

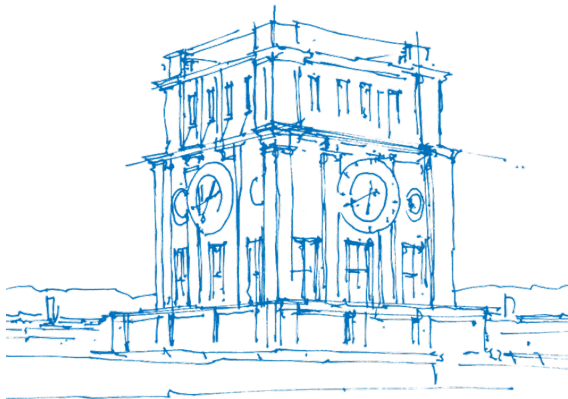
Grundlagen: Betriebssysteme und Systemsoftware

Tutorübung

Mario Delic

Lehrstuhl für Connected Mobility
School of Computation, Information and Technology
Technische Universität München

Übungswoche 2



TUM Uhrenturm

Begrifflichkeiten

Betriebsarten

- **Stapelverarbeitung:**
Keine Nutzerinteraktion, der Ablauf des Programms ist bereits definiert. Anwendung oft bei großen/monotonen Aufgabenmengen.
- **Transaktionsbetrieb:**
Muss Atomarität, Konsistenz etc. gewährleisten. Zu finden z.B. bei Datenbanken.
- **Dialogbetrieb:**
Benutzer und BS interagieren miteinander. Beispielsweise Desktopbetriebssysteme.
- **Echtzeitbetrieb:**
Eine maximale Reaktionszeit darf nicht überschritten werden. Zu finden z.B. in der Robotik.

Begrifflichkeiten

BS Typen

Monolithische Systeme: BS als ein großes Programm ausgeführt.

- BS permanent im Kernel-Mode (privilegierter Modus) und Arbeitsspeicher.
- Möglichkeit für ein breites Funktionsspektrum und viel Flexibilität.
- Weniger Strukturiert, somit schwerer zu Warten.
- Anfällig für schwere crashes.

Mikrokernel Systeme: Kleinerer Kern durch weniger Funktionen.

- Reduzierung der Fehleranfälligkeit durch aufteilen in Module. Nur der Mikrokern im Kernel-Mode.
- Mikrokern bieten nur noch Basismechanismen. Er ist kleiner, sicherer und leichter wartbar.
- Subsysteme laufen im User-Modus ohne Privilegien.

Linux

Standardfunktionen

Eine Reihe von essentiellen Funktionen die in der Vorlesung vorgestellt wurden:

- **cat someFile**: Gibt Inhalt von someFile aus.
- **cat >someFile**: Erstelle someFile (auch mögl. mit **touch**) und schreibe in sie.
- **grep stringToMatch someFile**: Printed alle Zeilen aus someFile, die (teilweise) mit stringToMatch matchen. Verschiedene Modifier: * = Durchsuche Verzeichnis (statt Datei); **-i** = case insensitive; **-v** = Zeilen, die NICHT matchen.
- **mkdir someDir**: Erstellt das Verzeichnis someDir.
- **cd pathToDir**: Wechselt in das Verzeichnis unter dem Pfad pathToDir. (\= root directory; . = current directory; .. = parent directory).
- **pwd**: Gibt aktuelles Verzeichnis aus.
- **ls -la**: **ls**: gibt Inhalt des Verzeichnisses aus; **-l**: ausführliche Liste; **-a**: zeigt hidden files.

