Programmieren in C

SS 2019

Vorlesung 6, Dienstag 28. Mai 2019
(nochmal Felder, Debugger)

Prof. Dr. Peter Thiemann
Professur für Programmiersprachen
Institut für Informatik
Universität Freiburg

UNI
FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem Ü4 — Babylonisch
  - Ankündigungen
- Inhalt
  - Felder und Zeiger — Operatoren [] und *
  - Debugging — gdb
  - Hinweise zum Ü4 — Maus, Zellen, zwei Felder
  - Noch mehr zu make — ganz generisch, .PRECIOUS
  - **Übungsblatt 5:**

Erfahrungen mit dem Ü4 1/1

- Zusammenfassung / Auszüge
  - Schwierigkeiten mit Anzeige und Eingabe der Schriftzeichen
  - Wie funktioniert das Babylonische Zahlensystem?
  - Probleme mit wchar_t (Forum)
  - Nicht viel feedback bis Montagabend

Ankündigungen 1/2

- Eigene Tests
  - Viele Abgaben haben keine eigenen Tests neben den Vorgaben
  - Ab dem heutigen Blatt 5 sind eigene Tests obligatorisch
  - Fehlen diese, müssen wir Punkte abziehen
- Treffen mit den Tutoren
  - Teil der Studienleistung: 10-15 minütiges Gespräch
  - Beginnt diese Woche mit dem aktuellen Blatt
  - Melden Sie sich beim Tutor wegen eines Termins dafür!

Ankündigungen 2/2

- Gruppe 1 heute nachmittag
  - Fällt aus wegen Krankheit
- Gruppe 7 findet Donnerstag um 10 statt
  - Gruppe 8 und 9 entfallen
  - Feiertag / Vatertag

Felder 1/11

UNI FREIBURG

- Eindimensionale Felder
  - Sequentiell im Speicher abgelegt

```
int a[6] = {10, 20, 30, 40, 50, 60};
```

10	20	30	40	50	60

- Mehrdimensionale Felder
  - Werden auf eindimensionale Felder abgebildet

```
int b[2][3] = { {10, 20, 30}, {40, 50, 60} };
```

  - Zwei Zeilen (rows), drei Spalten (columns)
  - Im Speicher genau wie a

10	20	30	40	50	60

Felder 2/11

UNI FREIBURG

- Wie wird die Adresse für den Zugriff auf zweidimensionale Felder berechnet?
  - Lineare Funktion der I...
  - Beispiel für 0 <= i <

    Elemente pro Zeile

```
int* bp = b;       // Pointer to address b[0][0].
assert (b[i][j] == *(bp + 3*i + j)); // int b[2][3]
```

  - Nennt sich row-major order, weil erst alle Spalten (row) durchlaufen werden, bevor die nächste Zeile beginnt.
  - Als Matrix (jeweils Offset : Inhalt)

0 : 10	1 : 20	2 : 30
3 : 40	4 : 50	5 : 60

Felder 3/11

UNI FREIBURG

- Zugriff auf mehrdimensionale Felder
  - Auch eine lineare Funktion
  - Allgemeine Definition eines Feldes

```
int multi[d1][d2]...[dn];     // n dimensional array.
int *p = multi;     // Pointer to first element.
```

$$\texttt{assert(multi}[i_1][i_2]\ldots[i_n] ==$$

$$*(p + i_n + d_n * (i_{n-1} + d_{n-1} * (i_{n-2} + \ldots + d_2 * i_1)))$$

  - Strides für Indexe
  - Vorausberechnen
  - Dope-Vektor

Index	Schrittweite
i_n	1
i_{n-1}	d_n
i_{n-2}	$d_n * d_{n-1}$
:	:
i_1	$d_n * d_{n-1} * \ldots * d_2$

Felder 4/11

- Dynamische Felder
  - Was tun, wenn die Anzahl der Elemente eines Feldes vor Programmstart nicht bekannt ist bzw sich im Lauf des Programms verändern (meist vergrößern) kann?
  - In dem Fall muss das Feld dynamisch angelegt werden mit malloc(), aber der Zugriff darf nicht direkt per Index erfolgen. Beispiel

```
int* p = malloc(6 * sizeof(int));   // Pointer to int[6].
*(p + 10) = 42;                     // Illegal.
```

