Prog C

HS 2023, Prof. Dr. Christian Werner

Fabian Steiner, 31. Januar 2026

V2.1.2

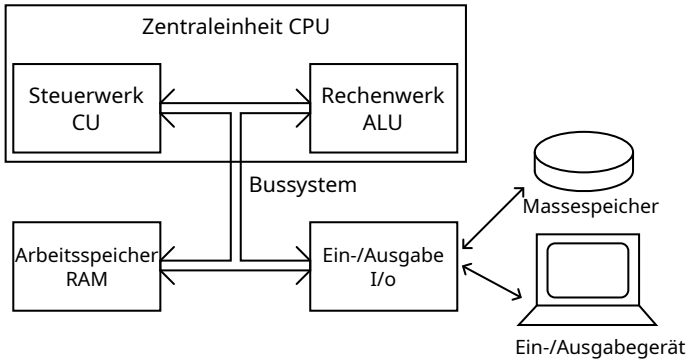
1 Grundlagen

1.1 Trivia

Designer : Dennis Ritchie
Erstveröffentlichung : 1972

1.2 Rechnersystem

Grundaufbau eines Rechnersystems:



Grösse	Genauer Wert	näherungswert
Kilobyte (KB)	2^{10} Bytes = 1024 Bytes	10^3 Bytes
Megabyte (MB)	2^{20} Bytes = 1024 KB	10^6 Bytes
Gigabyte (GB)	2^{30} Bytes = 1024 MB	10^9 Bytes

Die CPU kann ($2^{\text{Bitgrösse des Busses}} \cdot$ 8 bits) Ansteuern

1.3 Zahlensysteme

1.3.1 Binär direkt

Binär kann mittels 2er Potenz berechnet werden (geht auch bei Kommazahlen) Beispiel :

Gewichte:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Bitmuster:	1	0	1	0	0	1	1	0

$$\begin{aligned}\text{Wert} &= 1\cdot 2^7 + 0\cdot 2^6 + 1\cdot 2^5 + 0\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 \\ &= 128 + 0 + 32 + 0 + 0 + 4 + 2 + 0 \\ &= 166 \text{ (dezimal)}\end{aligned}$$

1.3.2 Binär Zweierkomplement

Das Zweierkomplement erlaubt das Darstellen von negativen Zahlen. Das MSB hat dabei den üblichen Wert negativ.
Um von Binär zum Zweierkomplement zu kommen, muss man alle Bits einer binär zahl invertieren und 1 addieren

$$\overbrace{0110}^{6} \xrightarrow{\text{invert}} \overbrace{1001}^{-7} \xrightarrow{+1} \overbrace{1010}^{-6}$$

1.3.3 ASCII

American **S**tandard **C**ode for **I**nformation **I**nterchange standardisiert die Repräsentation eines Bytes als Char. Da ASCII ein 7-bit Code ist, ist das MSB immer 0.

Dec	Hex	Bin	Char	Dec	Hex	Bin	Char	Dec	Hex	Bin	Char
0	0	0	[NULL]	43	2B	101011	+	86	56	1010110	V
1	1	1	[start of heading]	44	2C	101100	,	87	57	1010111	W
2	2	10	[start of text]	45	2D	101101	-	88	58	1011000	X
3	3	11	[end of text]	46	2E	101110	.	89	59	1011001	Y
4	4	100	[end of transmission]	47	2F	101111	/	90	5A	1011010	Z
5	5	101	[enquiry]	48	30	110000	0	91	5B	1011011	[
6	6	110	[acknowledge]	49	31	110001	1	92	5C	1011100	\
7	7	111	[bell]	50	32	110010	2	93	5D	1011101	]
8	8	1000	[backspace]	51	33	110011	3	94	5E	1011110	^
9	9	1001	[horizontal tab]	52	34	110100	4	95	5F	1011111	_
10	A	1010	[line feed]	53	35	110101	5	96	60	1100000	`
11	B	1011	[vertical tab]	54	36	110110	6	97	61	1100001	a
12	C	1100	[form feed]	55	37	110111	7	98	62	1100010	b
13	D	1101	[carriage return]	56	38	111000	8	99	63	1100011	c
14	E	1110	[shift out]	57	39	111001	9	100	64	1100100	d
15	F	1111	[shift in]	58	3A	111010	:	101	65	1100101	e
16	10	10000	[data link escape]	59	3B	111011	;	102	66	1100110	f
17	11	10001	[device control 1]	60	3C	111100	<	103	67	1100111	g
18	12	10010	[device control 2]	61	3D	111101	=	104	68	1101000	h
19	13	10011	[device control 3]	62	3E	111110	>	105	69	1101001	i
20	14	10100	[device control 4]	63	3F	111111	?	106	6A	1101010	j
21	15	10101	[neg. Acknowledge]	64	40	1000000	@	107	6B	1101011	k
22	16	10110	[synchonous idle]	65	41	1000001	A	108	6C	1101100	l
23	17	10111	[end of trans.block]	66	42	1000010	B	109	6D	1101101	m
24	18	11000	[cancel]	67	43	1000011	C	110	6E	1101110	n
25	19	11001	[end of medium]	68	44	1000100	D	111	6F	1101111	o
26	1A	11010	[substitute]	69	45	1000101	E	112	70	1110000	p
27	1B	11011	[escape]	70	46	1000110	F	113	71	1110001	q
28	1C	11100	[file separator]	71	47	1000111	G	114	72	1110010	r
29	1D	11101	[group separator]	72	48	1001000	H	115	73	1110011	s
30	1E	11110	[record separator]	73	49	1001001	I	116	74	1110100	t
31	1F	11111	[unit separator]	74	4A	1001010	J	117	75	1110101	u
32	20	100000	[space]	75	4B	1001011	K	118	76	1110110	v
33	21	100001	!	76	4C	1001100	L	119	77	1110111	w
34	22	100010	"	77	4D	1001101	M	120	78	1111000	x
35	23	100011	#	78	4E	1001110	N	121	79	1111001	y
36	24	100100	$	79	4F	1001111	O	122	7A	1111010	z
37	25	100101	%	80	50	1010000	P	123	7B	1111011	{
38	26	100110	&	81	51	1010001	Q	124	7C	1111100	|
39	27	100111	'	82	52	1010010	R	125	7D	1111101	}
40	28	101000	(	83	53	1010011	S	126	7E	1111110	~
41	29	101001	)	84	54	1010100	T	127	7F	1111111	[DEL]
42	2A	101010	*	85	55	1010101	U				

1.3.4 Oktalzahlen

Oktalzahlen erlauben nur 8 Ziffern (0 - 7). Als Bitmuster brauchen sie nur 3 Bits um 1 Ziffer darzustellen.

$$\overbrace{110}^{6}\overbrace{001}^{1} \rightarrow 61$$

Schreibweisen: 61_{o}, 61_{q}, 61_{Oct}, 061
Typischerweise mit 0 vor Beginn der Zahl.

