i + j : j * j + i
  - Die Funktion ist ebenfalls umkehrbar



  - Quelle: http://szudzik.com/ElegantPairing.pdf

Felder 6/6

UNI FREIBURG

- Zweidimensionale dynamische Felder
  - Implementierung analog zu eindimensionalen dynamische Feldern
  - Zum Lesen/Schreiben auf i, j berechne

    ```
    size_t offset = address(i, j);
    ```

  - Falls der offset die aktuelle Größe überschreitet, muss das unterliegende eindimensionale Feld entsprechend vergrößert werden.
  - Danach kann die Lese-/Schreiboperation durchgeführt werden.

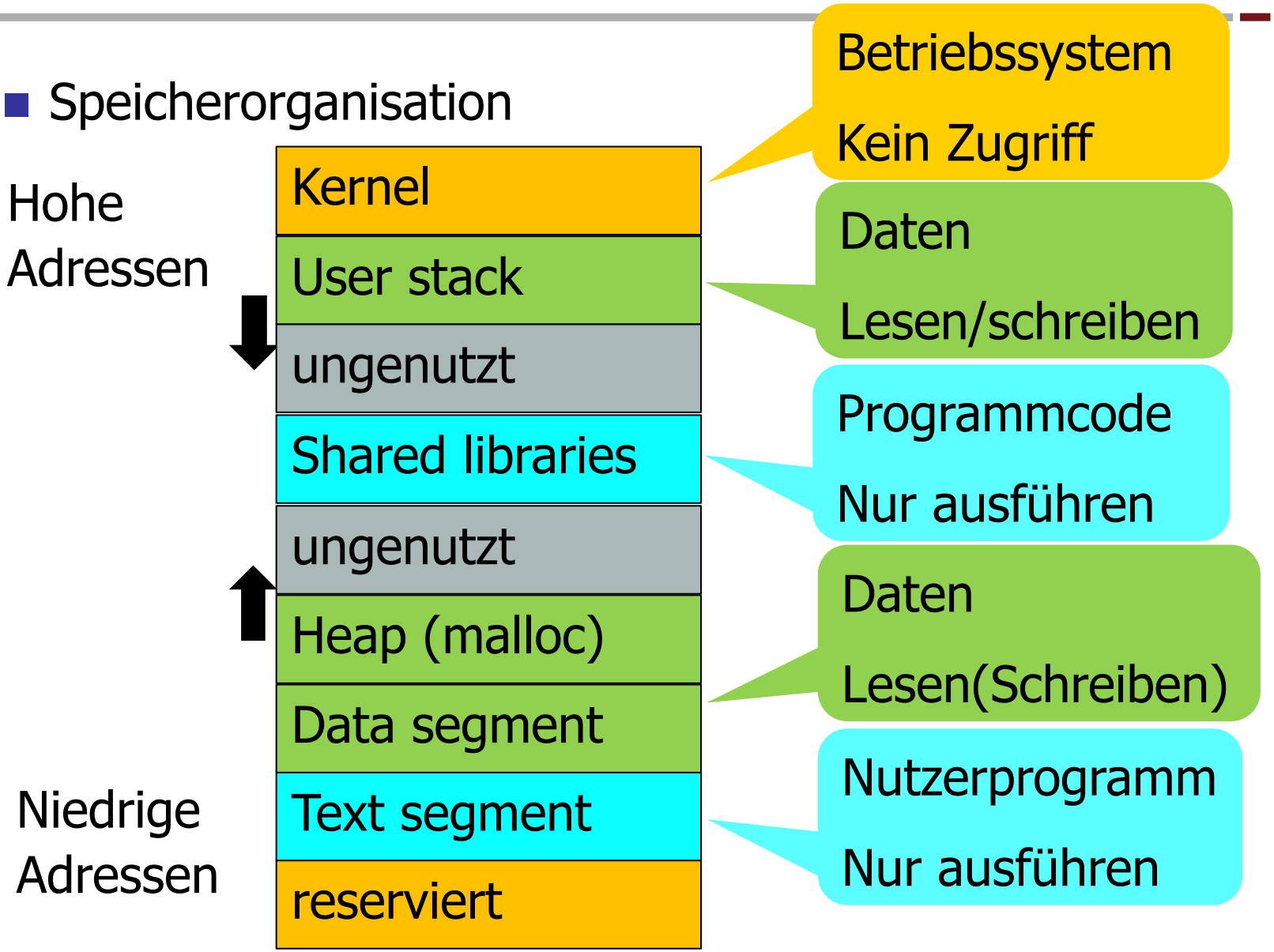
Speicher 1/11

UNI FREIBURG

- Physikalischer Hauptspeicher
  - Mehrere GB
  - Verwaltet vom Betriebssystem
  - Geteilt zwischen Betriebssysten und laufenden Prozessen
- Virtueller Speicher
  - Jeder Prozess hat die Illusion sich „allein“ im Adressraum des Prozessors zu befinden
  - Adressraum 2^{64} Bytes
  - Der Prozess darf nur auf kleine Bereiche davon zugreifen
  - Der Fehler segmentation violation zeigt einen unberechtigten Zugriff an

Speicher 2/11

- Speicherorganisation



Speicher 3/11

- Zugriffsrechte
  - Grüne Bereiche
    - Lesen und Schreiben ☑ 😀
    - Ausführen verboten ❌ 🤮
  - Blaue Bereiche
    - Lesen und Schreiben verboten ❌
