Inhalt

Code-Snippets

Durchschnitt

```
// (double), ansonsten kommt eine ganze Zahl raus
double durchschnitt = (double)summe / anzahl;
```

Quersumme (Summe aller Ziffern)

```
int berechneQuersumme(int n) {
    int summe = 0;
    while (n > 0) {
        summe += n % 10;
        n /= 10;
    }
    return summe;
}
```

Qsort Beispiel

```
/* qsort example */
#include <stdio.h>      /* printf */
#include <stdlib.h>     /* qsort */

int values[] = { 40, 10, 100, 90, 20, 25 };

int compare (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}

int main ()
{
  int n;
  qsort (values, 6, sizeof(int), compare);
  for (n=0; n<6; n++)
     printf ("%d ",values[n]);
  return 0;
}
```

Typ vergleichen (Qsort Docs)

```
int compareMyType (const void * a, const void * b)
{
  if ( *(MyType*)a <  *(MyType*)b ) return -1;
  if ( *(MyType*)a == *(MyType*)b ) return 0;
  if ( *(MyType*)a >  *(MyType*)b ) return 1;
}
```

Fibonacci (Rekursiv & Iterativ)

```
unsigned long fibonacci(unsigned int n) {
    unsigned long a = 0, b = 1;
    while (n--) {
        unsigned long temp = a;
        a = b;
        b += temp;
    }
    return a;
}

int fib_rek(int n) {
    if (n < 3) {
        return 1;
    }

    return fib_rek(n - 1) + fib_rek(n - 2);
}
```

Binär zu Dezimal

```
unsigned toDecimal(unsigned binary) {
    int decimal = 0;
    int count = 0;
    while (binary) {
        int num = binary % 10; //auslesen der Zahl von rechts nach links
        if (num == 1) {
            num <<= count; //Bit-Schiebeoperator nach links (num * 2^count)
        }
        decimal += num;
        binary /= 10;
        count++;
    }
    return decimal;
}
```

Dezimal zu Binär

```
unsigned toBinary(unsigned decimal) {
    unsigned binary = 0;
    int place = 1; // Aktuelle Position in Binär (1, 10, 100, ...)

    while (decimal != 0) {
        int remain = decimal % 2;
        binary += remain * place;
        decimal /= 2;
        place *= 10;
    }

    return binary;
}
```

ASCII Kalender

```
const char *WEEK[] = {"Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"};

void calendar(unsigned nr_days, unsigned weekday_1st) {
    for (int i = 0; i < 7; i++) {
        printf("%s  ", WEEK[i]);
    }

    printf("\n");

    int day = 0;

    while (1) {
        for (int i = 0; i < 7; i++) {
            if (day >= nr_days) {
                printf("\n\n");
                return;
            }

            if (day < weekday_1st - 1) {
                printf("    ");
            } else {
                printf("%2d  ", day + 1);
            }
            day++;
        }
        printf("\n");
    }
}
```

Zweierkomplement Binär <-> Dezimal

```c
#define BIT_WIDTH 8

int binaryToDecimal(const char *binary) {
    int len = strlen(binary);  // <string.h>
    int isNegative = (binary[0] == '1');
    int result = 0;

    for (int i = 0; i < len; i++) {
        result = (result << 1) | (binary[i] - '0');
    }

    if (isNegative) {               // Zweierkomplement
        int mask = (1 << len) - 1;  // Mask for bit-width
        result = -(mask + 1 - result);
    }

    return result;
}

void decimalToBinary(int num, int bits, char *binary) {
    unsigned int mask = 1 << (bits - 1);

    for (int i = 0; i < bits; i++) {
        binary[i] = (num & mask) ? '1' : '0';
        mask >>= 1;
    }
    binary[bits] = '\0';  // Null-terminate
}
char binaryStr[BIT_WIDTH + 1];

const char *binaryInput = "11111010";  // -6
int decimalValue = binaryToDecimal(binaryInput);

int decimalInput = -6;
decimalToBinary(decimalInput, BIT_WIDTH, binaryStr);

return 0;
```

String Reverse

```c
void reverse_string(char *str) {
    if (str == NULL) return;  // Handle null pointers

    int i = 0;
    int j = strlen(str) - 1;
    while (i < j) {
        // Swap the characters at positions i and j
        char temp = str[i];
        str[i] = str[j];
        str[j] = temp;
        i++;
        j--;
    }
}

// char myString[] = "Hello, world!";
// reverse_string(myString);
```