1.3.5 Hexadezimal

Hexadezimal erlaubt 16 Ziffern (0 - 9, A - F). Binär stellen 4 Bits eine Hex Zahl dar.

$$\overbrace{1100}^{C}\overbrace{0101}^{5} \rightarrow \text{0xC5}$$

Schreibweisen: $C5_{\text{h}}$, C5H, $C5_{\text{hex}}$, C5$, 0xC5
Typischerweise mit 0x vor Beginn der Zahl.

1.3.6 Umrechnung beliebiges Zahlensystem zu dezimal

Die jeweiligen Ziffern stellen jeweils ein Zahlenwert im System:
$n^0 \cdot x_1 + n^1 \cdot x_2 + n^2 \cdot x_3 + ...$ wobei n die Zahlenbasis ist.

1.3.7 Umrechnung dezimal zu beliebigem Zahlensystem

Die umzurechnende Zahl muss durch die Zahlenbasis mit Rest dividiert werden. Der Rest ist dann die erste Ziffer der entstehenden Zahl. Der erhaltene Quotient muss erneut dividiert werden und man erhält dann die folgenden Ziffern. Wenn der Quotient 0 erreicht hat, ist die Umrechnung fertig.

2 Datentypen

Datentypen legen fest,

- wie lang das Bitmuster an der zugehörigen Speicherstelle ist,
- was dieses Bitmuster bedeutet.

2.1 simple Ganzzahltypen

char	immer 8 Bit (Als einziger Typ **IMMER** 1 Byte)
short	für kleine ganzzahlige Werte (mind. 16 Bit)
int	effizienteste Grösse für Prozessor (mind. 16 Bit)
long	für grosse ganze werte (mind. 32 Bit)
long long	für sehr grosse ganze werte (mind. 64 Bit)
size_t	mindestens so gross wie die maximale Anzahl Adressen, nie negativ

Ganzzahltypen können **signed** (Vorzeichenbehaftet) oder **unsigned** sein. Wenn nicht definiert, meist signed ausser Datentyp **char**, dieser ist nicht standardisiert.
Überlauf ist nicht definiertes Verhalten
Der **unsigned** Wertebereich befindet sich von 0 bis $2^n - 1$
Der **signed** Wertebereich (Zweierkomplement) befindet sich von -2^{n-1} bis $2^{n-1} - 1$
(n = anzahl bits)

2.2 Stdint.h

Stdint stellt vordefinierte Typen zur Verfügung welche Plattform unabhängige feste Grössen haben. z.B. :

- int8_t = signed 8 Bit
- uint64_t = unsigned 64 Bit
- .....

Allerdings sind diese auf dem Zielsystem möglicherweise nicht sehr effizient, da sie die native Busbreite über oder unterschreiten. Eine variable mit int Datentyp typischerweise der effizienteste.

2.3 Gleitpunktzahl

Gleitpunktzahlen speichern Zahlen wissenschaftlicher Schreibweise. Dabei treten allerdings fast immer Rundungsfehler auf. Daher **nie auf Gleichheit testen**, sondern auf einen kleinen Bereich. Sie brauchen generell viel brauchen viel Rechenleistung aber sie können auch Werte wie ± inf, ± 0, NaN (not a number) darstellen.
Speicherzusammensetzung, Standardisierter float (IEEE 754):

(in bit)	Sign	Exponent	Mantisse	total
float	1	8	23	32 bits
double	1	11	52	64 bits

Wenn eine Zahl mit einem Koma (z.B: 12.91) definiert wird, ist sie automatisch eine Gleitpunktzahl. Dieser Effekt kann verwendet werden, um bei eine Mathematischen Berechnung die Genauigkeit zu verbessern, wenn mit Ganzzahltypen gerechnet wird. Wenn eine Zahl implizit als Gleitpunktzahl (Möglichst am Anfang der Rechnung) definiert wird, wird implizit mit einer Gleitpunktzahl weiter gerechnet und am Schluss wieder zu einer Ganzzahl umgewandelt.

2.4 Enum als Aufzählungstyp

Mit einem Enum (im code immer klein schreiben) kann ein Aufzählungstyp erreicht werden für z.B. für state machines. Generelle Syntax:
`enum type-name {item1 = value1, item2 = value2, ...};`
Die Valuedeklaration kann ausgelassen werden. Der Compiler vergibt von 0 aufsteigend automatisch Werte. Es kann strategisch eine Zahl dazwischen gesetzt werden, um gewisse numerische Zahlen zu Erhalten (sehr spezifisch).
Wenn man ein Enum verwenden will, muss das Schlüsselwort "Enum" immer mitgeschrieben werden. Mit einer Typendeklaration kann das vereinfacht werden.

```
enum groesse {LOW,MEDIUM,HIGH}; //deklaration
enum groesse myVar;             //Variable mit Typ enum
enum groesse myFunc();          //Funktion mit rueckgabetyp Enum
```

2.5 Enum als Ganzzahkonstanten

Man kann per Enum auch Ganzzahlkonstanten definieren. Sie sind sicherer als ein `#define` und erlauben auch das Definieren von grössen eines Arrays:

```
enum{goodYear = 1291, badYear = 1848};
int arr[goodYear][badYear];//erlaubt, erstellt ein 2d Array
```