    - Ausführen ok ☑
  - Beige Bereiche
    - Jeder Zugriff gibt segmentation violation 🤦
  - Graue Bereiche
    - Werden nach Bedarf dem Prozess zugeteilt

Speicher 4/11

UNI FREIBURG

- Das Datensegment

Data segment

  - Enthält (Platz für) alle Variablen, die global definiert sind
  - Beispiel

```
char *message = "Hello world!";
int counter = 0;
int myarray[42] = {0};
int main (void) {... }
```

  - Wieviel Platz brauchen wir hierfür im Datensegment?

Speicher 5/11

UNI FREIBURG

- Heap (Haufen)

Heap (malloc)

  - Wird durch stdlib verwaltet
  - Malloc() & Freunde vergrößern den Heap
  - Fordern notfalls neue Speicherbereiche vom Betriebssystem an
  - Leer bei Programmstart
  - Bleibt leer, falls ein Programm kein malloc() verwendet

Speicher 6/11

UNI FREIBURG

- Stack (Stapel)

User stack

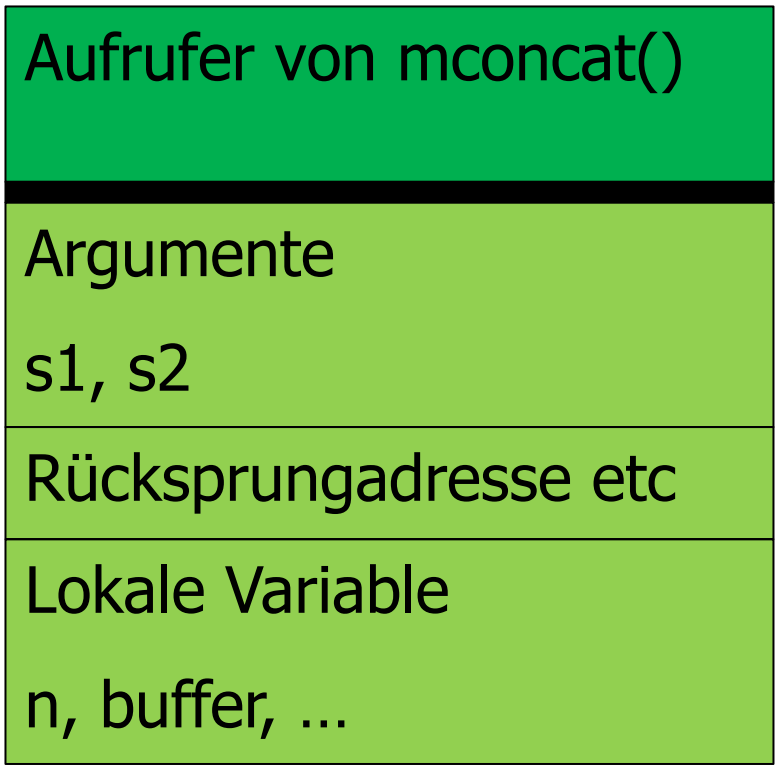
  - Speicherbereich zur Verwaltung von Funktionsaufrufen und Funktions-lokalen Daten
  - Jeder aktive Funktionsaufruf belegt ein **Stackframe** im Stacksegment
  - Stack wächst mit jedem Aufruf; schrumpft bei return
  - Beispiel aus V04:

```
char * mconcat (const char *s1, const char *s2) {
    size_t n = mstrlen (s1) + mstrlen (s2) + 1;
    char * buffer = …;
```

