Programmieren in C

SS 2021

Vorlesung 4, Dienstag 11. Mai 2021

(Strings, Zeiger, Allokation, Strukturen)

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
  - Donnerstag ist Feiertag! → Notfahrplan für die Tutorate
  - Erfahrungen mit dem Ü3
- Inhalt
  - Präprozessor — 101
  - Zeiger — Operatoren * und &
  - Dynamischer Speicher — malloc und free
  - Strukturen — struct
  - **Übungsblatt 4: Fingerübungen und kleine A damit**

Erfahrungen mit dem Ü3 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 Ü3 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.

Präprozessor 101

UNI FREIBURG

- Der Präprozessor wird noch vor dem eigentlichen Compiler ausgeführt. Dies sind die wichtigsten Anweisungen:
  - #include - kennen wir schon
  - #define <name> <optionaler text>
  - #ifdef <name> - Konditional
  - #ifndef <name> - Konditional, kennen wir schon
  - #if <expr> - Konditional
  - #else
  - #endif
  - #error

Präprozessor 102

UNI FREIBURG

- #define definiert ein Makro

```
#define BUFSIZE 1024
/* every future occurrence of BUFSIZE is replaced by 1024 */
int buffer[BUFSIZE];
for(size_t i = 0; i < BUFSIZE; i++) {
   … buffer[i] …
}
```

- Präprozessormakros werden meist für Konstanten verwendet
- Gewöhnlich in .h Dateien

Präprozessor 103

UNI FREIBURG

- Verwendung von #if und #error

```
#if __STDC_VERSION__ < 201112
#error “This program requires a C11 compliant compiler.“
#endif
```

Zeiger 1/9

UNI FREIBURG

- **Zeiger** sind Variablen, deren Wert eine **Adresse** ist
- Die Deklaration gibt an, wie der Inhalt des Speichers an dieser Adresse zu interpretieren ist

```
int* p;  // Pointer to an int
```

- Zugriff auf diesen Wert mit dem * **vor** der Variablen

```
printf("%d\n", *p);  // Print the (int) value pointed to.
```

Zeiger - Zeigerarithmetik 2/9

UNI FREIBURG

- Ein Zeiger kann mit einer ganzen Zahl addiert werden ... sie wird automatisch mit der Größe des Typs multipliziert

```
printf("%d\n", *(p + 3));  // Int at p + 3 * sizeof(int).

p = p + 3;  // Increase the address by 3 * sizeof(int).

p++;  // Increase by sizeof(int).
```

- Die Zahl kann sowohl Konstante oder Variable sein.
- Auch negativ...

```
p = p - 10;  // Decrease the address by 10 * sizeof(int).

p--;  // Decrease by sizeof(int).
```

Zeiger – Zeigervergleiche 3/9

UNI FREIBURG

- Zwei Zeiger ins gleiche Feld dürfen subtrahiert werden. Das Ergebnis ist die Anzahl der Elemente dazwischen

```
int *q = p + 10;  // pointer to 11th element of p.
printf ("%d", q - p);  // Prints 10.
```

- Sie dürfen auch verglichen werden.

```
assert (!(p == q));  // p and q are different.
assert (p < q);  // By initialization of q.
```

- Und gedruckt…

```
printf ("%p < %p\n", p, q);  // p and q are different.
printf ("%d == %d\n", *p, *(q– 10));  // what they point to.
```

Zeiger 4/9

UNI FREIBURG

- Zeigerarithmetik ist generell nur **innerhalb des gleichen Felds** erlaubt!!!

```
int f[10] = {0};

int *p = f;  // pointer to 0th element of f. OK.

int *q1 = p + 5;  // OK. Inside array

int *q2 = p – 1;  // NOT OK. Outside array

int *q3 = p + 20; // NOT OK. Outside array

int *q4 = p + 10; // OK. Just past array. Don‘t dereference!
```

- Zeiger direkt hinter das Feld ist nützlich zum Testen ob das Feld komplett verarbeitet worden ist.

