Prog C

HS 2023, Prof. Dr. Christian Werner

Fabian Steiner, 27. Dezember 2023

V1.3.0, Vorlage von Simone Stitz

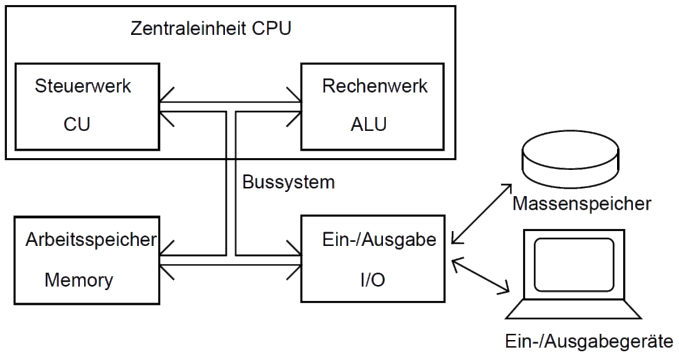
1 Grundlagen

1.1 Trivia

Designer : Dennis Ritchie
Erstveröffentlichung : 1972

1.2 Rechnersystem system

Grundaufbau eines Rechnersystem



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

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

1.3 Zahlensysteme

1.3.1 Binär direkt

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

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

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

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

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

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.

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[END OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

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.

$$\overbrace{1100}^{\text{A}}\,\overbrace{0101}^{5} \rightarrow \text{0xA5}$$

Schreibweisen: $A5_{\text{h}}$, A5H, $A5_{\text{hex}}$, A5$, 0xA5
Typischerweise mit 0x vor Beginn der Zahl.

1.3.6 mrechnung 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

Sie legen fest,

- wie lang das Bitmuster an der zugehörigen Speicherstelle ist,
- und 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 Typedef 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. Ein Int ist der effizienteste Datentyp.

2.3 Gleitpunktzahl

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

	Sign	Exponent	Mantisse
float	1	8	23
double	1	11	52

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

2.4.1 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 Value Deklaration kann ausgelassen werden. Der Compiler vergibt von 0 auf aufzählend automatisch Werte. Es kann strategisch eine Zahl dazwischen gesetzt werden um gewisse numerische Zahlen zu Erhalten.
Wenn man ein Enum verwendet will, muss das Schlüsselwort Ënumïmmer 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
```

2.4.2 Ganzzahkonstanten

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

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

2.5 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 ein Pointer dereferenziert(Operator : *) wird, 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 die 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.

```
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
```

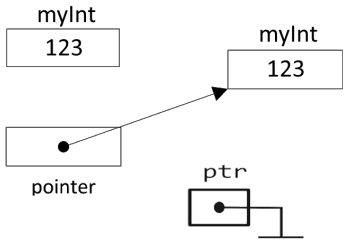
2.5.1 Pointer Arithmetik

- 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(==, !=,...).

2.5.2 Memorymap

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



2.6 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 ichbins; //korrekt
int 0ichbins; //falsch wegen der Zahl am Anfang
char ichbins1; //erlaubt da Zahl am Ende
double schwarz = 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        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
inline      _Alignof    _Complex    _Noreturn
restrict    _Atomic     _Generic    _Static_assert
_Alignas    _Bool       _Imaginary  _Thread_local
```

2.7 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 durch Schreiben des Typs neben der umzuwandelnden Zahl erreicht.

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

2.8 Scope

- **global**
  nicht in eine 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** variablen sind dabei eine Ausnahme(siehe Static→ Typattribute).

2.9 Arrays

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

```
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.
```

2.9.1 Initialisierung

Man kann ein Array entweder komplett oder gar nicht initialisieren.

```
int a[4] = {1,2,9,1};//vollstaendige initialisierung
int b[4] = {6,9};// 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
```

2.9.2 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!
```

2.9.3 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.

2.9.4 Handhabung von Arrays

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

2.9.5 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 vorgestellt wobei beim Aufruf [Zeile][Spalte] mitgegeben werden.
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, 3, 5, 7, 2, 4, 6, 8, 3, 5, 7, 9};
```

2.10 Pointer und Arrays

Pointer und Arrays können ähnlich verwendet werden. In aufgerufenen Funktionen kann mit einem Pointer sowie 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
```

2.10.1 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);
```

2.11 Typ-attribute

2.11.1 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 ein 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.
```

2.11.2 Volatile

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

2.11.3 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.

2.12 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];
}
```

2.12.1 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
}
```

2.12.2 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.

2.12.3 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.

2.13 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.

```c
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.
```

2.14 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.

```c
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

3.0.1 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 die String Funktionen einen Typ, der alle Zeichen bearbeitet oder der nur n Zeichen bearbeitet.

3.1 scanf

Header <stdio.h>wird verwendet.
Scanf wird verwendet, um Eingaben des Nutzers entgegenzunehmen.
General 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	%s	%d
ein char	eine Zeichenkette	Signed int in dezimaldarstellung
%u	%o	%x
unsigned int in dez darstellung	unsigned int in okt darstellung	unsigned int in hex darstellung
%f	%lf	
float	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
bsp:

```c
#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 :
scanf("text %<Format Spzifizierung>text", <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 ".ünd einer Zahl kann die Präzision spezifiziert werden. Standart bei Float & double ist 6.

```c
#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 returnt 0 wenn diese gleich sind.

```c
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.

```c
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.

```c
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.

```c
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.

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

Bestimmt die Länge des Strings.

3.8 Mem Funktionen

Es gibt einige Mem funktionen die nützlich sein können

```c
#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(mit korrekter Syntax)

```
#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
    }

    case(<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 nie gerechtfertigt verwendet)

4.1.2 GoTo

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

```
#include <stdio.h>

int func()
{
    printf("Do func Stuff\n");
inFunc: //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 ist eine Zusammenfassung von Anweisungen, die ein und Ausgabewerte haben kann.
<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 (Meist in .h Files festgelegt). 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. Es kann auch weggelassen werden.

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)
- Jeder rekursive Algorithmus kann auch iterativ formuliert werden
- 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. Haben Operatoren dieselbe Priorität besagt die Assoziativität in welcher Reihenfolge ausgewertet wird.

Priorität	Operatoren		Assoziativität

Priorität 1	`()`
	`[ ]`
	`-> .`
	`++ --`
	
	`(Typname) {}`
		Funktionsaufruf
		Array-Index
		Komponentenzugriff
		Inkrement, Dekrement als Postfix
		compound literal[96]
			links
			links
			links
			links
			
			links

Priorität 2	`! ~`
	`++ --`
	
	`sizeof`
	`+ -`
	`(Typname)`
	`* &`
		Negation (logisch, bitweise)
		Inkrement, Dekrement als Präfix
		
		Vorzeichen (unär)
		cast
		Dereferenzierung, Adresse
			rechts
			rechts
			
			rechts
			rechts
			rechts
			rechts

Priorität 3	`* /` `%`	Multiplikation, Division modulo	links links
Priorität 4	`+ -`	Summe, Differenz (binär)	links
Priorität 5	`<< >>`	bitweises Schieben	links

Priorität 6	`< <=`
	
	`> >=`
		Vergleich kleiner, kleiner gleich
		Vergleich größer, größer gleich
			links
			
			links

Priorität 7	`== !=`	Gleichheit, Ungleichheit	links
Priorität 8	`&`	bitweises UND	links
Priorität 9	`^`	bitweises Exklusives-ODER	links
Priorität 10	`|`	bitweises ODER	links
Priorität 11	`&&`	logisches UND	links
Priorität 12	`||`	logisches ODER	links
Priorität 13	`?:`	bedingte Auswertung	rechts
Priorität 14	`=` `+=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=`	einfache Wertzuweisung kombinierte Zuweisungs-operatoren	rechts rechts
Priorität 15	`,`	Komma-Operator	links

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 Machinecode 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 Includeguards

```
#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 die selbe Funktion wie die des Präprozessors, allerdings ohne Potenzial für undefined behaviour.

5.5 Anwendung des compilers

Es wird angenommen das clang verwendet wird:

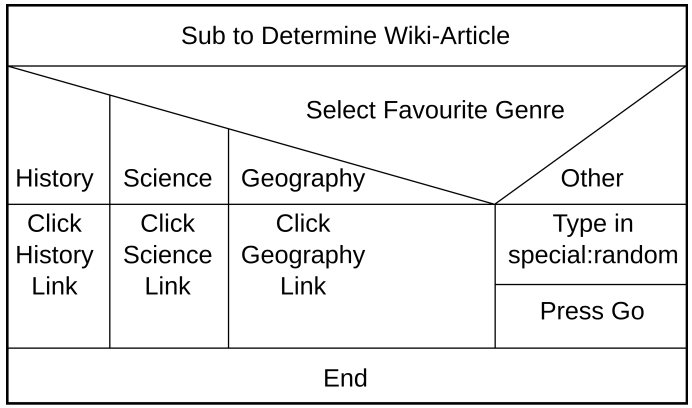
Ein beispielhafter compilvorgang wäre:
clang -Wall -o test test.c

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:

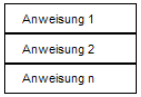


6.1 Sinnbilder in C

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

6.1.1 Prozessblöcke

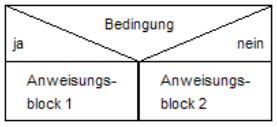
Eine Anweisung, die nach einem Block abgearbeitet wird, ist in einem Rechteck unter dem Block.




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

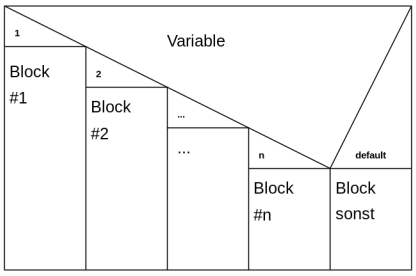
6.1.2 Decision / if

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



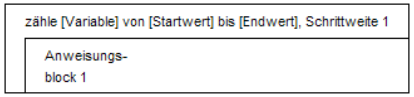

```
if(bedingung)
{
    Anweisungsblock 1;
}
else
{
    Anweisungsblock 2;
}
```

6.1.3 switch



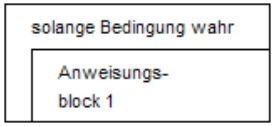

```
switch(variable)
{
    case 1:
        block1;
        break;
    case 2:
        block2;
        break;
    case n:
        blockn;
        break;
    default:
        blocksonst;
        break;
}
```

6.1.4 for



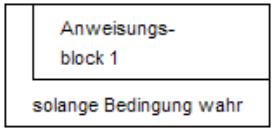

```
for(int i = Startwert;i<
    Endwert;i+=1)
{
    anweisungsblock1;
}
```

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");
    return 0;
}
```

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;
}
```

8 emotional support meme

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