String Trim (Leerzeichen)

```c
void trim(char *s) {
    char *start = s;
    char *end;

    // Entferne führende Leerzeichen
    while (*start != '\0' && isspace((unsigned char)*start)) {
        start++;
    }

    // Falls nötig: den getrimmten Teil an den Anfang kopieren
    if (start != s) {
        memmove(s, start, strlen(start) + 1);  // +1 für den Null-Terminator
    }

    // Entferne nachfolgende Leerzeichen
    end = s + strlen(s) - 1;
    while (end >= s && isspace((unsigned char)*end)) {
        *end = '\0';  // Überschreibe das Leerzeichen mit dem Null-Zeichen
        end--;
    }
}

// char text[] = "   Hallo, Welt!   ";
// trim(text);
```

Datentypen

Boolean

Alles, was nicht „0“ ist, ist wahr.

Array

```
// man kann die Länge (4) auch auslassen
int liste[4] = {0, 1, 2, 3};
liste[1] = 9; // ändere das 2. Element
```

Länge eines Arrays

```
(int)( sizeof(array) / sizeof(array[0])
```

Größte Zahl

Beachte: i < 5, weil 5 die Länge des Arrays entspricht!

```
int arr[] = {0, 1, 2, 3, 4};
int max = 0;


for (int i = 0; i < 5; i++) {
    if (arr[i] > max) {
        max = arr[i];
    }
}
printf("max=%d", max);
```

Kleinste Zahl

...Gleiches wie oben, nur mit den folgenden Änderungen:

```
#include <limits.h>
// die größtmögliche Zahl
int min = INT_MAX;
if (arr[i] < min) {
```

Ist Array sortiert?

```
// Prüfung von Werten im Array, ob nach Reihenfolge sortiert
int isSorted(int a[], unsigned n) {
    if (n < 2) {
        return 1;
    }

    for (unsigned i = 0; i < n - 1; i++) {
        if (a[i + 1] < a[i]) {
            return 0;
        }
    }
    return 1;
}
```

Strings

Warnung: Für beide: `#include <string.h>`

```
// Ob 2 Strings gleich sind
strcmp(a, b) == 0
```

```
char a[] = "Hello,";
strcat(a, " world!");
// a: Hello, world!
```

```
char c = 'a';
printf("%d", c);  // 97
```

```
char c = "a";
printf("%d", c);
// bei "...": immer 0
```

```
char c = 97;
printf("%d", c);  // 97
```

Find the length of a string excluding '\0' NULL character.	**strlen**(str);
Copies a string from the source to the destination.	**strcpy**(dest, src);
Copies n characters from source to the destination.	**strncpy**(dest, src, n);
Concatenate one string to the end of another.	**strcat**(dest, src);
Concatenate n characters from the string pointed to by src to the end of the string pointed to by dest.	**strncat**(dest, src, n);
Compares these two strings lexicographically.	**strcmp**(s1, s2);
Compares first n characters from the two strings lexicographically.	**strncmp**(s1, s2, n);
Find the first occurrence of a character in a string.	**strchr**(s, c);
Find the last occurrence of a character in a string.	**strchr**(*s*, *ch*);
First occurrence of a substring in another string.	**strstr**(s, subS);
Format a string and store it in a string buffer.	**sprintf**(s, format, ...);
Split a string into tokens based on specified delimiters.	**strtok**(s, delim);

Schleifen

a: Anfang (erste Zahl, welche in der Schleife „verarbeitet" wird).

b: Zahl, welche in der Bedingung verwendet wird.

Beachte! Bei (do) while gehen wir davon aus, dass `i++;` nach der Verarbeitung von i aufgerufen wird!

Bedingung	Inkrement-operator	Durchläufe	Letzte Zahl *ǧǫ̣s-*	Letzte Zahl *-đǫ̣--xḥîl̓ê*
i != b	++	a	b-1	b
i != b	--	a	b+1	b
i >= b	--	a+1	b	b-1
i > b	--	a	b+1	b
i < b	++	b	b-1	b
i <= b	++	b+1	b	b+1

