Programmieren in C

SS 2021

Vorlesung 7, Dienstag 8. Juni 2021

(Speicher, Debugger)

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

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem Ü6
  - Ankündigungen
- Inhalt
  - Generische Felder — memcpy()
  - Variablen und Regelmuster — Makefile
  - Debugging — gdb
  - Interaktives TUI
  - **Übungsblatt 7: TUI-Anwendung (Spiel)**

Erfahrungen mit dem Ü6 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 Ü6 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.

Generische Vektoren 1/3

UNI FREIBURG

- Dynamische Vektoren mit beliebigem Typ
  - Vgl int vector:

```
typedef struct _intvector {
    size_t ia_size;          // Current number of elements.
    int * ia_mem;            // Actual array.
} intvector;
```

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

```
void ** ga_mem;
```

  - Array von void-Zeigern
  - Änderungen in API erforderlich

Generische Vektoren 2/3

UNI FREIBURG

- Dynamische Vektoren mit beliebigem Typ
  - API für int

    ```
    bool int_vec_push (Vec *xs, int);
    int int_vec_at(Vec *xs, size_t i);
    ```

  - API für beliebigen Typ

    ```
    bool vec_push (Vec *xs, void * x);
    const void * const* vec_at(Vec *xs, size_t i);
    ```

    - vec_push übergibt mit malloc() alloziertes Datenelement an den Vektor
    - vec_at liefert einen Zeiger in die Mitte des Vektors
    - Das Ergebnis sollte nicht modifiziert werden, daher const

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;
genarray* d = ga_new(20, sizeof(double), &v0);
ga_write(d, 10, &v1);
double r;
ga_read(d, 5, &r);
assert (r == v0);
ga_read(d, 10, &r);
assert (r == v1);
```

Makefile 1/3

UNI FREIBURG

- Variable
  - Im Makefile lassen sich auch Variable definieren

    ```
    CFLAGS= -g –fsanitize=address

    EXECUTABLES= tui_example worm
    ```

  - Wert der Variable = Text bis Zeilenende
  - Verwendung überall im Makefile

    ```
    gcc $(CFLAGS) –c worm.c
    rm *.o a.out $(EXECUTABLES)
    ```

  - Aber auch in den Abhängigkeiten

    ```
    compile: $(EXECUTABLES)
    ```

  - Oder auch im Goal

    ```
    $(GOAL): …
    ```

Makefile 2/3

UNI FREIBURG

- Variable
  - Variablen lassen sich beim Aufruf von make setzen

    make CFLAGS=-DDEBUG
  - Das Flag –D des C-Compilers setzt Makros im Präprozessor überall im Makefile
  - Sinnvolle Verwendung der Variablen in den Regeln

    worm.o: worm.c

    $(CC) $(CFLAGS) –c worm.c

Makefile 3/3

- Variable
  - Oder noch kürzer mit automatischen Variablen

```
worm.o: worm.c
        $(CC) $(CFLAGS) –c $<
```

  - Die Variable $< wird automatisch auf die erste Abhängigkeit gesetzt
  - Verwende für jedes C-Modul die gleiche Regel!
  - Diese Regel kann als Regelmuster (pattern rule) hingeschrieben werden: wie mache ich eine .o Datei aus einer .c Datei?

```
%.o: %.c
        $(CC) $(CFLAGS) –c $<
```

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 zudem 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 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

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 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

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

- Speicher
  - https://manybutfinite.com/post/anatomy-of-a-program-in-memory/
  - https://en.wikipedia.org/wiki/Call_stack
  - http://c-faq.com/decl/spiral.anderson.html
- Debugger / gdb
  - http://sourceware.org/gdb/current/onlinedocs/gdb