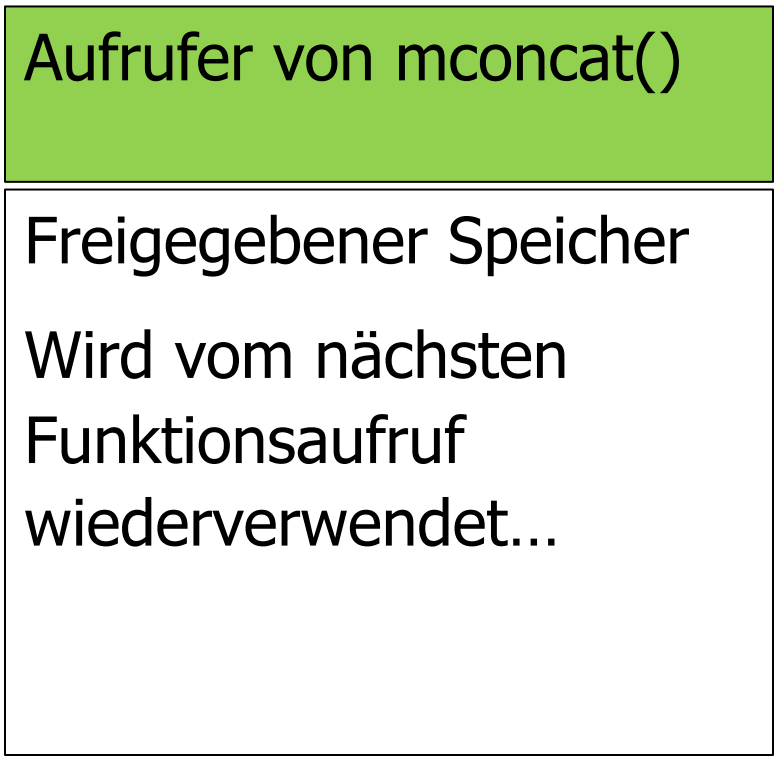
  - Die Variablen s1, s2, n und buffer befinden sich im Stackframe eines Aufrufs von mconcat()

Speicher 7/11

- Stackframe
  - Direkt nach Aufruf von.
  - mconcat("hello ", "world");



Nach return:



Speicher 8/11

UNI FREIBURG

- Stackframes
  - Stackframes werden wiederverwendet
  - Der nächste Funktionsaufruf verwendet den gleichen Speicherbereich
  - Lokale Variablen des früheren Aufrufs können eingesehen werden
  - **Niemals** Adressen von lokalen Variablen zurückgeben
- Stacktrace
  - Liste der aktuell aktiven Stackframes
  - Alle offenen Funktionsaufrufe mit Parametern etc
  - Kann durch den Debugger angezeigt werden

Speicher 9/11

UNI FREIBURG

- Nullpointer

  - Der unterste Speicherbereich ab Adresse 0 ist reserviert
  - Zeigervariablen dürfen den Wert Null annehmen (Nullpointer), aber ein Zugriff darüber ist nicht erlaubt und liefert segmentation violation
  - Daher muss vor jedem Zugriff über einen Zeiger sichergestellt werden, dass der Zeiger nicht Null ist.
  - Das wird oft vergessen und führt zu Fehlern

