2018

Zusammenfassung ProgC

SEP 2018

PASCAL BRUNNER

1 Inhaltsverzeichnis

2 Grundlegendes

- Erlaubt es sehr effizienten Code zu schreiben
    - Grosse Kontrolle was der Code macht
    - Effizienz hinsichtlich Memory-Footprint und Ausführgeschwindigkeit
- Wichtigste Systemsprache
    - General Purpose Betriebssysteme sind in C/C++ (Unix, Linux, Windows…)
    - VM in C geschrieben
    - Grosse Relevanz in Embedded Systems (IOT etc.)
- Fördert das Verständnis für das unterliegende System
    - Speichermanagement
    - Berücksichtigung Architektur notwendig
- Grosser Umfang an Libraries
- C grundsätzlich fehleranfälliger wie Java
- C-Compiler erzeugt Maschinencode (Java erzeugt Bytecode)
- C ist eine prozedurale Programmiersprache (**nicht** objektorientiert)
    - Keine Klassen
    - Funktionen statt Methoden
    - C-Programme bestehen im Wesentlichen aus Funktionen

3 Programmelemente

Einfaches Hello-World-Programm

```c
/* helloworld.c */
#include <stdio.h>   // declares printf
#include <stdlib.h>  // declares EXIT_SUCCESS

/* Hauptprogramm */
int main(void)
{
  (void)printf("Hello World in C\n");
   return EXIT_SUCCESS;
}
```

Abbildung 1 Einfaches HelloWorld Programm in C

Jedes C-Programm hat eine *main*-Funktion, welche als Einstiegspunkt ins Programm dient

- Es kann beliebig in weitere Funktionen verzweigt werden
- Wird die main-Funktion verlassen, so terminiert das Programm

Ein C Programm soll die Endung .c haben (bspw. main.c / person.c / xyz.c)

Ein C-Programm wird mit einem C-Compiler (bspw. gcc) zu einem ausführbaren Programm.

```
$> gcc -o helloworld helloworld.c
```
→ -o spezifiziert den Namen des Programms

Danach kann das Programm direkt (ohne VM) gestartet werden

```
$> ./helloworld
```

Wichtig! Ein C-Programm muss für jedes System neu kompiliert werden und manchmal müssen sogar Teile des Source-Codes angepasst werden.

3.1 Schlüsselwörter

Die Schlüsselwörter sind reserviert und dürfen nicht als Namen (z.B. Variablen) verwendet werden

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
typedef	union	unsigned	void	while	volatile	
_Bool	_Complex		_Imaginary			

Abbildung 2 Schlüsselwörter in C

3.2 Variablen- und Funktionsnamen

- Sämtliche alphanumerische Werte sind zulässig
- Case-Sensitiv -> Hello ≠ hello
- Das erste Zeichen muss ein Buchstabe oder _ sein
- Darf keines der durch die Sprache reservierten Wörter sein

Konvention

- Einfache Variablenname sollen prinzipiell aus Kleinbuchstaben bestehen
- Zusammengesetzte Namen werden entweder mit _ oder CamelCase geschrieben → max_value oder maxValue

3.2.1 Definierte Variabeln

```
1. /* Definierte Variablen */
2. return EXIT_SUCCESS; // in main, == exit(0)
3. return EXIT_FAILURE; // in main, == exit(1)
4. if (EOF) {...} // EOF == -1
5. if (pointer == NULL) {...} // NULL = #define NULL (void *) 0
```

3.3 Datentypen

C kennt vier Elementare Datentypen, wobei die Wertebereiche hardwareunabhängig sind

char	1 Byte	-128 bis 127 oder 0 bis 255
int	4 Bytes	-2^{31} bis $2^{31}-1$
float	4 Bytes	$-3.4 \cdot 10^{38}$ bis $3.4 \cdot 10^{38}$
double	8 Bytes	$-1.79 \cdot 10^{308}$ bis $1.79 \cdot 10^{308}$

Abbildung 3 Die vier elementare Datentypen in C

Diese vier Datentypen können noch weiter definiert werden, welche somit die Grösse und den Zahlenbereich weiter angeben.

signed char	1 Byte	-128 bis 127
unsigned char	1 Byte	0 bis 255
[signed] short [int]	2 Bytes	-32768 bis 32767
unsigned short [int]	2 Bytes	0 bis 65535
[signed] int	4 Bytes	-2^{31} bis $2^{31}-1$
unsigned [int]	4 Bytes	0 bis $2^{32}-1$
[signed] long [int]	8 Bytes	-2^{63} bis $2^{63}-1$
unsigned long [int]	8 Bytes	0 bis $2^{64}-1$
[signed] long long [int]	8 Bytes	-2^{63} bis $2^{63}-1$
unsigned long long [int]	8 Bytes	0 bis $2^{64}-1$
long double	10 Bytes	$-1.2 \cdot 10^{4932}$ bis $1.2 \cdot 10^{4932}$

Abbildung 4 Erweiterung der vier Datentypen in C

Je nach unterliegender Hardware, werden unterschiedliche Anzahl Bytes reserviert. In einer 32-Bit-Architektur wird bspw. ein Int immer in 4-Bytes dargestellt, ein Short 2-Bytes und ein *long* in 8-Bytes.

Die einzige Bedingung welche C macht ist, dass in jedem Fall *short* nicht länger als ein *int* ist und ein *long* sicher nicht kürzer als ein int.

Unsigned bedeuted, dass nur positive Zahlen (inkl. 0) zulässig sind. Defaultwert von *int* ist *signed*.

Grundsätzlich sollte man in C immer mit *double* statt mit *float* arbeiten, ausser man hat einen guten Grund für float.

Soll eine Zahl explizit als long interpretiert werden, so wird l oder L angehängt (für long long: LL oder ll; zusätzlich unsigned: U oder u):

- 35000l (oder 35000L)
- 123LLU
- 0104270L
- 0x8868L

Abbildung 5 explizite Angabe von Nummern

Mit `#include <stdint.h>` besteht die Möglichkeit Integer Typen von fixer Grösse anzubieten:

8-Bit Integer:	`int8_t,`	`uint8_t`
16-Bit Integer:	`int16_t,`	`uint16_t`
32-Bit Integer:	`int32_t,`	`uint32_t`
64-Bit Integer:	`int64_t,`	`uint64_t`

➔ Diese Typen sind vor allem für die hardware-nahe Programmierung relevant

3.4 Literale (dezimal, oktal, hexa)

Zahlen welche mit 1…9 beginnen, werden als **dezimal** *(bspw. 397)* interpretiert. Zahlen welche mit 0 beginnen, werden als **oktal** *(bspw. 037)* interpretiert. Zahlen mit 0x oder 0X werden als **hexadezimal** *(bspw. 0x23)* interpretiert. Per Default wird *dezimal* als *signed int* und *oktal/hexadezimal* als *unsigned int* interpretiert

3.5 Literale (Symbolische Konstante)

In C gibt es keinen Datentyp Strings. Strings werden als char-Arrays intern (mit dem ASCII Wert) abgespeichert und terminieren auf '\0'. → Ein Wort hat immer Wortlänge+1 Array-Elemente!