2.6 Pointer

Pointer sind Variablen welche eine Speicheradresse enthalten. Sie sind mindestens so gross, dass alle Speicheradressen definiert werden können.
Wenn Pointer Referenziert (**Operator** : `&`) werden, liest man die Adresse, auf die der Pointer zeigt.
Wenn Pointer dereferenziert (**Operator** : `*`) werden, liest man den Wert, auf der gezeigt wird.
Pointer müssen auch den Variabeltyp, auf den sie Zeigen, definiert bekommen um das Bitmuster auf das sie zeigen Interpretieren zu können. Ausnahme davon sind **Voidpointer**. Ihnen kann jeder Pointertyp übergeben werden. Diese dürfen aber **nicht** dereferenziert werden.
Wenn ein Pointer nicht gebraucht wird, muss er "**NULL**" (nicht zwingend numerisch 0) gesetzt werden. Das Dereferenzieren eines NULLpointer führt zu undefined behaviour. Daher müssen unbekannte Pointer **immer** auf NULL geprüft werden vor Verwendung.

```
int* aPointer;//deklaration eines Pointers mit typ int
int* ptr = NULL;//NULL initialisierter Pointer
void* ptrb;//Deklaration eines Voidpointer
aPointer = &myInt;//aPointer zeigt nun auf die adresse von myInt
otherInt = *aPointer;//other uebernimmt den Wert von myInt
ptr = apointer;//Einem Voidpointer wird ein Pointer des Typs int*
     zugewisen
```

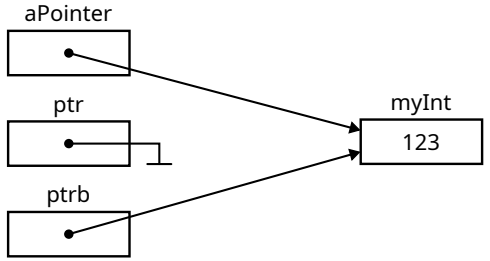
2.6.1 Pointer Arithmetik

- Pointer unterschiedlicher Datentypen dürfen einander nicht zugewiesen werden (Schutzmechanismus)
- Einem Pointer eines bestimmten Typs dürfen Pointer dieses Typs oder void-Pointer zugewiesen werden
- Einem void-Pointer dürfen beliebige Pointer zugewiesen werden (nützlich, aber gefährlich)

Auf Pointer darf des weiteren addiert und subtrahiert werden. Ganze zahlen verschieben ihn dabei jeweils um ganze Elemente. Daher kann man nicht zwischen 2 Elemente Zeigen. Darum ist der Pointer Typ wichtig. Es ist keine Arithmetik auf einen Voidpointer möglich.
Pointer darf man auch mit anderen Pointer vergleichen (==, !=,...).

2.6.2 Memorymap

Eine Memorymap wird verwendet, um den Zustand von Pointer und Variablen zu signalisieren. Das folgende Beispiel zeigt den Zustand von der Deklaration von Abschnitt 2.6.



2.7 Deklarieren

Generelle Syntax : `<Datentyp> <Variablenname>;`
Erlaubte Zeichen sind Buchstaben, Ziffern und Underscore(_). Eine Zahl darf nicht zu beginn stehen. Umlaute sind nicht erlaubt. Variablennamen sind **case sensitive!**
Beispiele:

```
int derName;                 // korrekt
int 0derName;                // falsch wegen der Zahl am Anfang
char derName1;               // erlaubt da Zahl am Ende
double binDefiniert = 9.9;   // mit Initialisierung
char eins, zwei, drei = 'a'; // mehrere Variablen werden
     initialisiert aber nur "drei" wird mit 'a' initialisiert
```

Es gibt Namen, die nicht erlaubt sind, da sie vom C Standard reserviert sind. Diese sind:

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	inline	int
long	register	restrict	return	short	signed
sizeof	static	struct	switch	typdef	union
unsigned	void	volatile	while	_alignas	_alignof
_Atomic	_Bool	_Complex	_Generic	_Imaginary	_Noreturn
_Static_assert	_Thread_local				

2.8 Umwandeln

Es gibt die implizite und explizite Umwandlung. Implizit erledigt der Compiler da er selbst ein missmatch von Typen erkennt → schlecht da es eine Warnung erzeugt.
Explizit wird Umwandeln durch Schreiben des Zieltyps in Klammern neben der umzuwandelnden Zahl erreicht.

```
//implizit: eine Kommazahl ohne f am Ende hat Typ double
float flZahl = 41.7;
//explizit: x hat den Wert 41, Nachkommastellen werden
     abgeschnitten!
int x = (int)flZahl;
```

2.9 Scope

Der Scope beschreibt die Sichtbarkeit von Variabeln. Unterschieden wird:

- **global**
  nicht in einer Funktion deklariert, 0 initialisiert, Laufzeit des Programms, immer sichtbar
- **lokal**
  in Funktion deklariert, nicht initialisiert, Laufzeit der Funktion, nur in der Funktion Sichtbar, auch wenn eine neue Funktion aufgerufen wird
- **static**
  Ausnahme → kommt auf Anwendung an. (siehe Static→ 2.12.3)

2.10 Arrays

Ein Array bietet eine kompakte Zusammenfassung von mehreren Variablen des gleichen Basistyps. Array Nummerierung **beginnen immer bei 0**. Sie sind so lang wie bei der Deklarierung definiert.

```
float messDat[12];//Ein Array fom typ float mit 12(0-11) Elementen
      wird deklariert
char text[groesse];//Ein Array fom typ char wird mit der
     konstanten "groesse" initialisiert
messDat[20] = 12; //fuehrt zu undefined behaviour. Es wird
     ausserhalb des Arrays '12' geschrieben. Es kann irgendeine
     Speicherstelle ueberschriben werden.
```

2.10.1 Initialisierung

Man kann ein Array entweder komplett oder gar nicht initialisieren.

```
int a[4] = {1,2,9,1};    //vollstaendige initialisierung
int b[4] = {6,7};        // b[2],b[3] werden mit 0 initialisiert
int c[4] = {};           //alle Elemente werden mit 0 initialisiert
int d[] = {1,2,9,1};     //die groesse wird vom Compiler festgelegt
```

2.10.2 Char Arrays

Mit Char Arrays werden Zeichenketten realisiert. Char Arrays müssen immer NULL('\0') terminiert werden, wenn mit diesen Standardfunktionen zur Char Array Verarbeitung verwendet werden. Die Nullterminierung wird verwendet, um zu wissen, wann die Zeichenkette fertig ist.

```
char name[15] = {77, 101, 106, 101, 114, 0};//numerische werte
     entsprechen der Ascii definition
char name[15] = {'M', 'e', 'i', 'e', 'r', '\0'};//einzelne chars
char name[15] = "Meier"; // bevorzugte Variante, enthaelt
     automatisch das Nullzeichen!
```

2.10.3 Memorymap

Das Format der Memorymap für Arrays sieht folgendermassen aus:

Wert 0	Wert 1	Wert 2	Wert 3	Wert 4

Unter den jeweiligen Werten muss die Element Nummer Stehen.