Speicher 11/11

UNI FREIBURG

- Speicherschutz mit const Typen
  - Das Schlüsselwort const kann in Typen verwendet werden um anzuzeigen, dass der Wert eines Speicherbereichs nicht geändert werden darf
  - Beispiele

```
size_t const buffersize = 2048; // don‘t assign to buffersize
char const *splash = “Hello, World!“;
                                  // must not assign into the string
char * const splosh = “Hi!“;    // must not assign to splosh
char const * const rigid = “Ultimate protection“;
                       // must not assign to rigid nor into the string
```

Tipp 1/2

UNI FREIBURG

- Lese Typen mit der Spiralregel
  - Beginne beim Namen der Variablen
  - Gehe weiter in einer Spirale im Uhrzeigersinn
    - [X] – Feld (der Größe X) von …
    - * -- Zeiger auf …
    - Const – Konstante
  - (Achtung: Heuristik!)

Tipp 2/2

UNI FREIBURG

- Quiz
  - int const buffer[20] = {0,1,2,3};
  - char (*ap)[20];
  - int * const * p;

Debugging 1/6

UNI FREIBURG

- Fehler im Programm kommen vor
  - Mit Feldern, Zeigern und malloc() lassen sich unangenehme **seg**mentation **faults** produzieren
  - Das passiert beim versuchten Zugriff auf Speicher, der dem Programm nicht gehört, zum Beispiel

```
int* p = NULL;    // Pointer to address 0.
*p = 42;          // Will produce a segmentation fault.
```

  - Schwer zu debuggen, es kommt dann einfach etwas wie:

```
Segmentation fault (core dumped)
```

    Ohne Hinweis auf die Fehlerstelle im Code (gemein)
  - Manche Fehler sind nicht deterministisch, weil sie von nicht-initialisiertem Speicherinhalt abhängen

Debugging 2/6

UNI FREIBURG

- Methode 1: printf
  - printf statements einbauen
    - an Stellen, wo der Fehler vermutlich auftritt
    - von Variablen, die falsch gesetzt sein könnten
  - **Vorteil:** geht ohne zusätzliches Hintergrundwissen
  - **Nachteil 1:** nach jeder Änderung neu kompilieren, das kann bei größeren Programmen lange dauern
  - **Nachteil 2:** printf schreibt nur in einen Puffer, dessen Inhalt bei segmentation fault nicht ausgedruckt wird, wenn die Ausgabe in Datei umgeleitet wird. Abhilfe: nach jedem printf

```
fflush(stdout);
```

Debugging 3/6

UNI FREIBURG

- Methode 1a: printf
  - **Nachteil 3:** nach Ende der Debug-Session müssen die printf entfernt werden.
  - Abhilfe: Verwende Makro eprintf …

```
#ifdef DEBUG
#define eprintf(...) fprintf(stderr, __VA_ARGS__)
#else
#define eprintf(...)
#endif
```

  - Einschalten durch Kompilieren mit -DDEBUG

Debugging 4/6

UNI FREIBURG

- Methode 2: gdb, der **GNU debugger**
  - Gbd Features
    - Anweisung für Anweisung durch das Programm gehen
    - Sogenannte breakpoints im Programm setzen und zum nächsten breakpoint springen
    - Werte von Variablen ausgeben (und ändern)
  - **Vorteil:** beschleunigte Fehlersuche im Vergleich zu printf
  - **Nachteil:** ein paar gdb Kommandos merken

Debugging 5/6

UNI FREIBURG

- Grundlegende gdb Kommandos
  - **Wichtig:** Programm kompilieren mit der –g Option!
  - gdb aufrufen, z.B. gdb ./ArraysAndPointersMain
  - Programm starten mit run <command line arguments>
  - stack trace (nach seg fault) mit backtrace oder bt
  - breakpoint setzen, z.B. break Number.c:47
  - breakpoints löschen mit delete oder d
  - Weiterlaufen lassen mit continue oder c
  - Wert einer Variablen ausgeben, z.B. print x oder p i