Weitere Funktionen und Systemcalls

- **int open (const char* pathToFile, int flags):** Öffnet Datei und gibt einen zugehörigen filedescriptor zurück, welcher von read/write/close verwendet werden kann. Flags: O_RDONLY, O_WRONLY, O_RDWR, O_CREAT uvm.
- **int close (int filedescriptor):** Schließt eine von open geöffnete file. Returned 0 wenn erfolgreich, sonst -1.
- **ssize_t write (int filedescriptor, const void *buf, size_t bytesToWrite):** Schreibt bytesToWrite-viele Bytes aus dem Inhalt eines Puffers buf in eine von open geöffnete file. Returned die Anzahl an geschriebenen Bytes; -1 bei Fehler.
- **ssize_t read (int filedescriptor, void *buf, size_t bytesToRead):** Liest bytesToRead-viele Bytes aus einer geöffneten file raus und schreibt diese in buf. Return ist die Anzahl an gelesenen Bytes; -1 bei Fehler.
- **off_t lseek (int filedescriptor, off_t offset, int whence):** Verschiebt das file-offset (=aktuelle Lese-/Schreibposition der file). **whence:** SEEK_SET: Dateistart; SEEK_CUR: Aktuelle Position; SEEK_END: Dateiende. **offset:** Auf whence addierte Bytes

Operatorpräzedenz

C-Lesen

Zur Evaluierung und Veranschaulichung der Operatorreihenfolge kann man die relevanten Strukturen jeweils 1:1 in die englische Sprache überführen:

- **x**: “declare x as” oder “x is”
- *****: “pointer to”
- **&**: “reference to”
- **[]**: “array of” $\hookrightarrow$ [length]: “array of length”
- **()**: “function returning” $\hookrightarrow$ (type): “function (with type argument) returning”

Operatorpräzedenz

In C gelten u.a die folgenden Präzedenzregeln:

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(type){list}	Compound literal(c99)	
2	++ --	Prefix increment and decrement ^[note 1]	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of ^[note 2]	
	_Alignof	Alignment requirement(c11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	

```
char *efg[5]();
```

```
short (**(*hij)(void));
```

https://en.cppreference.com/w/c/language/operator_precedence

Operatorpräzedenz

In C gelten u.a die folgenden Präzedenzregeln:

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(type){ list }	Compound literal(c99)	
2	++ --	Prefix increment and decrement ^[note 1]	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of ^[note 2]	
	_Alignof	Alignment requirement(c11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	

`char *efg[5]();`
efg is array of 5 function returning pointer to char

`short (**(*hij)(void));`

https://en.cppreference.com/w/c/language/operator_precedence

Operatorpräzedenz

In C gelten u.a die folgenden Präzedenzregeln:

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(type){ list }	Compound literal(c99)	
2	++ --	Prefix increment and decrement ^[note 1]	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of ^[note 2]	
	_Alignof	Alignment requirement(c11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	

`char *efg[5]();`
efg is array of 5 function returning pointer to char

`short (**(*hij)(void));`
hij is pointer to function with no arguments returning pointer to pointer to short

https://en.cppreference.com/w/c/language/operator_precedence

Aufgabe 1

Operatorpräzedenz

[] (Array) und () (Funktion) werden von links nach rechts abgearbeitet und haben Vorrang vor * (Pointer/Dereferenzierung) und & (Adresse), welche von rechts nach links abgearbeitet werden. Bestimmen Sie davon ausgehend, wie die folgenden Ausdrücke gelesen werden:

- a) `long **foo[7];`
- b) `unsigned long int **x();`
- c) `char *(*(**foo[][8])())[];`
- d) `int (*(*foo)(void))[];`

Aufgabe 1

Operatorpräzedenz

[] (Array) und () (Funktion) werden von links nach rechts abgearbeitet und haben Vorrang vor * (Pointer/Dereferenzierung) und & (Adresse), welche von rechts nach links abgearbeitet werden. Bestimmen Sie davon ausgehend, wie die folgenden Ausdrücke gelesen werden:

a) `long **foo[7];`

foo is array of 7 pointer to pointer to long

b) `unsigned long int **x();`

c) `char *(*(**foo[][8])())[];` ´

d) `int (*(*foo)(void))[];`

Aufgabe 1

Operatorpräzedenz

[] (Array) und () (Funktion) werden von links nach rechts abgearbeitet und haben Vorrang vor * (Pointer/Dereferenzierung) und & (Adresse), welche von rechts nach links abgearbeitet werden. Bestimmen Sie davon ausgehend, wie die folgenden Ausdrücke gelesen werden:

a) `long **foo[7];`

foo is array of 7 pointer to pointer to long

b) `unsigned long int **x();`

x is function returning pointer to pointer to unsigned long int

c) `char *(*(**foo [] [8])())[];`

d) `int (*(*foo)(void))[];`

Aufgabe 1

Operatorpräzedenz

[] (Array) und () (Funktion) werden von links nach rechts abgearbeitet und haben Vorrang vor * (Pointer/Dereferenzierung) und & (Adresse), welche von rechts nach links abgearbeitet werden. Bestimmen Sie davon ausgehend, wie die folgenden Ausdrücke gelesen werden:

a) `long **foo[7];`

foo is array of 7 pointer to pointer to long

b) `unsigned long int **x();`

x is function returning pointer to pointer to unsigned long int

c) `char *(*(**foo [] [8])())[];`

foo is array of array of 8 pointer to pointer to function returning pointer to array of pointer to char

d) `int (*(*foo)(void))[];`

Aufgabe 1

Operatorpräzedenz

[] (Array) und () (Funktion) werden von links nach rechts abgearbeitet und haben Vorrang vor * (Pointer/Dereferenzierung) und & (Adresse), welche von rechts nach links abgearbeitet werden. Bestimmen Sie davon ausgehend, wie die folgenden Ausdrücke gelesen werden:

a) `long **foo[7];`

foo is array of 7 pointer to pointer to long

b) `unsigned long int **x();`

x is function returning pointer to pointer to unsigned long int

c) `char *(*(**foo[][8])())[];`

foo is array of array of 8 pointer to pointer to function returning pointer to array of pointer to char

d) `int (*(*foo)(void))[];`

foo is a pointer to function with no arguments returning pointer to array of int

Aufgabe 2

Hexdump

Gegeben sei ein Hexdump.

Außerdem:

- 32-bit Architektur
- Integer 32-bit breit
- little-endian

- a) Wie viele Hex-Zeichen umfasst eine Speicheradresse im obigen Hexdump?
- b) Nehmen wir an, an der Adresse 0x8c ist ein Pointer gespeichert. Wie lautet die Adresse, auf die der Pointer zeigt? Beachten Sie die Endianness der Architektur!

Aufgabe 2

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0x0000	89	50	4e	47	0d	0a	1a	0a	ff	00	00	00	49	48	44	52
0x0010	00	00	05	00	00	00	02	82	08	06	00	00	00	8e	3b	74
0x0020	aa	00	00	00	a7	73	42	49	54	08	08	08	08	7c	08	64
0x0030	47	42	53	00	88	00	00	00	4d	00	00	0f	61	00	00	0f
0x0040	61	01	a8	3f	a7	69	00	00	00	38	74	45	58	74	53	6f
0x0050	66	74	77	61	72	65	00	6d	61	74	70	6c	6f	74	6c	69
0x0060	f1	20	f5	65	72	73	69	6f	6e	33	2e	31	2e	31	2c	20
0x0070	68	74	74	70	3a	2f	73	74	72	69	6e	67	00	74	6c	69
0x0080	62	2e	6f	72	67	2f	10	66	17	19	00	00	20	00	49	44
0x0090	41	54	78	9c	ec	dd	77	9c	54	e5	a1	ff	f1	cf	6c	85
0x00a0	5d	76	97	be	85	de	62	69	6e	67	6f	00	a4	05	93	dc
0x00b0	18	63	bb	37	a2	49	ae	29	e6	17	d4	28	22	45	16	05
0x00c0	c4	44	62	12	4d	31	d7	80	b1	5f	5b	12	51	41	20	8a
0x00d0	08	00	00	00	39	05	00	00	b2	d4	2d	2c	db	cf	ef	8f
0x00e0	49	f6	c6	58	00	dd	e5	ec	ce	7e	de	af	d7	bc	5e	3c
0x00f0	87	39	33	df	59	75	9d	f9	ce	73	9e	27	12	04	41	80

20 00 49 44

Little
Endion!