Zeiger und 1D Felder 5/9

UNI FREIBURG

- Der Name eines ein-dimensionalen Feldes a ist ein Zeiger auf das erste Element a[0] des Feldes

```
int a[5] = {3, 9, 2, 14, 5};
int* p = a;
printf("%d\n", *p);  // Will print 3.
```

- Der Zugriff über [...] ist definiert durch die auf der vorherigen Folie beschriebene Zeigerarithmetik:

```
int a[5] = {3, 9, 2, 14, 5};
int* p = a;
printf("%d\n", p[3]);       // Prints the fourth element (14).
printf("%d\n", *(p +3)); // Does exactly the same.
```

Zeiger - Adressoperator 6/9

UNI FREIBURG

- Der Adressoperator & liefert die Adresse einer Variablen.

```
int v = 17;
int* p = &v; // get address of v.
*p = 42; // overwrite value of v!
printf("%d\n", v);  // Will print 42.
```

- Der Adressoperator kann die Adresse eines Feldelements liefern:

```
int a[5] = {3, 9, 2, 14, 5};
int* p = &a[3];  // Address of fourth element (14).
*p = 999;// Overwrites the fourth element (14).
printf("%d\n", a[3]);          // Prints 999.
```

- Mit & gewonnene Adressen **nicht** abspeichern, **nicht** als Rückgabewert verwenden! (Erklärung folgt…)

Zeiger vs 2D Felder 7/9

UNI FREIBURG

- Der Name eines zwei-dimensionales Feldes ist ein Zeiger auf ein ein-dimensionales Feld

```
int b[4][2] = { {0, 1}, {10, 11}, {20, 21}, {30, 31} };
int (*q)[2] = b;                // Pointer to array of size 2.
printf("%d\n", sizeof(q));      // Prints 8 (size of an address).
printf("%d\n", b[3][1]);        // Prints 31.
printf("%d\n", q[3][1]);        // Exactly the same.
```

- Wenn wir die Adresse des ersten Elementes verwenden, geht die 2D-Arithmetik verloren:

```
int* p = &b[0][0];              // Same address as b and q.
printf("%d\n", p[3][1]);        // Does not compile.
printf("%d\n", p[7]);           // This does, and prints 31.
```

Zeiger - Strings 8/9

UNI FREIBURG

- Eine **Zeichenkette** ist ein Feld bzw. Zeiger ... und zwar von Elementen vom Typ **char** = 1 Zeichen

```
char a[5] = {'f, 'o', 'o', 0};

char* p = a + 2;  // Points to the cell containing the 'o'.
```

- Einfacher geht‘s so zu initialisieren

```
const char* s = “foo";  // s points to the byte with the 'f'.
```

Ohne das const gibt es eine Compiler-Warnung

- Strings in C/C++ sind **null-terminiert**, d.h. bei “foo" wird Platz für **vier** Zeichen gemacht mit **0** am Ende
- Achtung Fehlerquelle: vergessene 0 am Stringende!!

Zeiger 9/9

UNI FREIBURG

- Felder von Zeichenketten
  - Das erklärt den Typ von **argv** in der main Funktion

    int main(int argc, **char** argv**)
  - Und zwar ist **char**** ein Zeiger auf Werte vom Typ **char***
  - Mit anderen Worten: **argv** ist ein Feld von Zeigern, die auf Zeichenketten (die Felder von Zeichen sind) zeigen

    Und **argc** wie viele Elemente das Feld hat
- Generell in Funktionssignaturen
  - (type *var) ist dasselbe wie (type var[]), also:

    int main(int argc, **char* argv[]**)

    bedeutet das gleiche wie die Signatur oben

Zeiger 10/9

UNI FREIBURG

- Nullpointer
  - Zeigervariablen dürfen den Wert Null annehmen (Nullpointer → verwende Makro NULL), aber ein Zugriff darüber ist nicht erlaubt und liefert eine 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

```
int* p = NULL;    // Pointer to address 0.
*p = 42;          // Produces a segmentation violation.
```

Zeiger 11/9

- Der Billion-Dollar Mistake
  - C.A.R. Hoare ist Erfinder von QuickSort, Hoare Logic, CSP, Monitoren und … vom Nullpointer!
  - Er hat sich 2009 dafür entschuldigt:
  - I call it my **billion-dollar mistake**. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). **My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler**. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Dynamische Speicherallokation 1/4

UNI FREIBURG

- Was, wenn die Größe eines Feldes vorab nicht bekannt ist?
  - Zum Beispiel, wenn wir zwei unbekannte Strings verketten wollen?