2.10.4 Handhabung von Arrays

Arrays können nicht direkt miteinander verglichen werden, in Funktionen direkt übergeben werden oder direkt auf ein anderes überspielt werden (**memcpy** implementiert solch eine Funktion).
Bei einem Funktionsaufruf wird ein Pointer auf das erste Element des Arrays übergeben.

2.10.5 Mehrdimensionale Arrays

Mehrdimensionale Arrays können so verstanden werden das ein Array Element wieder ein ganzes Array enthaltet.
Bei einem 2D Array kann man sich eine Matrix vorstellen, wobei beim Aufruf [Zeile][Spalte] mitgegeben wird.
Definition eines mehrdimensionalen Arrays:

```
int alpha[3][4] = {
{1, 2, 3, 4},
{5, 6, 7, 8},
{9, 0, 1, 2}};
```

1	2	3	4
5	6	7	8
9	0	1	2

Ebenso könnte man alpha so initialisieren:

```
int alpha[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2};
```

2.11 Pointer und Arrays

Pointer und Arrays können ähnlich verwendet werden. In aufgerufenen Funktionen kann mit einem Pointer so wie mit einem Array weitergearbeitet werden.

```
int* iptr;
int iarr[4] = {1,8,4,8};
iptr = iarr;// ein Array wird implizit zu einem Pointer des Basis-
     Typs gewandelt
iptr = &iarr[0];//macht dasselbe
iptr = &iarr;// Achtung: macht nicht dasselbe! Fuehrt zu einem
     Compiler-Fehler!

iarr[3] = 12;
iptr[3] = 12;
*(iptr+3) = 12;//macht 3-mal dasselbe
```

2.11.1 Char Array

Eine String Definition kann auch so erfolgen:

```
char str[] = "Ich gehoere ins printf";
char* str = "Ich gehoere ins printf";
printf("%s\n", str);
```

2.12 Typ-Attribute

Typ-Attribute definieren spezielles Variablen verhalten.

2.12.1 const

Das Attribut const (vor dem Variablentyp geschrieben) markiert eine Variable als read only für den Quellcode, die Hardware kann aber dennoch den Wert verändern.
Ein Const Array besteht nur aus konstanten.
Für einen konstanten Pointer muss const nach dem * stehen. Sonst ist nur der Wert, auf den der Pointer zeigt read only.

```
const char* text;//pointer auf einen konstanten text
const char* const text;//konstanter pointer auf einen konstanten
     text
"text"//string literal -> konstanter text
```

Um sicherzugehen, was wirklich konstant ist, kann die Definition von rechts nach links gelesen werden. Was links neben einem const steht, ist wirklich const.
In Funktionsköpfen lohnt es sich const, als Sicherheit einzubauen, so das nicht versehentlich eine variable geschrieben wird.

```
void foo(const int* const ptr);//der int Wert und Pointer kann
     nicht mehr versehentlich veraendert werden.
```

2.12.2 Volatile

Volatile signalisiert dem Compiler das sich die Variable auch ausserhalb der normalen Programmausführung ändern kann. Einsatz bei Hardware naher Programmierung und multithreading.

2.12.3 static

Static kann eine Variabel und eine Funktion sein, diese haben aber jeweils verschiedene Bedeutungen:

- **Globale Variabel oder Funktion**
  Sie sind nur in der Compile Unit gültig in der sie definiert sind.
- **lokale Variabel**
  haben dieselbe **Lebensdauer wie eine Globale Variabel**, Sichtbarkeit auf Funktionsblock beschränkt und mit 0 oder einer konstanten einmalig initialisiert.

2.13 Structs

Mittels Struct kann man verschiedene Datentypen zu einem Typ zusammenführen. Auch Structs kann man in einen anderen Struct verwenden.

Definition:

```
struct structname
{
    typ1 name1;
    typ2 name2;
    typ3 name3;
    typn namen;
};
```

bsp:

```
struct adresse
{
    int postleitzahl;
    char strasse[20];
    int hausnummer;
    char land[3];
};
```

2.13.1 Beispiel mit Pointer

Das Beispiel verwendet Deklarationen von oben

```
int main()
{
    struct adresse home = {8640,"teststr",42,"CH"};
    printf("%d %s %s",home.postleitzahl,home.strasse,home.land);//
     direkter zugriff
    //der "." Operator wird verwendet fuer Elementzugriff

    struct adresse* ptr = &home;
    printf("%d %s %s",(*ptr).postleitzahl,ptr->strasse,ptr->land);
     //pointer zugriff
    //der "->" operator ist zu bevorzugen
}
```

2.13.2 Grösse

Die Grösse eines Struct ist nicht immer die Summe aller Typen. Aufgrund von Hardwarelimitationen (Alignment) können padding bytes in der Struct eingesetzt werden welche einfach leer sind. Daher kann man die wirkliche Grösse nur durch sizeof herausfinden.

2.13.3 Funktionen

Funktionsaufrufparameter und Funktionsreturns können Structs sein. Allerdings ist dies nicht effizient und eine Übergabe des Pointers wäre besser da in diesem Fall auch Arrays direkt übergeben werden.

2.14 Union

Unions haben eine identische Syntax wie Structs, allerdings teilen **alle** Elemente den **selben** Speicherbereich. Man kann es verwenden, um z.B. einfacher an die einzelnen Bytes einer IPv4 Adresse zu kommen.

```
union IPv4Address
{
    uint8_t addressByte[4];
    uint32_t address;
};//Die Ipv4 Adresse kann nun entweder als ganzer Block gelesen
     werden oder die 4 bytes seperat.
```

2.15 typedef

Man kann sich auch eigene Typennamen erstellen mit typedef.
Syntax : `typedef  <Typ> <Name des neuen typ>`
stdint.h verwendet dies. Auch sehr nützlich um das Schlüsselwort Enum/Struct zu sparen.

```
enum state {...};
typedef enum state State_t;
//jetzt kann ich State_t schreiben anstatt enum state.

typedef enum{...} State_T;
//Macht dasselbe aber kuerzer
```

3 Funktionen aus C Standard-Library

Einige gängige Programmierprobleme wurden bereits durch die **C Standard-Library** gelöst.