0x44 49 00 20

Aufgabe 2

Hexdump

- c) Bestimmen Sie die Ausgaben des folgenden Programms, ausgeführt im Kontext des obigen Speicherausschnitts:

```
1    char *x = (char*) 0x30;  
2    int* i = (int*) 0xd0;  
3  
4    printf("Some_string:_%s\n", x);  
5    printf("Some_other_string:_%s\n", x+0x46);  
6  
7    int a = i[1];  
8    int b = *(int*)i;  
9    printf("a:_%d,_b:_%d\n", a, b);
```

Aufgabe 2

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0x0000	89	50	4e	47	0d	0a	1a	0a	ff	00	00	00	49	48	44	52
0x0010	00	00	05	00	00	00	02	82	08	06	00	00	00	8e	3b	74
0x0020	aa	00	00	00	a7	73	42	49	54	08	08	08	08	7c	08	64
0x0030	47	42	53	00	88	00	00	00	4d	00	00	0f	61	00	00	0f
0x0040	61	01	a8	3f	a7	69	00	00	00	38	74	45	58	74	53	6f
0x0050	66	74	77	61	72	65	00	6d	61	74	70	6c	6f	74	6c	69
0x0060	f1	20	f5	65	72	73	69	6f	6e	33	2e	31	2e	31	2c	20
0x0070	68	74	74	70	3a	2f	73	74	72	69	6e	67	00	74	6c	69
0x0080	62	2e	6f	72	67	2f	10	66	17	19	00	00	20	00	49	44
0x0090	41	54	78	9c	ec	dd	77	9c	54	e5	a1	ff	f1	cf	6c	85
0x00a0	5d	76	97	be	85	de	62	69	6e	67	6f	00	a4	05	93	dc
0x00b0	18	63	bb	37	a2	49	ae	29	e6	17	d4	28	22	45	16	05
0x00c0	c4	44	62	12	4d	31	d7	80	b1	5f	5b	12	51	41	20	8a
0x00d0	08	00	00	00	39	05	00	00	b2	d4	2d	2c	db	cf	ef	8f
0x00e0	49	f6	c6	58	00	dd	e5	ec	ce	7e	de	af	d7	bc	5e	3c
0x00f0	87	39	33	df	59	75	9d	f9	ce	73	9e	27	12	04	41	80

```
char *x = 0x30;
printf ("%s", x);
```

string = char*

47 42 53
g B S

Aufgabe 2

Hexdump

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0x0000	89	50	4e	47	0d	0a	1a	0a	ff	00	00	00	49	48	44	52
0x0010	00	00	05	00	00	00	02	82	08	06	00	00	00	8e	3b	74
0x0020	aa	00	00	00	a7	73	42	49	54	08	08	08	08	7c	08	64
0x0030	47	42	53	00	88	00	00	00	4d	00	00	0f	61	00	00	0f
0x0040	61	01	a8	3f	a7	69	00	00	00	38	74	45	58	74	53	6f
0x0050	66	74	77	61	72	65	00	6d	61	74	70	6c	6f	74	6c	69
0x0060	f1	20	f5	65	72	73	69	6f	6e	33	2e	31	2e	31	2c	20
0x0070	68	74	74	70	3a	2f	73	74	72	69	6e	67	00	74	6c	69
0x0080	62	2e	6f	72	67	2f	10	66	17	19	00	00	20	00	49	44
0x0090	41	54	78	9c	ec	dd	77	9c	54	e5	a1	ff	f1	cf	6c	85
0x00a0	5d	76	97	be	85	de	62	69	6e	67	6f	00	a4	05	93	dc
0x00b0	18	63	bb	37	a2	49	ae	29	e6	17	d4	28	22	45	16	05
0x00c0	c4	44	62	12	4d	31	d7	80	b1	5f	5b	12	51	41	20	8a
0x00d0	08	00	00	00	39	05	00	00	b2	d4	2d	2c	db	cf	ef	8f
0x00e0	49	f6	c6	58	00	dd	e5	ec	ce	7e	de	af	d7	bc	5e	3c
0x00f0	87	39	33	df	59	75	9d	f9	ce	73	9e	27	12	04	41	80

```
char *x = 0x30;
printf ("%s", x+0x46);
```