Rechenoperationen

```
int n1 = 5;                 double r1 = 5.0;
int n2 = 3;                 double r2 = 3.0;
int x;                      double x;
x = -n1;      // x: -5      x = -r1; // x: -5.0
x = n1 + n2; // x: 8        x = r1 + r2; // x: 8.0
x = n1 - n2; // x: 2        x = r1 - r2; // x: 2.0
x = n1 * n2; // x: 15       x = r1 * r2; // x: 15.0
x = n1 / n2; // x: 1        x = r1 / r2; // x: 1.66666666
x = n1 % n2; // x: 2
```

Zahlen umwandeln

Warnung: #include <stdlib.h>

String zu Zahl

```
atoi("123")
```

Konsole

Print Formatierung

specifier	Output	Example
d ***or*** i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
%	A % followed by another % character will write a single % to the stream.	%

Nachkommastellen einer Float

```
// zwei Nachkommastellen
printf("%.2f")
```

Scanf (Eingabe)

```
// Anfang -> a (lf=double)
// letzten 3 Buchstaben -> b (s=char)
scanf("%lf %3s", &a, b);
```

Konsolen-Argumente

Kommandozeilen-Argumente bedeutet folgendes: wenn ich im Terminal das Programm mit dem Befehl `main.c a b` starte, dann sind „a“ und „b“ Argumente.

Wichtig: Bei `argc` **1 abziehen**, um die Anzahl der Argumente zu erhalten!

```
// argc: Anzahl der Argumente + 1
// argv: Kommandozeilen-Argumente
int main(int argc, char *argv[]) {
}
```

Man kann folgendermaßen jedes einzelne Kommandozeilen-Argument „verarbeiten“:

Warnung: Das erste `argc` Argument ist die Datei, die ausgeführt wird. Sie wird in der Regel ignoriert.

```
for (int i = 1; i < argc; i++) {
}
```

Return-Wert

`return 0;` heißt, dass das Programm richtig ausgeführt hat.

`return 1;` gibt einen Fehler zurück

Dateien

Dateien auslesen

```c
FILE *inFile = fopen("file.txt", "r");
char text[4096] = "";

if (inFile == NULL) {
    printf("Error opening file\n");
    return 1;
}

char buffer[256];
while (fgets(buffer, sizeof(buffer), inFile)) {
    strcat(text, buffer);
}

fclose(inFile);
```

sscanf()

```c
char buffer[256];
double temp, sum = 0;
int file_room, count = 0;

while (fgets(buffer,  MaxCount: sizeof(buffer), fptr)) {
    if (sscanf( source: buffer,  format: "%d. %lf\n", &file_room, &temp) == 2) {
        if (file_room == room) {
            sum += temp;
            count++;
        }
    }
}
```

Dateien schreiben

```c
FILE *fptr = fopen("out.txt", "w");
char text[4096] = "Hello, world!";
fprintf(fptr, text);
fclose(fptr);
```

Dateien anhängen

```c
int log_temp(double value, int room) {
    FILE *fptr = fopen( Filename: "log_temp.txt",  Mode: "a");
    fprintf(fptr,  format: "%d. %0.2lf\n", room, value);
    fclose(fptr);
}
```

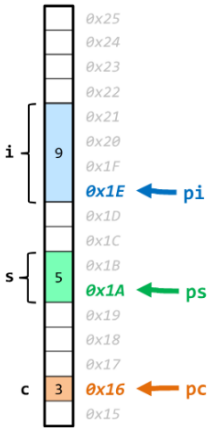
Pointer

Pointer speichern die Adresse einer Variablen. Sie „zeigen“ also auf eine Variable.

`*` = **dereference** (Wert von…) `&` = **Addresse** von…

Pointers Store **One** Address Only

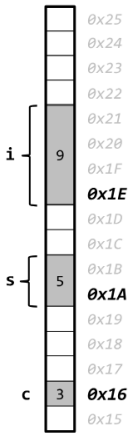
```c
char  c = 3;
short s = 5;
int   i = 9;

char*  pc = &c;
short* ps = &s;
int*   pi = &i;
```




```c
int n = 5;

int *p = &n;

printf("Value: %d", *p); // 5
```

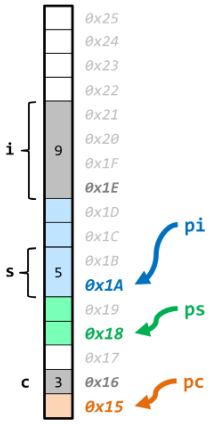
Remember: Types Have Different Memory Sizes

```c
char  c = 3;
short s = 5;
int   i = 9;
```



