

KONZEPTE SYSTEMNAHER PROGRAMMIERUNG

Technische Hochschule Mittelhessen

Andre Rein

– Lose Enden –

SCHLÜSSELWORT `static`

Dass Schlüsselwort `static` hat in `C` unterschiedliche Bedeutungen, je nachdem **wo** es im Code verwendet wird.

- Bei Variablendeklarationen **innerhalb von Funktionen** (lokale Variablen) bewirkt das Schlüsselwort `static` das Anlegen des Speicherplatzes im **statischen Datensegment** und **nicht** auf dem Stack! (*wurde bereits in Session 5 angesprochen*)
- Bei Variablendefinitionen **außerhalb von Funktionen** (globale Variablen) und bei Funktionsdefinitionen *beschränkt* das Schlüsselwort `static` die Sichtbarkeit des Namens auf die Quelltextdatei, in der die Definition steht.
 - Somit kann verhindert werden, dass ein Zugriff auf so definierte Funktionen oder Variablen, von anderen Dateien (compilation units), erfolgen kann.
 - Ähnlich zu dem Konzept `privater` -Daten/Methoden in OO-Sprachen.

static INNERHALB VON FUNKTIONEN

```
#include <stdio.h>

int f(void){
    static int i=0; ❶
    i++;
    return i;
}

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

    for (int i = 0; i < 8; ++i) {
        printf("loop [%d] f()=%d\n", i, f()); ❷
    }
    return 0;
}
```

```
$ gcc -Wall -g global_static.c -o global_static
$ ./global_static
loop [0] f()=1
loop [1] f()=2
loop [2] f()=3
loop [3] f()=4
loop [4] f()=5
loop [5] f()=6
loop [6] f()=7
loop [7] f()=8
```

- ❶ Schlüsselwort `static` - Initialisierung `int i=0;` erfolgt **einmalig!**
- ❷ Mehrfachaufruf von `f()`

- Ein Zugriff auf `i` kann nur aus der Funktion `f()` erfolgen.
- Die Variable liegt im **statischen Datenbereich** und behält den Wert über die Funktionsaufrufe hinaus → **verhält sich statisch**

static AUSSERHALB VON FUNKTIONEN

static.h

```
#ifndef STATIC_H
#define STATIC_H

void do_something(int);

#endif /*STATIC_H*/
```

static.c

```
#include <stdio.h>

static int counter=0;
static int add_to_counter(int x) {
    return counter+x;
}

void do_something(int what) {
    int tmp=counter;
    counter=add_to_counter(what);
    printf("counter [%2d]->[%2d]\n", tmp, counter);
}
```

main.c

```
#include <stdio.h>
#include "static.h"

int main(void) {
    for (int i=0; i<5; i++){
        do_something(5);
    }
}
```

```
$ gcc -o main main.c static.c; ./main
counter [ 0]->[ 5]
counter [ 5]->[10]
counter [10]->[15]
counter [15]->[20]
counter [20]->[25]
```

main2.c

```
#include <stdio.h>
#include "static.h"

extern int counter;

int main(void) {
    counter=3;
    for (int i=0; i<5; i++){
        do_something(4);
    }
}
```

```
$ gcc -o main2 main2.c static.c; ./main2
/usr/bin/ld: main2.o: warning: relocation against `
`counter' in read-only section `.text'
/usr/bin/ld: main2.o: in function `main':
main2.c:(.text+0xa): undefined reference to `counter'
...
```

`static` AUSSERHALB VON FUNKTIONEN

```
$ gcc -c main2.c; gcc -c static.c; gcc -o main2 main2.o static.o;  
/usr/bin/ld: main2.o: warning: relocation against \  
    `counter' in read-only section `.text'  
/usr/bin/ld: main2.o: in function `main':  
main2.c:(.text+0xa): undefined reference to `counter'  
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE  
collect2: error: ld returned 1 exit status
```



Wie man sieht hat man keinen **direkten** Zugriff auf die Variable `counter` aus einer anderen Datei als `static.c`.

Gleiches gilt auch für die Funktion `add_to_counter()`, selbst wenn diese im Header `static.h` deklariert gewesen wäre.

ZEIGERARITHMETIK

Wenn `p` ein Zeiger auf ein Objekt vom Typ `T` und `n` eine ganze Zahl ist, dann ist `p+n` und `p-n` auch ein Zeiger auf ein Objekt vom Typ `T`, das sich `n`-Objekte (**nicht Bytes!**) weiter hinten oder vorne im Speicher befindet.

Beispiel:

```
int a[6];  
int *p=&a[2];
```

	[0]	[1]	[2]	[3]	[4]	[5]
a						
	p-2	p-1	p	p+1	p+2	p+3

```
p+2;      // zeigt auf a[4]  
*(p+1)=9; // weist a[3] den Wert 9 zu.
```

$$\begin{aligned}a &\equiv \&a[0] \\ a + n &\equiv \&a[n] \\ *(a + n) &\equiv a[n]\end{aligned}$$

$$\begin{aligned}p &\equiv \&a[2] \\ p + 1 &\equiv \&a[3] \\ p[2] &\equiv *(p + 2) \equiv a[4]\end{aligned}$$