$$x = 0x30 + 0x46 = 0x76$$

→ string

Aufgabe 2

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0x0000	89	50	4e	47	0d	0a	1a	0a	ff	00	00	00	49	48	44	52
0x0010	00	00	05	00	00	00	02	82	08	06	00	00	00	8e	3b	74
0x0020	aa	00	00	00	a7	73	42	49	54	08	08	08	08	7c	08	64
0x0030	47	42	53	00	88	00	00	00	4d	00	00	0f	61	00	00	0f
0x0040	61	01	a8	3f	a7	69	00	00	00	38	74	45	58	74	53	6f
0x0050	66	74	77	61	72	65	00	6d	61	74	70	6c	6f	74	6c	69
0x0060	f1	20	f5	65	72	73	69	6f	6e	33	2e	31	2e	31	2c	20
0x0070	68	74	74	70	3a	2f	73	74	72	69	6e	67	00	74	6c	69
0x0080	62	2e	6f	72	67	2f	10	66	17	19	00	00	20	00	49	44
0x0090	41	54	78	9c	ec	dd	77	9c	54	e5	a1	ff	f1	cf	6c	85
0x00a0	5d	76	97	be	85	de	62	69	6e	67	6f	00	a4	05	93	dc
0x00b0	18	63	bb	37	a2	49	ae	29	e6	17	d4	28	22	45	16	05
0x00c0	c4	44	62	12	4d	31	d7	80	b1	5f	5b	12	51	41	20	8a
0x00d0	08	00	00	00	39	05	00	00	b2	d4	2d	2c	db	cf	ef	8f
0x00e0	49	f6	c6	58	00	dd	e5	ec	ce	7e	de	af	d7	bc	5e	3c
0x00f0	87	39	33	df	59	75	9d	f9	ce	73	9e	27	12	04	41	80

```
int *i = 0xd0;
int a = i [1]; = *(i+1)
printf ("%d",a);
```

39 05 00 00

↓ LE

0x 539 = 1337

Aufgabe 2

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0x0000	89	50	4e	47	0d	0a	1a	0a	ff	00	00	00	49	48	44	52
0x0010	00	00	05	00	00	00	02	82	08	06	00	00	00	8e	3b	74
0x0020	aa	00	00	00	a7	73	42	49	54	08	08	08	08	7c	08	64
0x0030	47	42	53	00	88	00	00	00	4d	00	00	0f	61	00	00	0f
0x0040	61	01	a8	3f	a7	69	00	00	00	38	74	45	58	74	53	6f
0x0050	66	74	77	61	72	65	00	6d	61	74	70	6c	6f	74	6c	69
0x0060	f1	20	f5	65	72	73	69	6f	6e	33	2e	31	2e	31	2c	20
0x0070	68	74	74	70	3a	2f	73	74	72	69	6e	67	00	74	6c	69
0x0080	62	2e	6f	72	67	2f	10	66	17	19	00	00	20	00	49	44
0x0090	41	54	78	9c	ec	dd	77	9c	54	e5	a1	ff	f1	cf	6c	85
0x00a0	5d	76	97	be	85	de	62	69	6e	67	6f	00	a4	05	93	dc
0x00b0	18	63	bb	37	a2	49	ae	29	e6	17	d4	28	22	45	16	05
0x00c0	c4	44	62	12	4d	31	d7	80	b1	5f	5b	12	51	41	20	8a
0x00d0	08	00	00	00	39	05	00	00	b2	d4	2d	2c	db	cf	ef	8f
0x00e0	49	f6	c6	58	00	dd	e5	ec	ce	7e	de	af	d7	bc	5e	3c
0x00f0	87	39	33	df	59	75	9d	f9	ce	73	9e	27	12	04	41	80

```
int *i = 0xd0;
int b = * (int*) *i;
printf ("%d",b);
```

$i \rightarrow d0 = 0d\ 00\ 00\ 00$

← LE

0x08

↓ (int*)

ff 00 00 00

↓ LE

0x ff = 255

Aufgabe 3

Sichere Programmierung

a)

```
1 char userInput[256];  
2 gets(userInput);
```

b)

```
1 char userInput[256] = {0};  
2 int ret = scanf("%256s", userInput);
```