Overview String Funktionen

Es gibt Funktionen aus der String Library welche mit 'str' beginnen. Diese erwarten eine **NULL** terminierte Zeichenkette. Ein anderer Typ Funktionen beginnt mit 'mem'. Bei diesen muss die Länge mitgegeben werden.
Ebenso gibt es für String Funktionen eine Version, die alle Zeichen bearbeitet oder nur n Zeichen bearbeitet.

3.1 scanf

Header `<stdio.h>` wird verwendet.
Scanf wird verwendet, um Eingaben des Nutzers entgegenzunehmen.
Syntax : `scanf(%<Format Spzifizierung>, <Pointer zu Speicher>);`
Es können auch mehrere eingaben aufs mal gemacht werden.

3.1.1 Formatspezifierer

Die Eingaben können verschieden interpretiert werden. Es gibt verschiedene Möglichkeiten dazu. Sie fangen immer mit einem % begonnen.

%c	ein char	**%o**	unsigned int in okt darstellung
%s	eine Zeichenkette	**%x**	unsigned int in hex darstellung
%d	Signed int in dezimaldarstellung	**%f**	float
%u	unsigned int in dez darstellung	**%lf**	double

Mit **vorangestelltem l oder ll** gibt man Datentypen länger als int an, z.B. %llu für einen unsigned long long in Dezimaldarstellung.
Mit **vorangestelltem h oder hh** gibt man Datentypen kürzer als int an, z.B. %hhx für einen unsigned char in Hexadezimaldarstellung.
Wenn Zahlen zwischen % und Formatspezifierer geschrieben werden dann kann die Länge limitiert werden

```
#include <stdio.h>
<correct type> var;
scanf("%3d",&var); //liest eine maximal 3 ziffern lange signed
     Dezimalzahl ein.
scanf("%lf %lf",&var,&var); //liest 2 double ein.
```

3.2 printf

Header `<stdio.h>` wird verwendet.
Printf wird zur Ausgabe von Text per Konsole verwendet.
General Syntax :
`printf("<Text[mit formatspecs]>", <variablen zu Wert>);`

3.2.1 Formatspezifierer

Auch hier gibt es Format Spezifizierer, Sie sind etwas anders als die von scanf. Die wichtigsten (die in der Tabelle) sind dieselben wie bei scanf. Der einzige Unterschied im jetzigen kontext ist das für double auch nur %f ohne l verwendet werden kann. Bei scanf **muss** es %lf sein.
Eine Zahl zwischen% und Formatspezifierer bewirkt das **mindestens** die Anzahl Stellen ausgegeben wird. Mit einem "." und einer Zahl kann die Präzision spezifiziert werden. Standart bei Float & double ist 6.

```
#include <stdio.h>
double Var 12.91;
printf("Test %1.1lf",var);  // Ausgabe : "Test 12.9"
printf("%5.3f",var);        // Ausgabe : "   12.910"
```

3.3 memcomp

Header `<string.h>` wird verwendet.
Vergleicht, ob 2 Speicherstellen gleich sind und gibt o zurück diese gleich sind.

```
int memcmp(const void *str1, const void *str2, size_t n)
```

Es darf nie mehr überprüft werden als das 1. Array gross ist. Das 2. Array darf grösser sein als das erste.

3.4 strncpy

Header `<string.h>` wird verwendet.

```
char* strncpy(char* dest, const char* src, size_t n)
```

Kopiert n Zeichen zur Destination. **strcpy** macht das selbe aber kopiert alle Zeichen.

3.5 strncat

Header `<string.h>` wird verwendet.

```
char* strncat(char* dest, const char* src, size_t n);
```

Hängt Zeichen n von String src an dest (mit Nullterminierung) an. **strcat** macht dasselbe aber kopiert alle Zeichen.

3.6 strncmp

Header `<string.h>` wird verwendet.

```
int strncmp(const char* s1, const char* s2, size_t n);
```

Vergleicht Länge n von s1 und s2. Returnt 0 wenn beide gleich sind. **strcpy** macht dasselbe aber vergleicht alle Zeichen.

3.7 strlen

Header `<string.h>` wird verwendet.

```
size_t strlen(const char* s);
```

Bestimmt die Länge des Strings.

3.8 Mem Funktionen

Es gibt einige Mem funktionen die nützlich sein können:

```
#include <string.h>
// Speicherbereich kopieren
void* memcpy(void* dest, const void* src, size_t n);
// Speicherbereich verschieben
void* memmove(void* dest, const void* src, size_t n);
// Speicherbereiche vergleichen
int memcmp(const void* s1, const void* s2, size_t n);
// Erstes Auftreten von Zeichen c in Speicherbereich s suchen
void* memchr(const void* s, int c, size_t n);
// Speicherbereich mit Wert belegen
void* memset(void* s, int c, size_t n);
```

4 Steuerstrukturen

4.1 basics

Die wichtigen Steuerstrukturen in einer Auflistung.

```
#include <stdio.h>//inkludiert stdio.h(brauchts hier nicht)
int main() //Programmaufruf
{
    if(<Bedingung>)
    {
        ;//Anweisung wenn wahr
    }
    else
    {
        ; //Anweisung wenn Falsch
    }

    while(<Bedingung>)//wenn Anweisung nicht wahr zu beginn wird
     Schleife nicht betreten
    {
        ;//mache so lange bis Bedingung falsch
        continue;//springe in den naechsten Schleifendurchlauf
    }

    do//mindestens 1 Durchlauf gibt es
    {
        ;//mache so lange bis Bedingung falsch
    } while(<Bedingung>);

    for(int i = 0;<Bedingung>;i++)// int I wird nur einmal
     initialisiert und i wird immer nach dem schleifendurchgang
     initialisiert
    {
        ;//mache so lange bis Bedingung falsch
    }

    switch(<variable>)//springe basierend auf der Variable
    {
        case 0://springt hier wen var = 0
            break;//ohne break wuerde switch hier weiterlaufen
        case 1 ... 4 : //range von 1-4
            break;
        default://sonst trifft nichts zu
            break;
    }

    return <wert>;//Return einer Funktion
    return 0;//Programm korrent durchlaufen
    return 1;//programm nicht korrekt durchlaufen Fehlercode 1
}
```