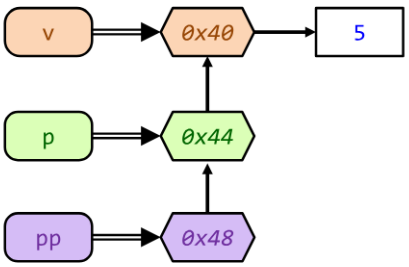
Pointer Arithmetic: Decrement by 1

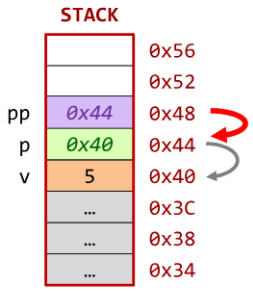
```c
char  c = 3;
short s = 5;
int   i = 9;

char*  pc = &c;
short* ps = &s;
int*   pi = &i;

pc--;
ps--;
pi--;
```



Pointers to Pointers

```
int   v = 5;   // value of v is 5
int*  p = &v;  // take address of v
int** pp = &p; // take address of p
```



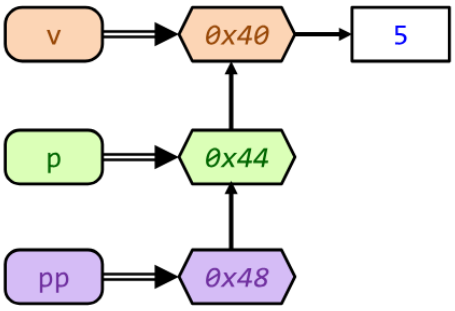


Pointers to Pointers

& one step back

***** one step forward

```
int   v = 5;
int*  p = &v;
int** pp = &p;

cout << v;      // 5
cout << p;      // 0x40
cout << pp;     // 0x44

cout << &v;     // 0x40 (= p)
cout << &p;     // 0x44 (= pp)
cout << &pp;    // 0x48

cout << *p;     // 5
cout << *pp;    // 0x40 (= p)
cout << **pp;   // 5
```



Pass by Reference

```
void change(int *var) { *var = 5; }

// main()

int var;

change(&var);
```

Matrizen

```
double mtrx[2][3] = {{0, 1, 2}, {3, 4, 5}};
```

Inverse Matrix

Eine Matrix, die, wenn sie mit der ursprünglichen Matrix multipliziert wird, die Einheitsmatrix ergibt.

$$A \cdot A^{-1} = E$$

$$\begin{pmatrix} 1 & 2 & -1 \\ 0 & 1 & -1 \\ 2 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Einheitsmatrix

Eine quadratische Matrix, bei der alle Hauptdiagonalelemente 1 und alle anderen Elemente 0 sind.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Transponierte Matrix

Eine Matrix, die entsteht, wenn man die Zeilen und Spalten einer Matrix vertauscht.

$$A_{3,3} = \begin{pmatrix} 5 & 7 & 3 \\ 8 & 16 & 9 \\ 22 & 1 & 7 \end{pmatrix} \quad A^T_{3,3} = \begin{pmatrix} 5 & 8 & 22 \\ 7 & 16 & 1 \\ 3 & 9 & 7 \end{pmatrix}$$

3 x 3 Matrix 3 x 3 Matrix

Stack Overflow

1. **char** (1 Byte)
    - **Bereich**:
        - signed char: -128 bis 127
        - unsigned char: 0 bis 255
    - **Überlauf**: Bei 128 (wird -128).
2. **short** (2 Bytes)
    - **Bereich**:
        - signed short: -32.768 bis 32.767
        - unsigned short: 0 bis 65.535
    - **Überlauf**: Bei 32.768 (wird -32.768).
3. **int** (4 Bytes)
    - **Bereich**:
        - signed int: -2.147.483.648 bis 2.147.483.647
        - unsigned int: 0 bis 4.294.967.295
    - **Überlauf**: Bei 2.147.483.648 (wird -2.147.483.648).
4. **long** (4 oder 8 Bytes)
    - **Bereich**:
        - signed long: -2.147.483.648 bis 2.147.483.647 (4 Bytes) oder -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807 (8 Bytes)
        - unsigned long: 0 bis 4.294.967.295 (4 Bytes) oder 0 bis 18.446.744.073.709.551.615 (8 Bytes)
    - **Überlauf**: Entsprechend der Größe.
5. **long long** (8 Bytes)
    - **Bereich**:
        - signed long long: -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
        - unsigned long long: 0 bis 18.446.744.073.709.551.615
    - **Überlauf**: Bei 9.223.372.036.854.775.808 (wird -9.223.372.036.854.775.808).