«ZHAW» => 'Z' 'H' 'A' 'W' '\0' → Array mit 5 Elemente

Wichtig! Es gibt keine Funktion bei welchem man die Länge eines Wortes abfragen könnte. Entweder man inkrementiert jeweils einen Counter und hat so die Länge oder man muss es anhand der Bytes herausfinden → sizeof(word)/sizeof(word[0]) → denn der Rückgabewert ist nur die Anzahl Bytes. Dementsprechend rechnet man alle Bytes geteilt durch die Bytes welches das erste Element einnimmt.

3.6 ASCII-Tabelle

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Abbildung 6 ASCII-Tabelle

3.7 Deklaration und Definition

Eine **Deklaration** bestimmt wie ein Name verwendet werden kann. Wobei eine Variablen-**Definition** Speicher alloziert, eine Funktions-**Definition** den Funktionsbody angibt. Eine Definition ist immer auch eine Deklaration.

```
double hoehe;
int laenge, breite;
double r, radius = 15.0;
```

```
const double pi = 3.14159;
```

```
typedef int Index;
```

Abbildung 7 Variablen / Konstante / typedef Deklaration

➔ Variablen werden meist innerhalb von Funktionen deklariert (Ausnahme: globale und statische Variablen)

In der Praxis wird man die Struktur deshalb oft gleich auch als Alias definieren → mit typedef möglich.

Mit typedef:

```
typedef struct {
  double x;
  double y;
  double z;
} Point3D;
```

Vgl. ohne typedef:

```
struct point3D {
  double x;
  double y;
  double z;
};
```

Abbildung 8 Typedef in Kombination mit struct

3.8 Operatoren

- Arithmetische Operatoren: `+ - * / %` (modulo) Vorzeichen (+ und -)
- Relationale Operatoren: `> >= < <=`
- Logische Operatoren: `&& ||`
- Gleichheitsoperatoren: `== !=`
- Negationsoperator: `!`
- Increment / Decrementoperator: `++ --`
- Bitoperatoren: `&` (AND) `|` (OR) `^` (XOR) << (bit shift left) >> (bit shift right) ~ (Einerkomplement)
- Zuweisung: `= += -= *= /= %= &= ^= |= <<= >>=`
- Conditional Expression: `? :` x = cond? truecase: falsecase;
- Address-of Operator, Dereferenz-Operator: `& *`

Abbildung 9 Operatoren in C

bindet: stärksten

Symbol	Type of Operation	Associativity
`() [] . ->`	Expression	Left to right
`! ~ ++ -- + - * & (type) sizeof`	Unary	Right to left
`* / %`	Multiplicative	Left to right
`+ -`	Additive	Left to right
`<< >>`	Bitwise shift	Left to right
`< <= > >=`	Relational	Left to right
`== !=`	Equality	Left to right
`&`	Bitwise-AND	Left to right
`^`	Bitwise-exclusive-OR	Left to right
`|`	Bitwise-inclusive-OR	Left to right
`&&`	Logical-AND	Left to right
`||`	Logical-OR	Left to right
`? :`	Conditional-expression	Right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	Simple and compound ass.	Right to left
`,`	Sequential evaluation	Left to right

wenigsten

Abbildung 10 Bindung und Leserichtung

4 Kontrollstrukturen

- If/else
- For-Schleife
- While-Schleife
- Do-While-Schleife
- Switch-Anweisungen

```
switch (n) {
case 1:
  result = 1;
  break;
case 2: case 3: case 4:
  result = 10;
  break;
default:
  result = 0;
  break;
}
```

Abbildung 11 Vorlage Switch-Case

4.1 Boolean

In C gibt es keinen boolsche Ausdrücke, jedoch gibt es die Regel, dass falls der Wert **0 = false** und **sonst true**. Dementsprechend muss es auch kein ganzzahliges Resultat sein → while (1.5) als Beispiel

```
1.  /* Boolean */
2.  int i = 0;
3.  int j = 1;
4.  #define TRUE 1;
5.  #define FALSE 0;
6.  if (i) {...}; // will never be executed because 0 is always false
7.  if (j) {...}; // Everything != 0 is true, will always be executed
8.  if (j == TRUE) {...}; // Will be executed
9.  if (i == TRUE) {...}; // Never be executed
10.
11. /*boolean als enum definieren*/
12. Typedef enum {
13.   false, //wird mit 0 initialisiert
14.   true //wird hier mit 1 initialisiert
15. } Boolean;
```

4.2 Enum

```
1.  // Beginnt bei 1 und zählt hoch --> Donnerstag hat Wert 4.
2.  enum wochentage {
3.      Montag = 1, Dienstag, Mittwoch, Donnerstag, Freitag
4.  };
5.  int i;
6.  i = Montag; // OK
7.  wochentag heute;
8.  heute = Mittwoch; // OK
9.  heute = 3; // Warning
```

```
1.  enum { rot, gruen, blau };
2.  printf("%d", blau); // prints: 2 ( enum value starts with 0
```

4.3 Struct

Siehe auch 7.5.4 Struct und Pointer S.33

```
typedef struct {
    char lastname[20];
    char firstname[20];
    unsigned age;
}
Person;
/** Listelement called "node"*/
typedef struct node {
    Person person; // Person (Struct)
    struct node * next; // Pointer to the next linked listnode
}
node;
/** Iterates through the list* @param the head node and the function to execute as a callback*/
void traverse(node * head, callback f) {
    node * cursor = head;
    Person person;
    while (cursor != NULL) {
        f(cursor);
        person = curser.person; // Access with normal dot
        cursor = cursor - > next; // Access with arrow because it's a pointer
    }
}
```

```
#include <stdio.h>
struct point3D * pptr;
(void)printf("Point is %d %d %d\n", pptr->x, pptr->y, pptr->z);
// -> is equivalent to (*pptr).x
```

4.4 Input/Output

%d, %i (int); %u (unsigned int)
%c (char)
%s (char *, Zeichen des Strings werden ausgegeben bis \0 gefunden wird)
%f (double, float)

Abbildung 12 Konvertierungsoperatoren

Bei Zahlen kann zwischen % und dem Konvertierungszeichen f die minimale Länge m (Anzahl Zeichen) und die Anzahl Dezimalstellen d angegeben werden → %m.df

```
double v = 5.12345;
(void) printf("%f", v);   /* Output: 5.123450 (default d=6) */
(void) printf("%.3f", v);     /* Output: 5.123 */
(void) printf("%10.3f", v);   /* Output:       5.123 */
```

Abbildung 13 Minimallänge und Dezimalstellen angeben

4.4.1 Konvertierungsoperatoren

```
/* printf(), scanf() --> use %argument sign*/
%c character;
%d decimal(integer) number(base 10);
%e exponential floating - point number;
%f floating - point number;
%i integer(base 10);
%o octal number(base 8);
%s a string of characters; //prints as long as nextchar is ‘\0’
%u unsigned decimal(integer) number;
%x number in hexadecimal(base 16);
```

```
11. %% print a percent sign;
12. \% print a percent sign;
```