    ```
    char *mconcat(const char*s1, const char* s2);
    ```

  - Wir könnten einen großen Puffer anlegen und die Verkettung nur ausführen, wenn genug Platz ist

    ```
    char buffer[1024] = {0};  // Large buffer.
    Assert(strlen(s1) + strlen(s2) < 1024); // Large enough?
    ```

  - Mäh! Weiteres Problem: jeder Aufruf der Funktion verwendet den gleichen Puffer…
  - Brauchen einen Mechanismus, der zur Laufzeit einen neuen Puffer der gewünschten Größe anlegt

Dynamische Speicherallokation 2/4

UNI FREIBURG

- malloc(int n) liefert einen bislang unbenutzten Speicherbereich von n Bytes
  - Zum Verketten der Strings s1 und s2:

```
#include <string.h>

int n = strlen(s1) + strlen(s2) + 1; // compute #bytes needed
```

  - Ein Extra-Byte für die Null am Ende!
  - Wir legen einen Puffer für genau n Zeichen an:

```
#include <stdlib.h>
char * buffer = malloc(n * sizeof(char));
```

  - Den können wir wie ein Feld benutzen, erreichbar über die Zeigervariable buffer.
  - Zugriffe mit Index <0 oder >=n liefern undefiniertes Ergebnis und müssen daher vermieden werden!

Dynamische Speicherallokation 3/4

UNI FREIBURG

- free() gibt einen Speicherbereich frei
  - Der Speicherbereich muss vorher durch malloc() alloziert worden sein:

```
#include <stdlib.h>

free(buffer);
```

  - Danach darf dieser Speicherbereich nicht mehr referenziert werden:

```
buffer[0] = 0;   // not allowed after free.
```

  - Free() nicht vergessen („memory leak“), aber auch nicht zu früh (Laufzeitfehler, Speicherbereich kann wieder verwendet werden).
  - Achtung Fehlerquelle!

Dynamische Speicherallokation 3/4

UNI FREIBURG

- sizeof() liefert Größe eines Typs/Variable in Bytes
  - Mit den Deklarationen:

```
char a;          char b[1];
char* c;         char d[9];
```

  - Liefert sizeof die folgenden Ergebnisse:

```
sizeof(char) == 1;  // size of a character
sizeof(char *);     // size of a pointer (impl dependent)
sizeof(a) == 1;     // size of a's type
sizeof(b) == 1;     // array of one character
sizeof(c) == sizeof(char *); // pointer
sizeof(d) == 9;     // array of nine characters
```

Struct 1/5

UNI FREIBURG

- Ein Struct ist eine Aggregat-Datenstruktur
  - Feld
    - bestimmte Anzahl von Variablen gleichen Typs
    - Zugriff über Index
  - Struct
    - Bestimmte Anzahl von Variablen verschiedenen Typs
    - Zugriff über Namen
    - Vgl. Python Objekte

Struct 2/5

UNI FREIBURG

- Beispiel: Person
  - Ein struct Typ:

```
struct person {
  char *name;    // components of the struct.
  unsigned int age;
} john = { "John Doe", 45 };
```

  - Deklariert Variable john vom Typ struct person; Zugriff auf Komponenten erfolgt mit .name bzw. .age (Punktnotation)

```
printf("Person %s is %d years old\n",
        john.name, john.age);
```

Struct 3/5

- Tags und Typen
  - person ist ein Tag, kein Typ
  - Nach der ersten Verwendung eines Tags können weitere Variablen mit struct person var definiert werden

    ```
    struct person mary = { "Mary Jane", 36 };
    ```

  - Es gibt Zeiger auf structe und auf Komponenten

    ```
    struct person *p = &mary;

    int * marys_age = &mary.age;

    printf("Mary's age is %d\n", *marys_age); // prints 36.
    ```

Struct 4/5

UNI FREIBURG

- Ein frisches Struct
  - Mit malloc() kann auch eine “frische“ struct Variable erzeugt werden:

```
struct person * p = malloc(sizeof(struct person));
(*p).name = “Johnny Depp“;
(*p).age = 57;
printf(“%s is %d years old\n“, (*p).name, (*p).age); //
prints “Johnny Depp is 57 years old“.
```

Struct 5/5

UNI FREIBURG

- Besser lesbar mit dem Pfeiloperator →

```
struct person * p = malloc(sizeof(struct person));
p->name = "Johnny Depp";
p->age = 57;
printf("%s is %d years old\n", p->name, p->age); //
prints "Johnny Depp is 57 years old".
```

  - Gleiche Bedeutung: p->name vs (*p).name!
  - Noch eine Alternative:

```
*p = { .name = "Johnny Depp", .age = 57 }
```

Literatur / Links

UNI FREIBURG

- Zeiger / Pointers / pointer arithmetic
  - https://www.tutorialspoint.com/cprogramming/c_pointers.htm
- Zeichenketten / Strings
  - https://www.tutorialspoint.com/cprogramming/c_strings.htm
- Structures
  - https://www.tutorialspoint.com/cprogramming/c_structures.htm
- Zufallszahlen
  - https://en.wikipedia.org/wiki/Random_number_generation