4.1.1 Sprunganweisungen

- **break**
  (do)while, for, switch abbrechen.
- **continue**
  bei (do)while und for in den nächsten Schleifendurchgang springen.
- **return**
  aus Funktion an aufrufende Stelle zurückspringen
- **goto**
  innerhalb einer Funktion an eine Marke (Label) springen (sind selten gerechtfertigt verwendet)

4.1.2 GoTo

Ein **goto** kann man **innerhalb einer Funktion** zu einer Marke Springen. Das kann sehr komfortabel sein, führt aber fast immer zu sehr schwer lesbaren Code. Daher möglichst nie verwenden. Ein Beispiel:

```
#include <stdio.h>

int func()
{
    printf("Do func Stuff\n");
inFunc: //Marke : hindert den normalen Programmablauf nicht
    return 0;
}

int main()
{
    printf("Anfang von main\n");
    func();
    goto hell;//Sprung zur Marke hell
    printf("Sollte nie ausgegeben werden\n");
hell: //Marken sind so indentiert
    printf("spring hierher\n");

    goto inFunc;//Funktioniert nicht! da out of scope

    return 0;
}
```

4.2 Funktionen

Funktionen sind eine Zusammenfassung von Anweisungen, welche Ein- und Ausgabewerte haben können.
Deklaration:
`<returnType> <functionName>(<paramType1> <paramName1>, <paramType2> <paramName2>, ...);`
Wenn eine Funktion aufgerufen wird, müssen immer **alle** Parameter mitgegeben werden.

4.2.1 Funktionsprototyp

Funktionen werden typischerweise weit oben im Quelltext definiert, als Funktionsprototyp. Der Funktionsprototyp legt das Interface der Funktion fest. Er **muss** verwendet werden, wenn die Funktion vor Definition verwendet werden möchte.

```
int myFunc(int input1, int input2,...); //Funktionsprototyp
int myfunc(int input1)                  //direkte definition
{
    return 0;
}
```

Der Parametername kann zwischen Prototyp und Definition unterschiedlich sein, ist aber nicht empfohlen.

4.2.2 Void

Void als Returntyp heisst das kein Wert zurückgegeben wird.
Return darf verwendet werden, aber es darf kein Wert hinterlegt werden.
Void als Parameter heisst das kein Parameter übergeben werden kann.

4.3 Pointer auf Funktionen

Pointer können auch auf Funktionen definiert werden zum z.B. zur Laufzeit dynamische Programmabläufe zu realisieren.
Syntax: `int (*name)(int,int);`

- **int** : Return Typ der funktion
- **(*name)** : name des Pointer
- **(int,int)** : aufrufparameter(hier 2*int)

bsp:

```
int compare (int a, int b) { ... }
int main()
{
    int (*ptr)(int, int);//pointer definieren
    ptr = &compare; // Zuweisen der Funktionsadresse
    ptr = compare; // tut dasselbe, d.h. man darf das & bei
     Funktionen auch weglassen
    printf("Vergleich liefert: ", (*ptr)(37, 12)); //
     Funktionspointer wird dereferenziert
}
```

4.4 Funktionen : Iteration und Rekursion

- **Rekursion:** eine Funktion ruft sich selbst auf.
- **Iteration:** Algorithmus enthält Abschnitte, die innerhalb einer Ausführung mehrfach durchlaufen werden (Schleife).
- Die rekursive Form kann eleganter sein, ist aber **praktisch immer ineffizienter** als die iterative Form.
- Das **Abbruchkriterium** ist bei beiden Formen **zentral**

4.4.1 Anwendungen

- Backtracking-Algorithmen z.B. Finden eines Weges durch ein Labyrinth (zurück aus Sackgasse und neuen Weg prüfen)
- Implementierung rekursiv definierter mathematischer Funktionen (z.B. Fakultätsfunktion)
- **Achtung**: der Speicherbedarf ist bei Rekursion schwer abzuschätzen. Rekursive Funktionen sind deshalb insbesondere bei der Embedded Programmierung zu vermeiden.
- **Achtung**: jede Instanz einer rekursiv gerufenen Funktion besitzt ihre eigenen, unabhängigen Variablenkopien. Ausnahme: static-deklarierte lokalen Variablen
- Indirekte Rekursion ist unbedingt zu vermeiden. Diese kommt z.B. zustande, wenn sich zwei oder mehrere Funktionen wechselseitig oder im Kreis aufrufen

4.4.2 Beispiel

Hier diese Folgenden Funktionen berechnen beide die Fakultät einer Zahl.

Fakultät **Iterativ**

```
uint64 fak(uint64 n)
{
    uint64 val = 1;
    for (int i = 2; i <= n; ++i)
    {
        val = val * i;
    }
    return val;
}
```

Fakultät **rekursiv**

```
uint64 fak(uint64 n)
{
    if (n > 1)
    {
        return n * fak(n-1);
    }//Erneuter Aufruf der
     Funktion fak
    else
    {
        return 1;
    }
}
```

4.5 Operatoren

Operatoren haben eine Priorität, welche aussagt in welcher Reihenfolge die Befehle ausgeführt werden. Von höchster Priorität(1) zu niedrigster(15).
Haben Operatoren dieselbe Priorität besagt die Assoziativität in welcher Reihenfolge ausgewertet wird.