ZEIGERARITHMETIK: ANWENDUNG

pointer.c

```
#include <stdio.h>

#define SIZE 10

void print_array(int *p, char * array_name) {
    int i=0;
    char * tmp="";
    while (i<SIZE){
        tmp = (i==SIZE-1) ? "\n": " ";
        printf("%s[%d]=%d%s", array_name, i, *(p+i), tmp); ❶
        i++;
    }
}

int main(void) {

    int a[SIZE] = {0,1,2,3,4,5,6,7,8,9};
    int b[SIZE] = {0,0,0,0,0,0,0,0,0,0};

    int *p_a=&a[0];
    int *p_b=&b[9];

    print_array(&a[0], "a");
    print_array(&b[0], "b");
    puts("");

    for (int i=0; i<SIZE; i++){
        *p_b-- = *p_a++; ❷
    }
    print_array(&a[0], "a");
    print_array(&b[0], "b");
    return 0;
}
```

```
$ gcc -o pointer pointer.c && ./pointer
a[0]=0 a[1]=1 a[2]=2 a[3]=3 a[4]=4 a[5]=5 a[6]=6 a[7]=7 a[8]=8 a[9]=9
b[0]=0 b[1]=0 b[2]=0 b[3]=0 b[4]=0 b[5]=0 b[6]=0 b[7]=0 b[8]=0 b[9]=0

a[0]=0 a[1]=1 a[2]=2 a[3]=3 a[4]=4 a[5]=5 a[6]=6 a[7]=7 a[8]=8 a[9]=9
b[0]=9 b[1]=8 b[2]=7 b[3]=6 b[4]=5 b[5]=4 b[6]=3 b[7]=2 b[8]=1 b[9]=0
```

- Zugriff auf Arrayfelder mittels Zeigerarithmetik.

```
printf("%s[%d]=%d%s",
        array_name, i, *(p+i), tmp); ❶
```

- Umsortierung der Arrayinhalte mittels Zeigerarithmetik.

```
for (int i=0; i<SIZE; i++){
    *p_b-- = *p_a++; ❷
}
```

FUNKTIONSZEIGER

Wenn `T f(. . .) { }` eine Funktionsdefinition ist, dann ist `f` ein Zeiger auf diese Funktion.

- Dieser Zeiger kann als Argument an andere Funktionen übergeben werden,
- von Funktionen zurückgegeben werden, oder
- auch in Variablen abgespeichert werden: `p=f ;`, wobei die Variablendeklaration `T (*p)(. . .)` lauten muss.
 - Zum Aufruf der in `p` hinterlegten Funktion verwendet man: `(*p)(. . .)`

FUNKTIONSZEIGER: BEISPIEL 1

```
#include <stdio.h>

int add(int a, int b) {
    return a+b;
}

int mul(int a, int b) {
    return a*b;
}

int expo(int a, int b) {
    int res;
    if (b==0)
        return 1;
    else
        res= a*expo(a, b-1);
}

int wrap(int (*func)(int, int), int x, int y){
    return (*func)(x,y);
}

int main(int argc, char *argv[]) {
    printf("wrap(add(2,8)) 2+8 = %3d\n",wrap( add, 2, 8));
    printf("wrap(mul(2,8)) 2*8 = %3d\n",wrap( mul, 2, 8));
    printf("wrap(expo(2,8)) 2^8 = %3d\n",wrap(expo, 2, 8));
    return 0;
}
```

```
$ gcc -o example_1 example_1.c && ./example_1
wrap(add(2,3)) 2+8 = 10
wrap(mul(2,3)) 2*8 = 16
wrap(expo(2,3)) 2^8 = 256
```

FUNKTIONSZEIGER: BEISPIEL FUNKTIONSAUSWAHL LAUFZEIT

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int add(int a, int b) {
    return a+b;
}

int mul(int a, int b) {
    return a*b;
}

typedef struct {
    char *name;
    char op;
    int (*func)(int, int);
} Command;

Command cmdList[] = {
    {"addition", '+', add},
    {"multiplication", '*', mul}
};

int execute(char *name, int arg1, int arg2){
    int i=0;
    int result;
    while (i<(sizeof(cmdList)/sizeof(cmdList[0]))){
        if(strcmp(name, cmdList[i].name) == 0){
            result=(*cmdList[i].func)(arg1, arg2);
            printf("Executing: %d %c %d = %d\n",
                arg1, cmdList[i].op, arg2, result);
            return result;
        }
        i++;
    }

    printf("Command [%s] not found!\n", name);
    return -1;
}
```

```
int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("./select <name> <int> <int>\n");
        return -1;
    }
    execute(argv[1], atoi(argv[2]), atoi(argv[3]));
    return 0;
}
```

```
$ gcc select.c -o select
$ ./select addition 7 10
Executing: 7 + 10 = 17
$ ./select multiplication 8 43
Executing: 8 * 43 = 344
$ ./select subtraction 9 2
Command [subtraction] not found!
```

FUNKTIONSAUFRUFE MIT VARIABLEN LANGER PARAMETERLISTE

Beispiel: `int error(char *fmt, ...)`. Der Zugriff auf die Argumente in `...` erfolgt mittels Makros, die in `stdarg.h` definiert sind.

Typische Verwendung: Eigene Fehlerausgabe soll so verwendet werden können wie `printf()`

```
#include <stdio.h>
#include <stdarg.h>

void error(char *fmt, ...){
    va_list ap;
    va_start(ap, fmt);
    printf("xxxErrorxxx: ");
    vprintf(fmt, ap);
    printf("\n");
}

int main(int argc, char *argv[])
{
    error("[%s] [%s] [%s] [%s] [%s] [%s]", "a", "b", "c", "c", "c", "c");
    error("[%s] one argument", __func__);
    error("[%s] two arguments #%02x", __func__, 42);
    error("[%s] three arguments #%02x, %s", __func__,
        0x127, "three");
    return 0;
}
```

```
$ gcc -o var_arguments var_arguments.c
$ ./var_arguments
Error: [main] one argument
Error: [main] two arguments #2a
Error: [main] three arguments #127, three
```