  - Stattdessen:
    - Datenstruktur, die sich die aktuelle Größe merkt
    - Indexfunktion, die Zugriffe überprüft und das Feld ggf. vergrößert

UNI FREIBURG

Felder 5/11

- Datenstruktur für dynamische int Felder
  - typedef struct _intarray {

```
    size_t ia_size;        // Current number of elements.
    int *ia_mem;           // Actual array.
    int ia_def;            // Default value.
} intarray;
```

  - API: erzeugen, löschen, lesen, schreiben

```
intarray * ia_new(size_t initial_size, int default_value);
void ia_destroy(intarray * ia);
int ia_read(intarray * ia, size_t i);
int ia_write(intarray * ia, size_t i, int val);
```

  - Schreiben vergrößert das Feld, wenn nötig

UNI FREIBURG

Felder 6/11

- 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

UNI FREIBURG

Felder 7/11

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 8/11

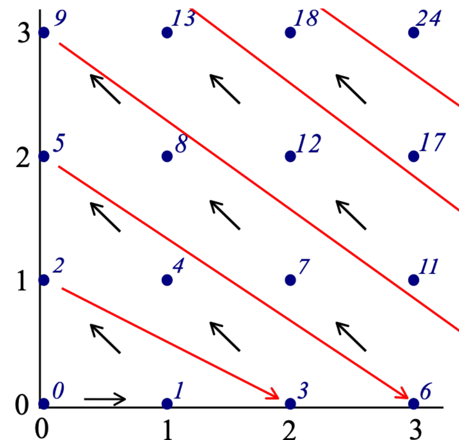
UNI FREIBURG

- 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 9/11

- Zweidimensionale dynamische Felder
  - Die Diagonalenmethode bildet

    (i, j) ab auf $address(i, j) = (i + j) * (i + j + 1) - 2 + j$
  - 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



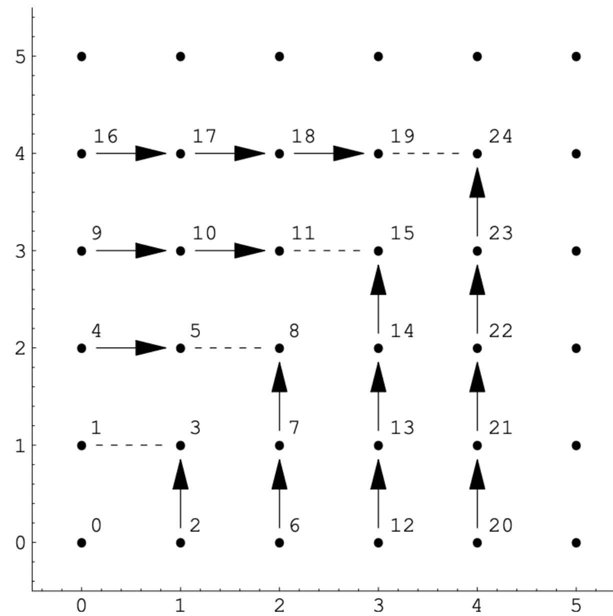
  - Illustration: By I, Cronholm144, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2316840

Felder 10/11

UNI FREIBURG

- 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 11/11

- 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.

Debugging 1/6

- 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 zudem nicht deterministisch, weil sie von nicht-initialisiertem Speicherinhalt abhängen

Debugging 2/6

- 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 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 Vgl zu printf
  - **Nachteil:** ein paar gdb Kommandos merken

Debugging 4/6

- 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.cpp: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 5/6

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 6/6

UNI FREIBURG

- Methode 3: valgrind
  - Mit Zeigern kann es schnell passieren, dass man über ein Feld hinaus liest / schreibt ... oder sonst wie unerlaubt auf Speicher zugreift
  - Solche Fehler findet man gut mit **valgrind**

    Machen wir später

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;
```

Genererische 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

- 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.

UNI FREIBURG

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

UNI FREIBURG

- 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