Bitwise Operatoren (Bitwise XOR, ...)

Für jedes Bit wird jeweils der Operator angewendet:

Symbol	Operator
`&`	bitwise AND
`|`	bitwise inclusive OR
`^`	bitwise XOR (exclusive OR)
`<<`	left shift
`>>`	right shift
`~`	bitwise NOT (ones' complement) (unary)

ASCII Tabelle

Dec	Oct	Hex	C	Dec	Oct	Hex	C	Dec	Oct	Hex	C	Dec	Oct	Hex	C
0	0	0	^@	32	40	20		64	100	40	@	96	140	60	`
1	1	1	^A	33	41	21	!	65	101	41	A	97	141	61	a
2	2	2	^B	34	42	22	"	66	102	42	B	98	142	62	b
3	3	3	^C	35	43	23	#	67	103	43	C	99	143	63	c
4	4	4	^D	36	44	24	$	68	104	44	D	100	144	64	d
5	5	5	^E	37	45	25	%	69	105	45	E	101	145	65	e
6	6	6	^F	38	46	26	&	70	106	46	F	102	146	66	f
7	7	7	^G	39	47	27	'	71	107	47	G	103	147	67	g
8	10	8	^H	40	50	28	(	72	110	48	H	104	150	68	h
9	11	9	^I	41	51	29	)	73	111	49	I	105	151	69	i
10	12	a	^J	42	52	2a	*	74	112	4a	J	106	152	6a	j
11	13	b	^K	43	53	2b	+	75	113	4b	K	107	153	6b	k
12	14	c	^L	44	54	2c	,	76	114	4c	L	108	154	6c	l
13	15	d	^M	45	55	2d	-	77	115	4d	M	109	155	6d	m
14	16	e	^N	46	56	2e	.	78	116	4e	N	110	156	6e	n
15	17	f	^O	47	57	2f	/	79	117	4f	O	111	157	6f	o
16	20	10	^P	48	60	30	0	80	120	50	P	112	160	70	p
17	21	11	^Q	49	61	31	1	81	121	51	Q	113	161	71	q
18	22	12	^R	50	62	32	2	82	122	52	R	114	162	72	r
19	23	13	^S	51	63	33	3	83	123	53	S	115	163	73	s
20	24	14	^T	52	64	34	4	84	124	54	T	116	164	74	t
21	25	15	^U	53	65	35	5	85	125	55	U	117	165	75	u
22	26	16	^V	54	66	36	6	86	126	56	V	118	166	76	v
23	27	17	^W	55	67	37	7	87	127	57	W	119	167	77	w
24	30	18	^X	56	70	38	8	88	130	58	X	120	170	78	x
25	31	19	^Y	57	71	39	9	89	131	59	Y	121	171	79	y
26	32	1a	^Z	58	72	3a	:	90	132	5a	Z	122	172	7a	z
27	33	1b	^[	59	73	3b	;	91	133	5b	[	123	173	7b	{
28	34	1c	^\	60	74	3c	<	92	134	5c	\	124	174	7c	|
29	35	1d	^]	61	75	3d	=	93	135	5d	]	125	175	7d	}
30	36	1e	^^	62	76	3e	>	94	136	5e	^	126	176	7e	~
31	37	1f	^_	63	77	3f	?	95	137	5f	_	127	177	7f	

Häufige Fehler

Mehrere Zuweisungen

Achtung: *min* wird hier nicht korrekt zugewiesen!

```
int min, max = a[0];
```

Zuweisung falsch herum

Achtung: Der rechte Teil ist falsch herum!

```
if (a[i] < min) a[i] = min;
if (a[i] > max) a[i] = max;
```

Schleife

Achtung: Summen aus *double*s sind logischerweise auch doubles.

Wenn n die Anzahl ist, dann gilt **i < n**, ansonsten wird ein nicht existierendes Element aufgerufen!

```
double average(double a[], unsigned n) {
  int sum = 0;
  for(unsigned i=0; i<=n; i++);
    sum += a[i];
  return sum / n;
}
```

Ausgabe

Achtung: **%d** gibt einen **falschen** Wert (nicht gerundeten!) bei *double*s aus! Für's Runden: %.0f!