4.4.2 Int – double- long – short Umrechnungen

```
1.  double x = 3.0, y = 2.0;
2.  printf("%f", x/y-(int)(x/y)); // 0.500000
3.  printf("%f", (int)(x/y)-(int)(x/y)); // Warning should be %d instead of %f (->Casting to
    int) and prints 0.000000
4.  printf("%f", (x/y) - (x/y)); // 0.000000
5.  printf("%f\n", (int)(x/y) - (x/y)); // -0.500000
```

```
/* expression.c */
int main(void) {
  double x, y = 3.0;        /* x = ?     y = 3,0   */
  int i, j = 4;             /* i = ?     j = 4     */

  i = 2.5 + y;              /* i = 5               */
  x = 5 * i / 3;            /* x = 8,00000         */
  x = 5.0 * i / 3;          /* x = 8,333333        */
  i += j;  — erhöht auch j  /* i = 9               */
  i = ++j;                  /* i = 5     j = 5     */
  i = j++;                  /* i = 5     j = 6     */
  x = 3 + (y = i + 5.0);    /* x = 13,0  y = 10,0  */
}
```

5 Funktionen

Funktionen sind die zentralen Komponenten in C. Es erlaubt die Programme zu strukturieren und Code-Duplikationen zu vermeiden. Der Einstiegspunkt eines Programmes (→ main) ist auch nichts anderes wie eine spezielle Funktion. Die Funktionen in C sind mit den Methoden in Java zu vergleichen.

Eine Funktion besteht immer aus einem *Funktionsnamen* (identifiziert die Funktion **eindeutig**), *Parameterliste* (mehrere Paremeter mit Komma getrennt; void bzw. leer, wenn kein Parameter übergeben werden), *Datentyp des Rückgabewerts* (void, wenn nichts zurückgegeben wird).

```
Datentyp_des_Rückgabewerts Funktionsname(Parameterliste)
```

Abbildung 14 Vorlage Funktion

In einer Funktionsdeklaration (bspw. im Header-File) müssen die Parameter in der Parameterliste keine Namen haben.

5.1 Declared-Before-Used

Dies ist ein Grundsatz welcher fundamental in C ist. Jeder Name (Typ, Variable, Funktion, #define) muss deklariert sein bevor er verwendet wird.

5.2 One-Definition-Rule (ODR)

Jeder Name darf nur eine Definition im gesamten Programm haben, sprich eine Funktion darf beliebig oft deklariert werden, jedoch nur einmal definiert werden.

Ausnahmen

- Dieselbe Typdefintion darf in verschiedenen Source Files vorkommen, muss aber identisch sein
- Inline Funktionen dürfen mehrfach identisch vorkommen

```
1. Inline int max_int(int a, int b) {
2.     return a < b ? b : a;
3. }
```

Best Practice

Öffentliche Deklarationen werden in Header Files untergebracht. Jedes File, welches diese Funktionen braucht, includiert (#include), dann dieses Header-File und gibt die Deklaration selber nicht mehr an.

5.2.1 Dich wichtigsten Header-Files

Header File	Funktionen
#include <stdlib.h>	defines EXIT_SUCCESS defines EXIT_FAILURE
#include <stdint.h>	Deklariert Integertypen für fixe Grössen (siehe 3.3 Datentypen S. 7)
#include <stddef.h>	Definiert size_t für sizeof() Funktion
#include <stdio.h>	Funktionen für einfachen Input/Ouput -> puts(«Hello»), putchar('A'), getchar(void) Funktionen für formatierte Ausgabe -> Printf(format-string, arg1, arg2, …)

	Funktionen für das Auslesen aus dem Standard Input -> scanf(format-string, arg1, arg2)
#include <string.h>	Inkludiert sämtliche Funktionen rund um Strings -> strcpy, strncpy, strcat etc.

```
/* Console*/
double number;
printf("Enter a number\n");
scanf("%f\n", & number); // Saves a number on the address from number
/*Error-Handling scanf*/
Int result = scanf(“%d %d”, &a, &b);
If(result != 2) {
…ERROR HANDLING…}
If(a<0){…ERROR HANDLING…}
If(b<0){…ERROR HANDLING…}
```

5.3 Definition vs. Deklaration

Deklaration: Dem Compiler sagen, dass es etwas gibt. Es gibt eine Funktion foo die void returned und keine Parameter nimmt

```
void foo(void); //Beispiel 1
int max (int a, int b); // Beispiel2
int max (int, int) //identisch Beispiel 2 wäre auch zulässig
```

Definition: Dem Compiler sagen, dass er dieses etwas anlegen soll. Du sagst dem Compiler, dass die (bereits bekannte Funktion *foo* – denn jede Definition impliziert eine Deklaration) exakt so aussieht

```
void foo() {}
```

Wieso braucht man Definition und Deklaration?

Der C-Compiler geht nur einmal linear durch den Source-Code. Wenn nun eine Funktion gebraucht wird vor deren Definition auftritt, so hat der Compiler diese Funktion noch nie gesehen und bricht mit einem Fehler ab. Durch das, dass man die Deklaration am Anfang des Source-Files macht, hat der Compiler diese Funktion schon einmal gesehen. Des Weiteren liefert es bereits dem Compiler die Information wie der Funktionsaufruf stattfinden wird. Falls es sich um eine Funktion handelt, welche keine Definition hat, so meldet der Linker einen Fehler.

5.4 Aufruf von Funktionen

- Es müssen sämtliche Parameter übergeben werden
- Reihenfolge der Parameter, entsprechend der Funktionsdefinition
- Der Rückgabewert kann, muss aber nicht ausgewertet werden
- Wenn der Rückgabewert nicht ausgewertet wird, sollte zur Dokumentation der Aufruf von (void) casted werden

```
(void) max(15, 4);
```

5.5 Code Beispiel Deklaration / Definition / Aufruf

```
#  include < stdio.h >

Int max(int a, int b);  //Deklaration
```

```
4.
5.  Int main(void) {
6.      Int x, y, p;
7.      (void) printf(«1. Zahl: «);
8.      (void) scanf(« % d», & x);
9.      (void) printf(«2. Zahl: «);
10.     (void) scanf(« % d», & y);
11.     (void) printf(« Maximum : % d\ n”, max(x, y)) ;  //Aufruf
12. }
13. Int max(int a, int b) {  //Definition
14.     If(a >= b) {
15.         Return a;
16.     }
17.     Return b;
18. }
```

5.6 Konsoleneingaben lesen

getchar()	Der Befehl hält das Programm an und es läuft erst weiter wenn der User die Entertaste drückt
	`char c;`
	`printf("Mit welchem Buchstaben beginnt ihr Vorname? ");`
	`c = getchar();`
	`printf("\nIch weiss jetzt, dass Ihr Vorname mit '%c' beginnt.\n", c);`

scanf()	• Liest beliebige Zeichen von der Konsole
	• Gibt eine negative Zahl aus wenn es nicht erfolgreich war, ansonsten wird die Anzahl gelesenen Ziffern ausgegeben.
	• Falls ein String in einem zu kurzem Array gespeichert wird wird dieser beim nächsten Aufruf von scanf() weiter gelesen → ACHTUNG mit %19s kann die Länge angegeben werden. → Für char array[20] nur 19 wegen '\0'
	• Wenn ein Fehler auftritt, z.B. %d und ein String versuchen zu speichern → Es wird nicht mehr weiter gelesen.
	`char str1[20];`
	`printf("Enter name: ");`
	`scanf("%19s", str1); // kein & nötig weil array immer eine Adresse ist`
	