Priorität	Symbol	Bedeutung	Assoziativität
1	(Postfix) ++ -- () [ ] -> . (Typ){init list}	Postfix-Ink/dekrement Funktionsaufruf Indexierung Elementzugriff compound literal (C99)	L - R
2	++ --(Präfix) + - (Vorzeichen) ! ~ & * (Typ) sizeof	Präfix-Ink/dekrement Vorzeichen logisches / bitweises NICHT Adresse Zeigerdereferenzierung Typumwandlung(cast) Speichergröße	R - L
3	* / %	Multiplikation, division Modulo	L - R
4	-	Addition, Subtraktion	L - R
5	<<	Links/rechts-Shift	L - R
6	< <= >= >	kleiner / grösser (gleich)	L - R
7	!= ==	(un)gleich	L - R
8	&	bitweises UND	L - R
9	^	bitweises exklusives ODER	L - R
10	|	bitweises ODER	L - R
11	&&	logisches UND	L - R
12	||	logisches ODER	L - R
13	?:	Bedingung /mini if	R - L
14	= *= /= %= += -= &= ^= |= <<= >>=	Zuweisung Zusammengesetzte Zuweisungen	R - L
15	,	Komma-Operator	L - R

4.5.1 Sizeof

Sizeof gibt die Grösse(in Bytes) eines Datentyp/array's zurück.

```
sizeof(char);//immer 8
sizeof(myArray)/sizeof(myArray[0]);//liefert die groesse eines
     Arrays
```

4.5.2 Mini if

Der ternäre Operator braucht 3 Argumente. Je nach Wahrheitswert wird, nach links oder rechts gesprungen.

```
<Bedinging>?"wahr":"unwahr";

return myInt==3?5:0;//Wenn myInt ist 3, wird 5 dem return Operator
      uebergeben. Ansonsten wird 0 uebergeben
```

5 Compiler

Der Compiler übersetzt den Quellcode zu Maschinencode welcher durch das Zielsystem ausführbar ist.

5.1 Molularisierung

Der gesamte Code wird auf mehrere Module aufgeteilt, um die Übersichtlichkeit und Wiederverwendbarkeit zu verbessern. Ein Modul besteht in der Regel aus einer .h- und einer .c-Datei.
Die Schnittstelle des Moduls wird in der .h-Datei (Header) definiert und dokumentiert. Sie enthält: Prototypen für alle Funktionen, welche die Schnittstelle nach aussen bilden, Dokumentation wie diese Funktionen benutzt werden, structs, enums, unions, typedef, Präprozessormakros. Die Implementationen sind in der .c Datei

5.1.1 Information hiding

Information hiding bedeutet: nur so viele Informationen nach Aussen geben wie nötig. Daher globale Variablen sparsam verwenden. Diese Strategie macht Code sicherer und weniger fehleranfällig.

5.1.2 Richtiges Inkludieren

```
#include "eigenesModul.h"//eigenes Modul
#include <stdio.h>//standartlibrary
```

Um Mehrfachincludes zu verhindern, gibt es sogenannte **Include-guards**

```
#ifndef switch_h
#define switch_h
... //implementation
#endif

#pragma once//moderne Variante
```

5.2 Build Prozess

Der Build Prozess erzeugt ein ausführbares Programm und besteht wesentlich aus:

- jede .c-Datei compilieren (erzeugt .o Dateien)
- jede erzeugte .o-Datei linken

5.3 Präprozessor

Der Präprozessor kann einfache Textersetzung machen und so z.B. konstanten einsetzen.
Wichtige Makros sind dabei:

```
#define ALT NEU //ersetzt ALT durch NEU
#include "Datei" //Fuegt Inhalt aus Datei an die aktuelle Position
      ein
#include <Datei> //inkludiert Standartlibrary Datei
#ifdef Marke #endif
#ifndef Marke #endif //Prueft, ob Marke definiert / nicht
      definiert ist.
#error Nachricht //Bricht den Compile-Vorgang mit Nachricht ab
#define QUAD(a) a*a //Ersetzt alle QUAD(a) mit a*a(a ist eine
      Variabel) Ein beispiel einer Praeprozessormakrofunktion
```

Allerdings sind solche Präprozessormakrofunktionen sehr fehleranfällig da sie nur Textersetzung machen.

5.4 inline

Das Schlüsselwort **inline** ermöglicht es dem Compiler, den Funktionsaufruf komplett wegzuoptimieren. Es gibt jedoch keine Garantie, dass er dies auch tut! Es wird häufig zusammen mit static verwendet.

```
#include <stdio.h>
static inline int quad(int a) {
    return a * a;
}
int main() {
    int a;
    scanf("%d", &a);
    printf("Quadrat von %d ist %d.\n", a, quad(a));
    return 0;
}
```

Das Beispiel hat dieselbe Funktion wie die des Präprozessors, allerdings ohne Potenzial für undefined behaviour.

5.5 Anwendung des compilers

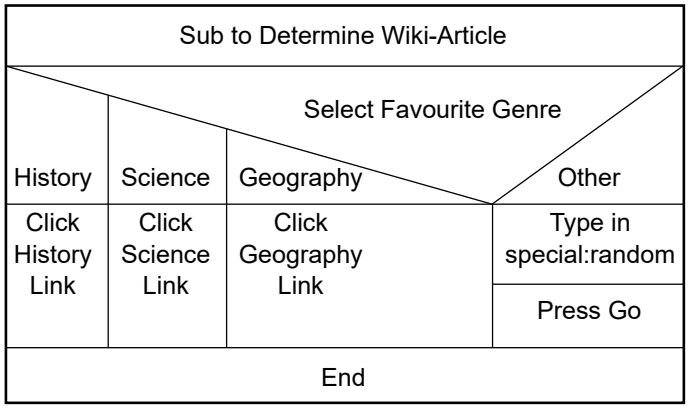
Es wird angenommen das clang verwendet wird:
Ein beispielhafter Compilvorgang wäre:
`clang -Wall -o test test.c`

5.5.1 Clang flags

- **-Wall** Alle Fehler und Warnungen ausgeben.
- **<File.c/.o/.a>** bindet die Datei in den compile Prozess ein.
- **-o** <File>Spezifiziert der Name der Ausgabedatei und macht diese ausführbar.
- **-c** <File.c>Kompiliert die .c Files zu .o Files. Diese sind noch nicht gelinkt. Diese müssen noch mit -o gelinkt werden.
- **-O3** Optimiert den Code auf Geschwindigkeit.
- **-Os** Optimiert den Code auf Grösse.
- **-I** <pfad> Pfad zu externen Libraries welche inkludiert werden sollen
- **-lm** falls math.h verwendet wird, muss zusätzlicher Machinencode verlinkt werden

6 Struktogramm

Nassi–Shneiderman Diagramme/Struktgramme werden verwendet, um eine Funktion eines Programmes zu visualisieren. Dafür gibt es verschiedene Grafische Elemente, welche man dann direkt in Quellcode übertragen kann.
Ein Beispiel:

