Programmieren in C

SS 2021

Vorlesung 5, Dienstag 18. Mai 2021

(nochmal Felder, Speicher, Debugger)

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

Die heutige Vorlesung

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem Ü4
  - Ankündigungen
- Inhalt
  - Weitere C Typen — typedef und void*
  - Felder, Speicher und Zeiger — Operatoren [] und *
  - Debugging — gdb
  - **Übungsblatt 5:**

Erfahrungen mit dem Ü4 1/2

UNI FREIBURG

- 276 Abgaben, 224 Erfahrungen, 217 mit lesbarer Zeit
- Zeitstatistik (in Stunden)

Min	Q1	Med	Q3	Max	Avg	MD	Var
0.0	4.5	6.0	8.0	24.0	6.44	2.39	10.82

- Regexp-Stats

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

Erfahrungen mit dem Ü4 2/2

UNI FREIBURG

- Probleme mit Tests

  Also das war ja mal ein grausiger kampf mit dem unity tests -

  Ansonsten ziemlich cool. Aber eben sass laaaaange an den tests.

  Und die letzte Aufgabe wäre ohne input nicht lustig gewesen.

  Interessante Erfahrung...OoooOoooOoooO :)

- Probleme mit Strings

  Aber das nervigste: "encrypt" ist nicht gleich "encrypt". Type() gibts nicht. char*, char[], char, *char ... was soll das?

  Hat dann doch funktioniert mit strcmp()....

  Programmieren mit C - des is nit schee.

Ankündigungen 1/1

UNI FREIBURG

- 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!

Preludium - typedef 1/4

UNI FREIBURG

- Bei struct-Typen muss entweder das Tag geschrieben werden oder die Komponenten deklariert werden

```
struct Person { char* name; unsigned int age; };
```

- Nach dieser Deklaration können wir den struct-Typ verwenden

```
struct Person jimmy_blue; // Person is a struct tag
```

- Alternativ können wir abkürzend einen neuen Typ deklarieren:

```
typedef struct _Person { … } Person; // _Person is the tag
```

- Definiert einen neuen Typ Person als Alias von struct _Person

```
Person mac_malone = { "Mac", 42 };

Person* mac_p = &mac_malone;
```

- Definiendum immer am Schluss; vgl. Variablendeklaration

Preludium - typedef 2/4

UNI FREIBURG

- Nicht auf struct-Typen beschränkt

```
typedef char* string;

typedef unsigned int uint;
```

- Nun können wir schreiben

```
typedef struct _Person { string name; uint age } Person;
```

- Für Array-Typen schreiben wir:

```
typedef int ia5_t[5]; // ia_t is defined as a type

ia5_t my_array = { 5, 4, 3, 2, 1};
```

Preludium – void* 3/4

UNI FREIBURG

- Der Basistyp eines Zeigertypen beeinflusst die Zeigerarithmetik

```
int* ip; ip++;   // increment ip by sizeof(int)

double* dp; dp++; // increment dp by sizeof(double)
```

- Beide können durch malloc() initialisiert werden

```
ip = malloc(sizeof(int));

dp = malloc(sizeof(double));
```

- Der return-Typ von malloc ist void *
- Der Typ void * ist kompatibel mit allen Zeigertypen

```
void *vp = malloc(sizeof(int)); // ia_t is defined as a type

ip = vp;
```

Preludium – malloc & friends 4/4

UNI FREIBURG

- Das (fast-) komplette Interface von malloc
  - Speicher reservieren (bei Fehlschlag NULL)

    ```
    void *malloc(size_t size);
    ```

  - Speicher freigeben

    ```
    void free(void *ptr);
    ```

  - Größe des Speicherbereichs verändern

    ```
    void *realloc(void *ptr, size_t new_size);
    ```

  - Speicher für Array reservieren

    ```
    void *calloc(size_t nr_members, size_t size_member);
    ```

Felder 1/12

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 2/12

UNI FREIBURG

- Wie wird die Adresse für den Zugriff auf zweidimensionale Felder berechnet?
  - Lineare Funktion der Indexe
  - Beispiel für 0 <= i <2 und 0 <= j < 3:

```
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 3/12

UNI FREIBURG

- Zugriff auf mehrdimensionale Felder
  - Adresse ist lineare Funktion in den Indexen
  - Allgemeine Definition eines Feldes

    int multi[d_1][d_2]…[d_n]; // n-dimensional array.
    int *p = multi; // Pointer to first element.

    assert(multi[i_1][i_2]…[i_n] ==

    *(p + i_n + d_n * (i_{n-1} + d_{n-1} * (i_{n-2} + … + 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} * … * d_2

Felder 4/12

UNI FREIBURG

- 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

Felder 5/12

UNI FREIBURG

- API für dynamische int Felder (intarray.h)
  - Ein intarray ist ein struct, aber die Komponenten zeigen wir nur in der Implementierung intarray.c

    ```
    typedef struct _intarray intarray;
    ```

  - Funktionen: 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

Felder 6/12

UNI FREIBURG

- Datenstruktur für dynamische int Felder (intarray.c)

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

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