	`int alter;`
	`printf("Wie alt sind sie? ");`
	`scanf("%d", &alter); // & nicht vergessen`

sscanf()	Int var = sscanf(argv[1], «%d», %rappen) ; • Argv, überprüfen ob vom typ int, speichern in variable rappen • Gibt 1 zurück wenn erfolgreich
fscanf()	• Für File input verwendet

5.7 Konsolenausgabe schreiben

printf()	• Immer mit (void) casten • (void)printf ("%s \n", "A string");

puts()	• Schreibt einen String auf die Konsole inklusive Newline
	• Nicht erfolgreich → EOF (negative Zahl, aus Standard library)
	• Erfolgreich → Positive Zahl
	`char str1[15];`
	`char str2[15];`
	`strcpy(str1, "tutorialspoint");`
	`strcpy(str2, "compileonline");`
	`puts(str1);`
	`puts(str2);`

5.8 File einlesen

fgets()	fgets(pointer auf String, Länge n, Pointer auf File) (n normalerweise Länge des Strings) • Liest so lange im File bis n-1 Zeichen gelesen sind, oder das Newline Zeichen oder EOF

	• Return Pointer auf String oder null-Pointer

5.9 Parameter und Rückgabewerte

Zugelassene Parameter:

- Basis Datentyp (int, double, ...)
- Strukturen (struct)
- Arrays
- Pointer

Zugelassene Rückgabewerte:

- Basis Datentypen (int, double, …)
- Strukturen (struct)
- Pointer

Arrays können **nicht** zurückgegeben werden, jedoch einen Pointer auf deren Datenbereich

Eigenheiten return:

- Wird das Ende einer Funktion erreicht, so wird die Funktion auch ohne Angabe von return verlassen. → Fehlt das return → Programmierfehler
- Returnwert erfolgt «by value»

5.9.1 Rückgabewert von lokalen Variablen

Achtung, wenn eine Adresse als Rückgabewert verwendet wird darf es keine lokale Variable sein. Z.B.

```
1.  //Deklaration Struct wurde vorgenommen
2.  Struct point3D* initPoint3D(void){
3.     struct Point3D p;
4.     p.x = 5;
5.     p.y = 10;
6.     p.z = 20;
7.     return &p; // Warning: function returns address of local variable
8.  }
```

Dies kann mit 3 Möglichkeiten gelöst werden:

1) Lokale Variable als static deklarieren

2) Der Speicherplatz innerhalb der Funktion dynamisch mit malloc() allozieieren

3) Der Speicherplatz wird von der aufrufenden Funktion alloziert. Danach wird ein Pointer der Funktion die den Pointer zurückgibt übergeben.

5.10 Lokale Variablen

Lokale Variablen sind nur innerhalb der Funktion sichtbar. Es entstehen somit keine Konflikte mit Variablen in andere Funktion, die denselben Namen haben.

Wichtig! Variablen in der main-Funktion, sind auch lokale Variablen → Alle innerhalb von Funktionen deklarierten Variablen sind lokale Variabeln.

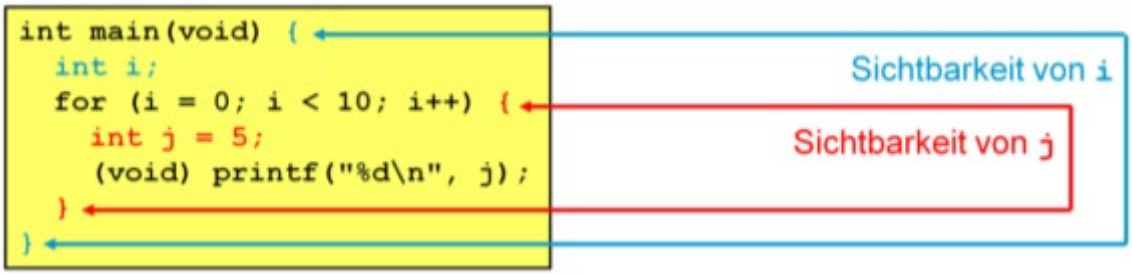


Abbildung 15 Sichtbarkeit von lokalen Variabeln

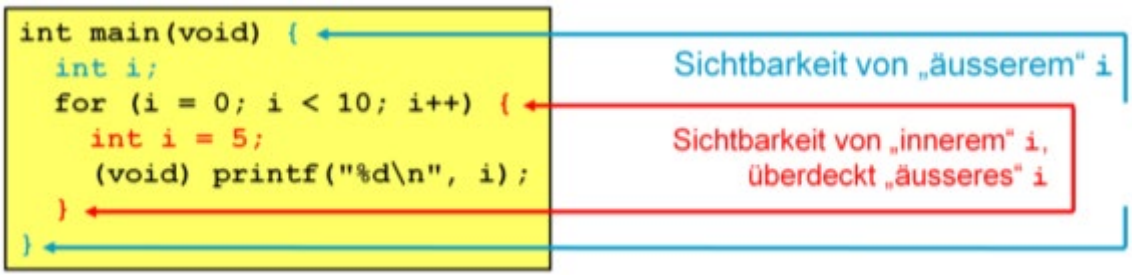


Abbildung 16 Sichtbarkeit / ebenfalls zulässig aber unschön

5.11 Globale Variablen

Globale Variablen werden ausserhalb von allen Funktionen definiert. Damit kann auf dem gesamten Programm auf diese Variablen zugegriffen werden. Der **Initialwert** ist implizit 0 oder der explizit angegebene Wert. Globale Variablen sollen grundsätzlich mit Vorsicht verwendet werden. Am besten gibt man diese gleich nach den #includes und #defines an.

Verwendung von globalen Variablen in anderen Source Files

Wenn eine globale Variable in einem anderen Source File verwendet werden soll, muss die Variable dort mit extern int max; deklariert werden → z.B. in einem Header-File, welches anschliessend includiert wird

```
1. extern int max;
```

5.12 Statische Variablen

Statische Variablen verhalten sich identisch wie globale Variablen, mit dem Unterschied, dass sie nur innerhalb der Quelldatei sichtbar sind. Statische Variablen **können nicht(!)** in anderen Source Files verwendet werden. Würde man eine statische Variable in einem Header-File deklarieren, so hätte jedes File, welches den Header includiert, seine eigene Instanz dieser Variable. Generell gelten statische Variablen in Header-Files als Programmierfehler. Es können auch Funktionen als statisch deklariert und definiert werden

```
1. static int max = 0;
```

Es besteht die Möglichkeit, dass eine statische Variable innerhalb eines Blocks definiert wird. Der gewünschte Effekt ist, dass der Wert der Variable beim nächsten Aufruf der Funktion «überlebt» hat.

```
1. int checkMax(int a) {
2.     Static int max = 0 //Deklaration
3.     …
4. }
```

5.13 Pure-Funktionen

Eine Funktion welche statischen Variable definiert, hat einen State, d.h. sie ist nicht mehr eine «pure» Funktion im mathematischen Sinn. Eine Pure-Function ist eine Funktion, die nur von den Argumenten abhängt. Inpure-Functions sind z.B. alle Funktionen die von einer globalen und/oder statischen Variablen abhängen, oder Funktionen, die nicht bei jedem Aufruf mit identischen Argumenten dasselbe zurückgeben (z.B. Zeitabfrage, etc.)