Debugging 6/6

UNI FREIBURG

- Weitere gdb Kommandos
  - Nächste Zeile im Code ausführen step bzw. next

    step folgt Funktionsaufrufen, next führt sie ganz aus
  - Aktuelle Funktion bis zum return ausführen finish
  - Aus dem gdb heraus make ausführen make
  - Kommandoübersicht / Hilfe help oder help all
  - gdb verlassen mit quit oder q
  - Wie in der bash command history mit Pfeil hoch / runter

    Es geht auch Strg+L zum Löschen des Bildschirmes

Debugging 7/6

UNI FREIBURG

- Methode 3: AddressSanitizers
  - Mit Zeigern können leicht lesende oder schreibende Zugriffe über Feldgrenzen hinaus passieren
  - Oder auch Zugriffe auf unerlaubte Bereiche (wie NULL)
  - Solche Fehler findet man gut mit **-fsanitize=address**
  - **Doku dazu**
  - **https://github.com/google/sanitizers/wiki/AddressSanitizer**

Generische Felder 1/3

UNI FREIBURG

- Dynamische Felder mit beliebigem Typ
  - Muss auch die Größe des Basistyps vorhalten

```
typedef struct _dynarray {
    size_t da_size;          // Current number of elements.
    ???? da_mem;             // Actual array.
    size_t da_elem_size;     // Bytes per element.
    ???? da_default;         // Default value.
 } dynarray;
```

  - Problem: Typ und Größe der Werte unbekannt
  - Lösung: Zeiger auf beliebigen Typ

```
void * da_mem;
void * da_default;
```

Generische Felder 2/3

UNI FREIBURG

- Dynamische Felder mit beliebigem Typ
  - API Versuch #1: erzeugen, lesen, schreiben

```
dynarray * da_new(
    size_t initial_size,
    size_t element_size,
    void * default_value);
void * da_read(dynarray * da, size_t i);
int da_write(dynarray * da, size_t i, void * val);
```

  - Probleme
    - Das Ergebnis von da_read zeigt in da_mem
    - Kann sich bei späteren da_write ändern
    - Wie kopieren?

Generische Felder 3/3

UNI FREIBURG

- Dynamische Felder mit beliebigem Typ
  - API Versuch #2: ..., lesen, schreiben

    ```
    int da_read(dynarray * da, size_t i, void * return_val);
    int da_write(dynarray * da, size_t i, void * val);
    ```

  - Lösung
    - da_read nimmt Zeiger zum Abspeichern des Ergebnisses
    - Kopiert selbst von da_mem dorthin
    - Rückgabewert int zeigt an ob angefragter Index innerhalb des Feldes
    - Bei da_write zeigt der Rückgabewert an, ob ggf. die Vergrößerung des Feldes erfolgreich war.

Generische Felder 3/3

UNI FREIBURG

- Verwendung von API #2
  - Typ der Elemente soll nun double sein

```
double v0 = 0.0;  // default value
double v1 = 1.0;
dynarray* d = da_new(20, sizeof(double), &v0);
da_write(d, 10, &v1);
double r;
da_read(d, 5, &r);
assert (r == v0);
da_read(d, 10, &r);
assert (r == v1);
```

Verwendete C-API <stdlib.h> 1/1

UNI FREIBURG

- void *realloc(void *p, size_t size)
  - Der Zeiger p muss von malloc(), realloc() oder calloc() angelegt worden sein.
  - Der size Parameter gibt die neue Größe (in Bytes) an.
- void *memcpy(void *t, const void *s, size_t n)
  - Kopiert n Bytes
  - Vom Speicherbereich beginnend ab s (nur lesend, daher const)
  - In den Speicherbereich beginnend ab t

Literatur / Links

- Felder / Arrays
  - https://en.wikipedia.org/wiki/Row-_and_column-major_order
  - https://en.wikipedia.org/wiki/Array_data_type
  - https://en.wikipedia.org/wiki/Dynamic_array
  - https://en.wikipedia.org/wiki/Dope_vector
- Debugger / gdb
  - http://sourceware.org/gdb/current/onlinedocs/gdb