Programmieren in C

SS 2019

Vorlesung 4, Dienstag 14. Mai 2019

(Strings, Zeiger, Allokation, Strukturen)

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 Ü3 — Wumpus und Freunde
- Inhalt
  - Zeiger — Operatoren * und &
  - Dynamischer Speicher — malloc und free
  - Strukturen — struct
  - **Übungsblatt 4:** Weiterentwicklung "Wumpus" und Fingerübungen

Erfahrungen mit dem Ü3 1/2

UNI FREIBURG

- Zusammenfassung / Auszüge
  - Die Datei heißt „Erfahrungen.txt“ …

    Bitte frühzeitig einchecken, damit wir sie Montagabend sehen!
  - Hoher Zeitaufwand

    Wir haben das aktuelle Blatt angepasst.
  - Schöne interessante Aufgaben (insbes. Wumpus)

    Da bleiben wir noch dran.
  - Zu unklare Aufgabenstellungen.

    Das meiste wurde im Forum geklärt. Wir machen jetzt genauere Vorgaben.

Erfahrungen mit dem Ü3 2/2

UNI FREIBURG

- Zusammenfassung / Auszüge
  - Unstrukturierte Vorlesung

    Manche Dinge müssen erstmal unvollständig behandelt werden, wenn man nicht mit Trockenübungen anfangen will. Die Vorlesung ist kein Lehrbuch und ersetzt auch nicht die Dokumentation.
  - Probleme mit Makefiles

    Sind jetzt hoffentlich behoben. Muster sind verfügbar.
  - Mehr Vergleiche Python vs C.

    Das werden wir aufgreifen.

Zeiger 1/9

UNI FREIBURG

- Zeiger
  - **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.
```

  - Man kann eine Zahl zu einem Zeiger dazu addieren ... die wird dann 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).
```

Zeiger 2/9

UNI FREIBURG

- Zeigerarithmetik
  - Mit Zeigern kann man rechnen…
  - Man kann eine Zahl zu einem Zeiger dazu addieren ... die wird dann 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 Konstante oder Variable sein.
  - Auch negativ…

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

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

Zeiger 3/9

UNI FREIBURG

- Zeigerarithmetik - Vergleiche
  - 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 - Warnung
  - 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 5/9

UNI FREIBURG

- Zeiger vs. ein-dimensionale Felder in C / C++
  - Den Namen eines ein-dimensionalen Feldes ist wie ein Zeiger auf das erste Element des Feldes

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

  - Der Zugriff über […] macht **exakt** dasselbe wie 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 6/9

UNI FREIBURG

- Der Adressoperator
  - 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.
```

Zeiger 7/9

UNI FREIBURG

- Zeiger vs. zwei-dimensionale Felder in C / C++
  - Den Namen eines zwei-dimensionales Feldes kann man benutzen wie einen 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.
```

  - Man könnte auch einfach die Adresse des ersten Elementes nehmen, verliert dann aber die 2D-Arithmetik:

```
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 8/9

UNI FREIBURG

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

    ```
    char a[5] = {'D', 'o', 'o', 'f', \x00};

    char* p = a + 3;  // Points to the cell containing the 'f'.
    ```

  - Einfacher geht's so zu initialisieren

    ```
    const char* s = "Doof";  // s points to the byte with the 'D'.
    ```

    Ohne das const gibt es eine Compiler-Warnung

  - Strings in C/C++ sind **null-terminated**, d.h. bei "Doof" wird Platz für **fünf** Zeichen gemacht, und am Ende steht **\x00**

    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

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?
    ```

  - 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
```

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

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

  - Den können wir jetzt wie ein Feld benutzen, aber 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
```

Structs 1/4

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, named tuples

Structs 2/4

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 struct Typ; Zugriff auf Komponenten mit .name bzw. .age

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

Structs 3/4

- 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 structs und auf Komponenten

    ```
    struct person *p = &mary;

    int * marys_age = &mary.age;

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

Structs 4/4

UNI FREIBURG

- Frische Structs
  - Mit malloc() kann auch eine "frische" struct Variable erzeugt werden:

```
struct person * p = malloc(sizeof(struct person));
(*p).name = "Johnny Depp";
(*p).age = 55;
printf("%s is %d years old\n", (*p).name, (*p).age); //
prints "Johnny Depp is 55 years old".
```

  - Besser lesbar anstatt (*p).name: p->name

```
printf("%s is %d years old\n", p->name, p->age); //
prints the same as above.
```

  - Exakt die gleiche Bedeutung

Hinweise zur Übung 1/2

UNI FREIBURG

- (Pseudo) Zufallszahlen
  - Pseudozufallszahlen werden durch einen Generator erzeugt, der immer wieder das gleiche Verfahren (oft eine lineare Kongruenz) auf einen internen Zustand anwendet.
  - Lineare Kongruenz
    - Xn+1 = (a Xn + b) mod m
    - Xn Folge der Zufallszahlen, Xn+1 nächste Zahl

Hinweise zur Übung 2/2

UNI FREIBURG

- (Pseudo) Zufallszahlen
  - In C erfolgt der Zugriff darauf mit

    ```
    int rand(void);
    ```

  - Will man eine reproduzierbare Folge, so kann man zu Beginn den Zustand des Generators setzen (“seeden“). Dies geht mit

    ```
    void srand(unsigned int seed);
    ```

  - Der seed bestimmt die Folge der Werte aller nachfolgenden Aufrufe von rand()
  - Standardseed, automatisch zu Beginn des Programms

    ```
    srand(1);
    ```

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