5.14 Rekursive-Funktionen

Rekursive Funktionen sind in C zulässig.

```
1. int fakultaet(int n) {
2.     if(n < 2) {
3.         return 1;
4.     } else {
5.         return n * fakultaet(n - 1);
6.     }
7. }
```

5.15 Call by Value

- Die aktuellen Argument Werte werden in die Parameter der Funktion kopiert
- Ein Ändern dieser Werte innerhalb der Funktion hat keinen Einfluss auf den Wert der Variablen, die der Funktion übergeben worden sind.
- Call by Value macht nicht immer Sinn
    - Ist bei grossen Datenstrukturen ineffizient
    - Die Funktion soll mehr als einen Wert zurückgeben

5.16 Call by Address

Da call by Value nicht immer Sinn macht, gibt es die Möglichkeit von call by address, dabei übergibt man den Pointer dieser Variable einer Funktion.

5.16.1 Call by address vs. Call by value bei Swap-Methode

- Beispiel: Funktion `vertausche`, die die Werte zweier Variablen vertauschen soll

Call by value (funktioniert nicht):

```
int a = 3, b = 5;
vertausche(a, b);

void vertausche(int c, int d) {
  int temp = c;
  c = d;
  d = temp;
}
```

Call by address (funktioniert):

```
int a = 3, b = 5;
vertausche(&a, &b);

void vertausche(int *pc, int *pd) {
  int temp = *pc;
  *pc = *pd;
  *pd = temp;
}
```

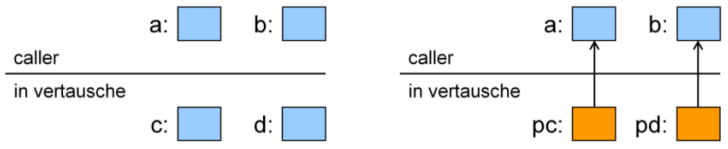
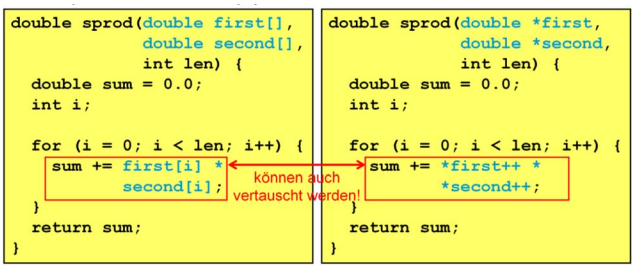


Abbildung 17 Swap-Funktion

5.17 Arrays als Parameter

- Wird ein Array in einem Ausdruck verwendet, so wird er implizit in den entsprechenden Pointer konvertiert
- Wird ein Array einer Funktion übergeben, wird implizit ein Pointer auf diesen Array übergegeben → Arrays werden immer «by address» übergeben

- Keine Roll ob vom Programmierer ein Array oder Pointer auf Array übergeben wird → für Funktion ist es immer ein Pointer
    - Rückgabewert funcName (int a[])
    - Rückgabewert funcName (int *a)




```
double sprod(double first[],
             double second[],
             int len) {
  double sum = 0.0;
  int i;

  for (i = 0; i < len; i++) {
    sum += first[i] *
           second[i];
  }
  return sum;
}
```

```
double sprod(double *first,
             double *second,
             int len) {
  double sum = 0.0;
  int i;

  for (i = 0; i < len; i++) {
    sum += *first++ *
           *second++;
  }
  return sum;
}
```

Abbildung 18 Parameteraufruf von Arrays

Der Rückgabewert einer Funktion kann **kein Array sein**, jedoch kann es ein Pointer auf ein Array sein.

5.18 Konstante Parameter

Mittels Spezifizieren eines Parameters mit const stellt der Compiler sicher, dass der entsprechende Parameter innerhalb der Funktion nicht modifiziert wird!

Beispiel: Funktion, die eine Zahl, die als String dargestellt ist, in einen int konvertiert; der String darf in der Funktion nicht verändert werden!

```
1.  /*Paramter als konstanter char-Array*/
2.  int stringToInt(const char s[]) {
3.      int i, res = 0;
4.      for (i = 0; s[i] != '\0'; i++) {
5.          res = res * 10 + s[i] - '0'; //mit '0' stellt man sicher, dass man jeweils die Zahl
    en in ASCII verwendet
6.      }
7.      return res;
8.  } /*Parameter als Pointer auf einen konstanten char*/
9.  int stringToInt(const char * s) {
10.     int res = 0;
11.     for (; s* != '\0'; s++) {
12.         res = res * 10 - '0'; //mit '0' stellt man sicher, dass man jeweils die Zahlen in A
    SCII verwendet
13.     }
14.     return res;
15. }
```

Der Gebrauch von const in Parameterliste (sofern sinnvoll) ist guter Programmierstil!

5.19 Mehr-dimensionale Arrays als Parameter

Wird ein mehr-dimensionaler Array a[3][4] einer Funktion übergeben, so muss in der Parameterliste alle ausser der ersten Dimension spezifiziert werden!

```
1.  rückgabewert funcName(int b[][4]); //i.O.
2.  rückgabewert funcName(int b[][]); //Kompilierfehler
```

Die hintere Dimensionen sind notwendig, damit in der Funktion Anweisungen wie b[2] korrekt ausgeführt werden können. → Die Angaben der zweiten Dimension ermöglicht es herauszufinden, wo die einzelnen Arrays im Array beginnen (wegen der linearen Anordnung im Speicher). Ohne die Angabe dieser zweiten Dimension wüsste der Compiler nicht, wie auf die einzelnen Arrays zugegriffen wird, denn bei der Paramterübergabe wird nur ein Pointer auf b übergeben.