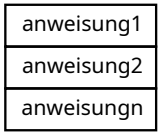


6.1 Sinnbilder in C

Es gibt verschiedene Sinnbilder für verschiedene Funktionen in C. Sie können auch ineinander verschachtelt werden:

6.1.1 Prozessblöcke

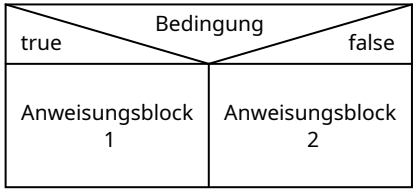
Eine Anweisung, die nach einem Block abgearbeitet wird, ist in einem Rechteck unter dem Block.




```
anweisung1;
anweisung2;
anweisungn;
```

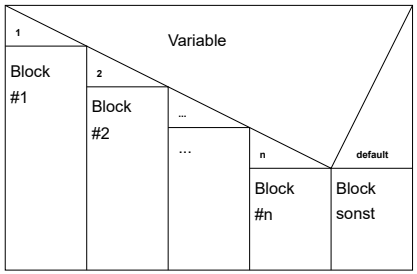
6.1.2 Decision / if

Eine Decision wird wie folgt realisiert. Wenn kein else verwendet wird kann der Block leer sein.



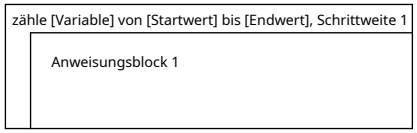

```
if(bedingung)
{
    Anweisungsblock 1;
}
else
{
    Anweisungsblock 2;
}
```

6.1.3 switch



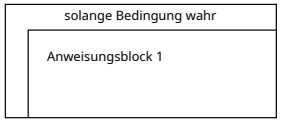

```
switch(variable)
{
    case 1:
        block1;
        break;
    case 2:
        block2;
        break;
    case n:
        blockn;
        break;
    default:
        blocksonst;
        break;
}
```

6.1.4 for



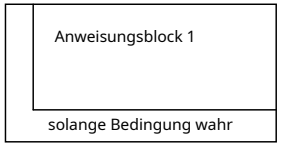

```
for(int i = Startwert;i<
      Endwert;i+=1)
{
    anweisungsblock1;
}
```

6.1.5 while




```
while(bedingung)
{
    anweisungsblock1;
}
```

6.1.6 do while




```
do
{
    anweisungsblock1;
}
while(bedingung);
```

7 Code beispiele

7.1 Hello World

```
#include <stdio.h>
int main()
{
    printf("Hello world\n");// schreibt Hello World in den output
    return 0;// beendet programm ohne fehler
}
```

7.2 Array Durchlaufen

7.2.1 1D

```
int speicher[10];//Array definition

for(int i = 0;i < 10;i++)
{
    speicher[i] = 1;//Das Array wird mit 1 gefuellt
}
```

7.2.2 mehrere Dimensionen

```
int speicher[3][4] = {
    {0, 1, 2, 3},
    {4, 5, 6, 7},
    {8, 9, 10, 11}};//Array definition mit initialisierung
for(int i = 0;i < 3,i++)
{
    for(int j = 0; j < 4; j++)
    {
        speicher[i][j] = (i*4) + j;//Das Array von 0 bis 11 der
     reihe nach gefuellt.
    }
}//Nach diesen for loops ist das Array gleich gebliben.
```

7.3 Iterativ vs rekursiv

Das Folgende Beispiel zeigt eine Implementation wo eine n-te Fibonacci Zahl bestimmt werden kann. Entweder iterativ oder rekursiv. Der Algorithmus :

$$f(1) = f(2) = 1 \qquad f(n) = f(n-1) + f(n-2)$$

wird verwendet.

```
#include <stdio.h>

unsigned int fibRek(unsigned int in);
unsigned int fibIt(unsigned int in);

int main()
{
    unsigned int input;
    int choose;

    scanf("%u",&input);
    printf("Iterativ oder rekursiv? (0 = rekursiv 1 = iterativ) \n
      ");
    scanf("%d",&choose);
    if(choose)
    {

        printf("Die %u te Fibonacci Zahl iterativ ist %u \n",input
     ,fibIt(input));
        return 0;
    }
    printf("Die %u te Fibonacci  Zahl rekursiv ist %u  \n",input,
     fibRek(input));
    return 0;
}

unsigned int fibRek(unsigned int in)
{

    if(in == 2 || in == 1)
    {
        return 1;
    }
    return (fibRek(in-1) + fibRek(in-2));
}

unsigned int fibIt(unsigned int in)
{
    unsigned int out = 1;
    unsigned int outPrev = 0;

    for(int i = 1;i < in;i++)
    {
        out = out + outPrev;
        outPrev = out - outPrev;
    }

    return out;
}
```

7.4 GGT

Die folgende Funktion nimmt 2 Zahlen (m,n) entgegen und gibt den GGT zurück.

```
unsigned int func(unsigned int m, unsigned int n)
{
    unsigned int r;
    unsigned int h;
    do
    {
        if (m < n)
        {
            h = m;
            m = n;
            n = h;
        }
        r = m % n;
        if (r != 0)
        {
            m = n;
            n = r;
        }
    } while (r!=0);
    return n;
}
```

8 Math.H

In der folgenden Tabelle sind befehle der Math.h Library aufgelistet. Wird math.h verwendet, darf nicht vergessen werden dem Compiler **-lm** anzugeben um die Math library zu linken.

Name	Mathematische Form	Notes
acos()	arccos()	
asin()	arcsin()	
atan()	arctan()	
ceil()	$\lceil x \rceil$	Aufrunden
cos()	cos()	
cosh()	cosh()	
exp()	e^x	
fabs()	$\lfloor x \rfloor$	Abrunden
fmod(x,y)	x mod y	% verwenden
log()	ln()	
log10()	$\log_{10}$	
pow(x,y)	x^y	
sin()	sin()	
sinh()	sinh()	
sqrt()	$\sqrt{x}$	
tan()	tan()	
tanh()	tanh()	

9 emotional support meme

Für den Fall das während der Prüfung emotionaler Support gefragt ist ein Meme: