



# C Programmierkurs

## 6. Stunde: Pointer, Datenstrukturen, dynamischer Speicher

---

Johannes Hayeß & Mirko Seibt

29. November 2018

Technische Universität Dresden

Pointer

Wieso funktioniert das?

```
int main(void) {  
    char input_buf[64] = {};  
    printf("Enter text consisting of any letter A-Z:\n");  
    fgets(input_buf, 64, stdin);  
    rot13(64, input_buf);  
    printf("%s", input_buf);  
    return EXIT_SUCCESS;  
}
```

Wir verändern input\_buf ja eigentlich nicht! Oder doch?

Dereferenz-Operator: \*

Adress-Operator: &

# Pointer in C

Dereferenz-Operator: \*

Adress-Operator: &

Deklaration:

```
[Typ] *[Pointer];
```

# Pointer in C

Dereferenz-Operator: \*

Adress-Operator: &

Deklaration:

```
[Typ] *[Pointer];
```

Zuweisung:

```
[Pointer] = &[Variable];
```

& gibt uns die Speicheradresse der Variable zurück

Pointer verändern:

```
[Pointer] = [Wert];
```

Ändert die Speicheradresse auf die der Pointer zeigt.

Pointer verändern:

```
[Pointer] = [Wert];
```

Ändert die Speicheradresse auf die der Pointer zeigt.

Speicher ändern:

```
*[Pointer] = [Wert];
```

Ändert den Inhalt an der Speicheradresse auf die der Pointer zeigt



Strings

# Strings als Pointer

Strings sind in Wahrheit Pointer!

```
char string[];  
char *string;
```

Gespeichert wird die Adresse der ersten Speicherzelle mit welcher der String beginnt.

Jeder String ist 0-terminierend, daher ist mit der Startadresse der String eindeutig bestimmt.

Inkrement/Dekrement:

```
char *string = "Hello World!";  
for(char *i = string; *i != 0; ++i) {  
    printf("%c", *i);  
}
```

**Wichtig:** ++i verschiebt den Pointer um die Größe des Pointertyps

```
char = 1 Byte, int = 4 Byte, long = 8 Byte, ...
```

# Pointer Arithmetik

Inkrement/Dekrement:

```
int number = 41; int *pointer = &a;  
(*pointer)++; // number == 42
```

# Pointer Arithmetik

Inkrement/Dekrement:

```
int number = 41; int *pointer = &a;  
(*pointer)++; // number == 42
```

Funktionen:

```
int *getAddress(void) {  
    int number = 1337;  
    return &number;  
}
```

Pointerwerte:

```
int number = 1337; int *pointer = &number;  
printf("%p", pointer); // like 0x2a
```

# Datenstrukturen

# Datenstrukturen - Definition

Schlüsselwort: **struct**

Definition:

```
struct [SName] {  
    [Typ] [VName];  
    // more declarations  
};
```

# Datenstrukturen - Definition

Schlüsselwort: **struct**

Definition:

```
struct [SName] {  
    [Typ] [VName];  
    // more declarations  
};
```

Beispiel:

```
struct student {  
    char *name;  
    int mat_nr;  
};
```



Deklaration:

```
struct [SName] [Name1], [Name2], ...;
```

Deklaration:

```
struct [SName] [Name1], [Name2], ...;
```

Beispiel:

```
struct student johannas, mirko;
```

Deklaration:

```
struct [SName] [Name1], [Name2], ...;
```

Beispiel:

```
struct student johannas, mirko;
```

Kombination:

```
struct student {  
    char *name;  
    int mat_nr;  
} johannes, mirko;
```

Zuweisung:

```
[VName] = { [Wert1], [Wert2], ... };
```

Zuweisung:

```
[VName] = { [Wert1], [Wert2], ... };
```

Beispiel:

```
johannes = { "Johannes", 421337};
```

Zuweisung:

```
[VName] = { [Wert1], [Wert2], ... };
```

Beispiel:

```
johannes = { "Johannes", 421337};
```

Kombination:

```
struct student johannes = { "Johannes", 421337};
```

Beispiel:

```
struct student {  
    char *name;  
    int mat_nr;  
} johannes;  
johannes.name = "Johannes";  
johannes.mat_nr = 421337;  
printf("%s\n%d\n", johannes.name, johannes.mat_nr);
```

# Enumeration



# Enumeration - Deklaration

Schlüsselwort: **enum**

```
enum [ENAME] {  
    [NAME1] = [Wert1 | 0],  
    [NAME2] = [Wert2 | 1],  
    ...  
}
```

# Enumeration - Deklaration

Schlüsselwort: **enum**

```
enum [EName] {  
    [NAME1] = [Wert1 | 0],  
    [NAME2] = [Wert2 | 1],  
    ...  
}
```

Beispiel:

```
enum traffic_colors {  
    RED = 1, // value == 1  
    YELLOW, // value == 2  
    GREEN // value == 3  
};
```

Geeignet für globale Konstanten usw.

# Enumeration - Zuweisung

Variablen vom Typ enum können nur definierte Werte annehmen:

```
enum traffic_colors red = RED;  
enum traffic_colors yellow = 2;
```

# Enumeration - Zuweisung

Variablen vom Typ enum können nur definierte Werte annehmen:

```
enum traffic_colors red = RED;  
enum traffic_colors yellow = 2;
```

Kombination:

```
enum traffic_colors {  
    RED = 1, // value == 1  
    YELLOW, // value == 2  
    GREEN // value == 3  
} red = RED;
```

Enumerationen sind einfache Integertypen also geeignet für  
**Switch**

typedef

Schlüsselwort: **typedef**

```
typedef struct student {  
    char *name;  
    int mat_nr;  
} student;
```

Erstellt einen neuen Typ **student**

Schlüsselwort: **typedef**

```
typedef struct student {  
    char *name;  
    int mat_nr;  
} student;
```

Erstellt einen neuen Typ **student**

**Vorsicht:** Verschleiert für Andere das es sich dabei um einen zusammengesetzten Typ handelt!

```
student johannes = { "Johannes", 421337};
```

# Dynamische Speicherallokation



stdlib.h bietet folgende Funktionen zur Speicherallokation an:

**malloc()** : Allokiert einen neuen Block

**calloc()** : Allokiert einen neuen Block und initialisiert ihn

**realloc()** : Ändert die Größe eines Blocks

**free()** : Gibt einen Block wieder frei

# Speicherallokation - malloc()

Definition von malloc():

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

Neuen Speicher allokieren:

```
int *new_int = malloc(sizeof *new_int); // 4 Byte  
int *new_char = malloc(sizeof (char)); // 1 Byte  
int *new_array =  
    malloc(42 * sizeof *new_array); // 168 Byte
```

# Speicherallokation - calloc()

Definition von calloc():

```
void* calloc(size_t num, size_t size);
```

Neuen Speicher allokieren:

```
size_t element_count = 42;  
int *new_array =  
    calloc(element_count, sizeof *new_array);
```

Der neue allokierte Speicher wird mit 0 initialisiert.

# Speicherallokation - realloc()

Definition von realloc():

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

Speichergröße ändern:

```
int *new_array = malloc(10 * sizeof *new_array);  
int *new_array_big =  
    realloc(new_array, 20 * sizeof *new_array);
```

Neuer Speicher wird nicht initialisiert, und der alte Pointer wird ggf. gelöscht.

# Speicherallokation - free()

**Wichtig:** Dynamisch allozierter Speicher wird **nicht** automatisch freigegeben!

Definition von free():

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

Speicher freigeben:

```
int *new_array = malloc(10 * sizeof *new_array);  
free(new_array);
```

Diese Präsentation ist lizenziert unter der **Creative Commons Attribution-ShareAlike 4.0 International** Lizenz.