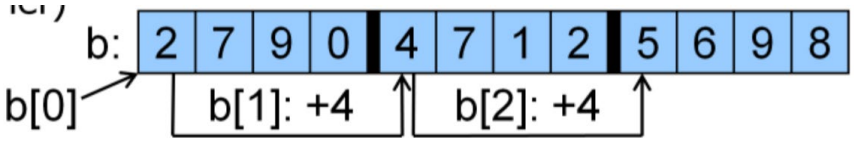


Abbildung 19 Zugriff auf mehr-Dimensionale Arrays

Bei der Konvertierung des 2-Dimensionalen Arrays a[3][4] in einen Pointer wird der Datentyp int(*)[4] (Pointer auf einen int Array der Länge 4). Aus diesem Grund sind folgende Funktionsdeklarationen äquivalent:

```
1.  rückgabewert funcName(int b[][4]);   //2D
2.  rückgabewert funcName(int(*b)[4]);   //2D äquivalent zu oben
3.  rückgabewert funcName(int b[]);  //1D
4.  rückgabewert funcName(int (*b)); //1D äquivalent zu oben
5.  rückgabewert funcName(int *b);   //1D äquivalent zu oben
```

5.20 Array of Pointers als Parameter

- Wird ein Array of Pointers int *a[10] einer Funktion übergeben, ist dies nichts anderes als die Übergabe eines 1-Dimensionalen Arrays

```
1. Rückgabewert funcName(int *b[]);
```

- Bei der Konvertierung in einen Pointer wird der Datentyp int*(*b) (Pointer auf einen Pointer auf einen int)

```
1. Rückgabewert funcName(int *(*b));
2. Rückgabewert funcName(int **b); //äquivalent zu oben
```

5.21 Funktion als Parameter

Man kann einer Funktion auch einen Pointer auf eine Funktion übergeben

Beispiel: Integration mit Trapezregel für bel. Funktionen

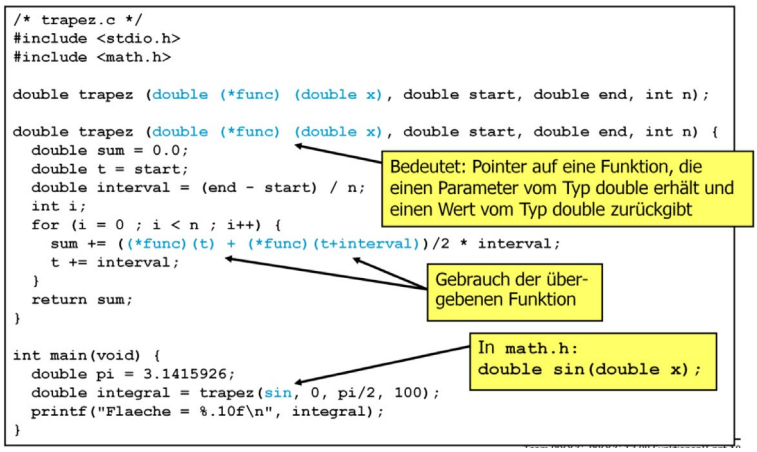


Abbildung 20 Funktion als Parameter

5.22 Funktion mit variabler Anzahl Parameter

Die Ellipse (…) zeigt an, dass noch eine beliebige Anzahl Parameter kommen können. Dabei muss die Ellipse am Ende der Paramterliste stehen. Ihr muss ein normaler Parameter vorangehen, an dem man innerhalb der Funktion erkennen kann, wie viele Parameter folgen. Eine andere Möglichkeit ist mit den va_Makros aus der Standard Library (stdarg.h); sie werden benötigt, um die Paramter zu übernehmen und auszuwerten

Beispiel: Mittelwert berechnen mit einer variablen Anzahl von int-Parametern

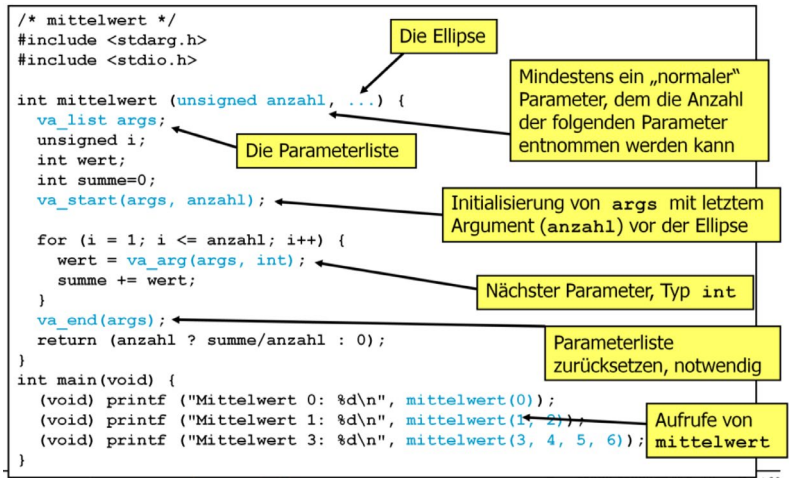


Abbildung 21 Funktion mit var. Anzahl Parameter

5.23 Kommandozeilen Parameter

Alternativ zu int main(void) als Einstiegspunkt, kann man mit

```
1. int main(int argc, char * argv[])
```

eine Kommandozeilen Parameter dem Programm übergeben werden.

Argc → beschreibt die Anzahl Parameter

Argv → ist ein Array, der die Pointer auf die Kommandozeilen Parameter (Strings) enthält

Argv[0] → ist dabei immer der Programmname

```
1. /*Dieses Programm gibt den Namen des Programms und die Kommandozeilen aus*/
2. #include < stdio.h >
3. int main(int argc, char * argv[]) {
4.  int i;
5.  for (i = 0; i < argc; i++) {
6.      (void) printf("%s\n", argv[i]);
7.  }
8. }
```

6 Build und Modulare Programmierung

Um aus dem Quellcode ein lauffähiges Programm zu generieren, sind drei Komponenten involviert:

- Präprozessor
  - Sucht nach #include, #define, #ifdef, …und ergänzt diese durch die eigentlichen Texte
- Compiler
  - Wandelt Quellcode in Objektcode
  - Kann noch offene Aufrufe enthalten (z.B. Funktionen oder globale Variablen)
- Linker
  - Verbindet die offene Aufrufe und generiert ein ausführbares Programm

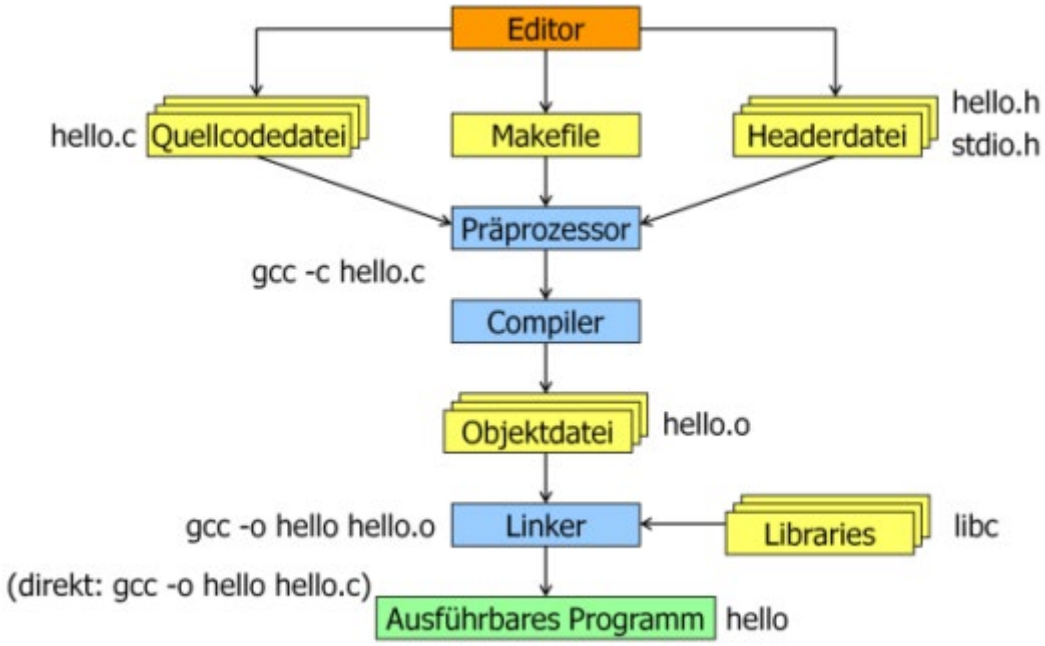


Abbildung 22 Zusammenspiel Präprozessor, Compiler und Linker

```
1. gcc -S hello.c //Kann Assembler-Code generieren
```