Aufgabe 3

Sichere Programmierung

a)

```
1 char userInput[256];  
2 gets(userInput);
```

fgets (userinput, 256, stdin)

↳ 255 + Nullbyte

Die Funktion gets setzt kein Limit an den einzulesenden String und führt somit zu Buffer-Overflow!

b)

```
1 char userInput[256] = {0};  
2 int ret = scanf("%256s", userInput);
```

Aufgabe 3

Sichere Programmierung

a)

```
1 char userInput[256];  
2 gets(userInput);
```

Die Funktion gets setzt kein Limit an den einzulesenden String und führt somit zu Buffer-Overflow!

b)

```
1 char userInput[256] = {0};  
2 int ret = scanf("%256s", userInput);
```

Scanf liest hier zwar nur so viele Zeichen ein wie in den Buffer passen ein (256), aber ist der Input 256 Zeichen lang, so wird der NULL-Terminator am Ende hinter das Array geschrieben!

Aufgabe 3

Sichere Programmierung

c)

```
1  #define MUL(x,y) x*y
2  int y = MUL(4+1, 3+6);
```

d)

```
1  int *p1, p2;
```

Aufgabe 3

Sichere Programmierung

c)

```
1  #define MUL(x,y) x*y
2  int y = MUL(4+1, 3+6);
```

Die Makros werden als erster Schritt vom Präprozessor aufgelöst. Dabei findet jedoch eine reine textuelle Ersetzung statt (und keine “logische” Ersetzung mit Klammerung wie bei Ersetzung von Variablen)! Also wird $4 + 1 * 3 + 6 = 13$ berechnet, anstelle von $(4 + 1) * (3 + 6) = 45$! Eine Sinngemäße Definition des Makros wäre: `#define MUL (x , y ) ((x) * (y))`

d)

```
1  int (*p1), (*p2) = int* p1, p2;
```

Handwritten note: int^ above $*p1$ and $*p2$, and int^* above $=$*

*Handwritten note: $\text{int} *p1, *p2;$*

Aufgabe 3

Sichere Programmierung

c)

```
1  #define MUL(x,y) x*y
2  int y = MUL(4+1, 3+6);
```

Die Makros werden als erster Schritt vom Präprozessor aufgelöst. Dabei findet jedoch eine reine textuelle Ersetzung statt (und keine “logische” Ersetzung mit Klammerung wie bei Ersetzung von Variablen)! Also wird $4 + 1 * 3 + 6 = 13$ berechnet, anstelle von $(4 + 1) * (3 + 6) = 45$! Eine Sinngemäße Definition des Makros wäre: `#define MUL (x , y ) ((x) * (y))`

d)

```
1  int *p1, p2;
```

Nur p1 ist ein int-pointer! Der Typ von p2 ist int! Der Asterisk ist der Variable zugeordnet, nicht dem Typen.

Aufgabe 3

Sichere Programmierung

e)

```
1  int *p = malloc(sizeof *p);
2  scanf("%d", p);
3  free(p);
4  printf("*p_is_%d", *p);
```

f)

```
1  int* list;
2  if (list == NULL) {
3      list = malloc(LIST_SIZE);
4  }
```

Aufgabe 3

Sichere Programmierung

e)

```
1  int *p = malloc(sizeof *p);
2  scanf("%d", p);
3  free(p);
4  printf("*p_is_%d", *p);
```

Use-After-Free!

f)

```
1  int* list;
2  if (list == NULL) {
3      list = malloc(LIST_SIZE);
4  }
```

Aufgabe 3

Sichere Programmierung

e)

```
1  int *p = malloc(sizeof *p);
2  scanf("%d", p);
3  free(p);
4  printf("*p_is_%d", *p);
```

Use-After-Free!

f)

```
1  int* list;
2  if (list == NULL) {
3      list = malloc(LIST_SIZE);
4  }
```

**int* list ist uninitialisiert! Der Wert von list könnte somit irgendwas sein.
Stattdessen: int* list = NULL;**