6.1 Präprozessor

Präprozessor-Befehle beginnen immer mit «#», welcher er anschliessend durch den effektiven Text ersetzt.

```
1. gcc -E //Output nach der Praeprozessorstufe betrachten
```

```
1. #include < Dateiname > //werden Header-Datei aus der Standard Library eingebunden
1. #include «Dateiname» //Eigene Header-Dateien einbinden
```

Konstante werden mit #define festgelegt (Flag setzen)

```
1. #define MAX 100 //Mit MAX kann jetzt auf der Wert 100 zugegriffen werden
```

Konstante-Werten werden mit #undef entfern (Flag nicht gesetzt)

```
1. #undef MAX
```

```
#if, #elif, #else, #endif, #ifdef, #ifndef //saemtliche Praeprozessor-Befehle
```

6.1.1 Include-Guard

Header-Guard ist äusserst wichtig, damit gewährleistet werden kann, dass Header-Files nur einmal eingelesen werden und somit auch die Funktion nur einmal deklariert werden.

```
#ifndef HEADERDATEI_KENNUNG
#define HEADERDATEI_KENNUNG
…
/* code */
…
#endif
```

6.2 Compiler

Der Compiler erhält den Quellcode, welcher vom Präprozessor überarbeitet wurde und generiert daraus den Objektcode. Dieser Objektcode enthält bereits Maschineninstruktionen, die aber noch nicht ausführbar sind, da er noch offene Aufrufe enthalten kann. Dabei überprüft der Compiler den Quellcode auf syntaktische Korrektheit und führt eine statische Typprüfung durch. Falls Fehler auftreten, gibt der Compiler Errors and Warnings aus.

```
test.c: 16: error: ’p’ undeclared(first use in this function) // Beispiel Error
test.c: 16. warning: assignment makes integer from point without a cast //Beispiel Warning
```

6.3 Linker

Der Linker generiert ein vollständiges, ausführbares Programm. Der Linker löst dabei die noch offenen Aufrufe auf.

6.4 Modulare Programmierung

Der Quellcode wird auf mehrere Module aufgeteilt bzw. in verschiedenen Dateien abgelegt. Dabei werden ein oder mehrere Headerdateien verwendet.

In C wäre es möglich, alles in einer grossen Datei (main.c) abzuspeichern. Jedoch setzt man hier sinnvollerweise auf die modulare Programmierung.

Variante 1:

```
/*Variante alles in einem File*/
#define MAX 1000 int doit(int b); /*Funktionsdeklaration*/
int main(void) {
    return MAX + doit(10);
}
int doit(int b) { /*Funktionsdefinition*/
    return MAX + b;
}
```

Verbesserte Variante 2 – zwei .c Files:

```
/*Aufteilung in zwei Files*/ /*main.c*/
#define MAX 1000 int doit(int b); /*Funktionsdeklaration*/
int main(void) {
    return MAX + doit(10);
}
```

```
/*doit.c*/
#define MAX 1000 int doit(int b); /*Funktionsdeklaration; optional, da doit in diesem Modul
 nicht aufgerufen wird*/
int doit(int b) { /*Funktionsdefinition*/
```

```
4.      return MAX + b;
5.  }
```

Beste Variante 3 – zwei .c Files + ein .h File:

```
1.  /*Aufteilung in zwei Files inkl. HeaderFile*/ /*header.h*/
2.  #define MAX 1000 int doit(int b); /*Funktionsdeklaration*/
```

```
1.  /*main.c*/
2.  #include "header.h" /*wichtig! hier mit "" Arbeiten, da eigenes HeaderFile*/
3.  int main(void) {
4.      return MAX + doit(10);
5.  }
```

```
1.  /*doit.c*/
2.  #include "header.h" /*wichtig! hier mit "" Arbeiten, da eigenes HeaderFile*/
3.  int doit(int b) { /*Funktionsdefinition*/
4.      return MAX + b;
5.  }
```

```
1.  gcc -Idir -o prog main.c doit.c //ausführbares Programm erzeugen – mit -I können
    Headerdateien eingebunden werden, welche nicht im aktuellen Verzeichnis liegen
```

6.5 Libraries

Die verschiedenen Standard-Headerfiles sind alles verschiedene Libraries. Man kann auch eigene Libraries generien; unter Unix mit dem Tool ar

```
1.  ar - r libown.a doit.o //kreiert Library libown.a, die doit.o enthält
```

7 Arrays und Pointer

7.1 Arrays

Ein Array in C ist ein **zusammenhängender Speicherbereich** von Elementen vom selben Typ. D.h. es sind keine Objekte, sondern eifach direkt hintereinander im Speicher abgelegt. Ein Array mit 100 int-Elementen beansprucht genau 400 Byte (sofern ein int 4 Bytes beansprucht). Die Anzahl Elemente und damit der benötigte Speicherbedarf wird zur Kompilierungszeit festgelegt.

Java	C
`int[] data = new int[100];`	`int data[100] = {0};`
`data[7] = 20;`	`data[7] = 20;`
`int b = data[200]; // exception`	`int b = data[200]; // ungeprüft`
`int c = data.length;`	`size_t c = sizeof(data)/sizeof(*data);`

Abbildung 23 Arrayvergleich Java vs. C

Die Arrayelemente erhalten bei der Deklaration einer lokalen Variablen keine **Default-Werte**, bei globalen und statischen Variablen den Wert 0. Werden bei der Initialisierung weniger Elemente angegeben, als die Arraygrösse ist, so wird der Rest mit 0 aufgefüllt.

Wichtig! Wenn keine Default-Werte vergeben werden (bspw. lokale Var), so erhalten die Variablen den Wert «der gerade im Speicher liegt» → irgendetwas

Arrays können auch als Konstante angegeben werden. Dann können die Werte nach der Deklaration nicht mehr verändert werden. Eine Initialisierung bei der Deklaration ist dabei natürlich sinnvoll, wird vom Compiler aber nicht erzwungen.

```
1.  int a[5] = {0, 1, 2, 3, 4};
2.  const int b[5]; //Funktioniert, macht aber wenig Sinn
3.  const int c[5] = {5, 6, 7, 8, 9}; //i.O.
4.  const int d[] = {10, 11, 12, 13, 14}; //i.O.
5.
6.  a[0] = 4 //i.O.
7.  b[1] = 55 //Kompilierfehler
8.  c[2] = 100 //Kompilierfehler
9.  d[3] = 33 //Kompilierfehler
```

7.1.1 Grösse berechnen

Die Grösse N des Arrays kann mit N * sizeof(ElementTyp) abgefragt werden. **Wichtig!** Hierbei erhält man nicht die Anzahl Element, sondern die Grösse des Speicherplatzes, sprich Bytes.

```
1.  sizeof(data) //Anzahl Bytes des data Arrays
2.  sizeof(*data) //Anzahl Bytes des ersten Elements des data Arrays
3.  sizeof(data) / sizeof(*data) // Anzahl Elemente des data Arrays; Anstatt *data wäre data[0]
     äquivalent
```

es gibt verschiedene Arten wie man zu seinem Ergebnis kommen kann:

```
1.  /*Zur Compile Zeit falls die Array Definition für den Compiler zugreifbar ist*/
2.  int array[100] = {0};
3.  size_t len = sizeof(array) / sizeof( * array);
4.
5.  /*Zur Compile Zeit via Konstante die vom Programmierer verwaltet wird*/
6.  #define N 100 int array[N] = {0};
7.  size_t len = N;
8.
9.  /*Zur Laufzeit über ein Sentry Element in Array -> Ansatz bei Strings */
10. int array[100] = {0, ..., -1};
11. //-1 ist hier das Sentry Element, bei Strings wäre dies beispielsweise '\0'
```

```
12. for (size_t i = 0; array[i] != -1; i++) {
13. //mach etwas mit dem Array
14. }
```

7.1.2 Zuweisung

```
1. int a[5] = {4, 7, 13, 77, 3}; //mögliche Schreibweise bei der Zuweisung beim Initialisieren

2. int b[3]; //keine explizite Initialisierung
3. b = {1, 2, 3}; //Zuweisung nach Initialisierung mit dieser Schreibweise nicht zulässig!
4. b[0] = 1; //Zulässige Zuweisung nach Initialisierung
```

7.1.3 Mehrdimensionale Arrays

Arrays können mehrere Dimensionen aufweisen, dabei ist das 2-dimensionale Array das häufigste.

```
1. int a[2][3] = {{1,2,3 },{4, 5, 6}}; //2x3 Matrix
```

a[0][0] = 1	a[0][1] = 2	a[0][2] = 3
a[1][0] = 4	a[1][1] = 5	a[1][2] = 6

Darstellung der 2x3 Matrix

Die Speicherung dieser Matrix sieht wie folgt aus:

1	2	3	4	5	6

```
char board[8][8][3];
for (int i = 0; i < LIMIT; ++i) {
  for (int j = 0; j < LIMIT; ++j) {
    int number = i;
    number++;
    switch(j) {
      case 0:
        board[i][j][0] = 'A';
        board[i][j][1] = number;
        board[i][j][2] = '\0';
        break;
      case 1:
        board[i][j][0] = 'B';
        board[i][j][1] = number;
        board[i][j][2] = '\0';
        break;
```

Abbildung 24 Drei-dimensionales Array (Schachbrett)

7.1.4 Char-Arrays und Strings

In C gibt es keinen Datentyp «String», stattdessen wird ein String als char-Array dargestellt. Jedes Zeichen entspricht einem char des Arrays (als ASCII-Wert). Das letzte Zeichen eines Strings ist ‘\0’ (NUL), dadurch kann das Ende eines Strings bestimmt werden. Dadurch braucht ein String immer AnzahlZeichen + 1 an Speicherplatz (für das ‘\0’ Element)

H	a	l	l	o	\o

➔ Entspricht 6 Zeichen (5 für Hallo + 1 ‘\0’)

Grundsätzlich gilt, dass ein char-Array nicht zwingend ein String sein muss. Erst wenn eine Folge von chars, welche auf \0 terminieren, ist es ein String!

Die Ausgabe von Strings wird mit dem Konvertierungsoperator %s, mit printf und %s kann nun ein String ausgegeben werden. %s heisst, dass printf alle Zeichen hintereinander ausgeben soll bis ein \0-Zeichen erkannt wird.

```
/* String */
char name[20];
        name = "hooooi"; // --> ERROR
        name[0] = 'H';
        name[5] = 'o';
        name[1] = 'y';
        printf("%p\n", name); // prints address from the first element
        printf("%s\n", name); // prints "Hy" because it's searching till '\0'
char name[20] = "Pascal"; // \0 added automatically
        printf("%s", name); // prints "Pascal"
char name[20];
        strcpy(name, "Pascal");
        printf("%s\n", name); // prints "Pascal"
        return name; // Returns a pointer to the first element

int * returnArray() // Function who returns a pointer
        char name[] = "Colin T";
        int length = (sizeof(name) / sizeof(*name)); // included \0 and spaces→ 8!
        if(array[7] == '\0') {…} → True!
        return name;
}

void getArray(int * array, int length) {
    int i;
    for (i = 0; i < length - 1; i++) {  // length == 8 → last i == 6 (-1 nur wegen String, bei int nicht!!)
        printf("%c\n", * (array + i)); // Equivalent to array[i]
    }
}
char a = '1', b[] = "1";
printf("%d %d", (int)sizeof(a), (int)sizeof(b)); // prints: 1 2
```

7.1.5 Wichtige String-Funktionen

#include <string.h> bindet String Funktionen in Std. Library ein.

- Int strlen (const char s[]) → Länge eines Strings (ohne abschliessendes \0)
- Int strcmp (const char s1[], const char s2[]) → vergleicht zwei Strings und gibt <0, 0, >0 aus.
- Char* strcpy(char dest[], const char source[](→ kopiert String von source nach destination
- Char* strcat (char s1[], const char s2[]) → Zusammenhängen zweier Strings s2 wird s1 angehängt

Wichtig! Alle Funktionen bearbeiten die Strings jeweils so lange, bis sie das \0-Zeichen erkennen.

7.2 Pointer

Ein Pointer ist eine Variable, welche eine Adresse enthält. Da Pointer eine Adresse enthalten, sind sie auf Computer mit 32-bit Architektur typischerweise 4Bytes gross. Pointers haben keinen Default-Value! Die Adresse 0 ist niemals eine gültige Adresse und kann deshalb als Sentry verwendet werden

Wichtig! Adressen werden immer in Hexadezimaler Schreibweise abgespeichert → auf 69 folgt 6A

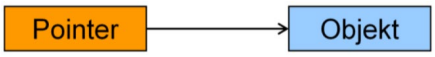


Abbildung 25 Ein Pointer zeigt auf ein Objekt

7.2.1 Deklaration von Pointers

```
/*Deklaration von Pointers*/
int *p; /*p ist ein Pointer auf ein Objekt vom Typ int */
int *p; /*Eine andere Schreibweise für das Gleiche*/
double *d[20]; /*d ist ein Array von 20 Pointern auf Objekte vom Typ double*/
double(*d)[20]; /*d ist ein Pointer auf ein double Array mit 20 Elementen*/
char **ppc; /*Pointer auf einen Pointer auf ein Objekt vom Typ char*/
int *p, q; /*p ist ein Pointer auf ein Objekt vom Typ int, q ist eine Variable Typ int */
int *p, *q; /*p und q sind Pointer auf ein Objekt vom Typ int*/

//Pointer und const
double *cdp; /*Pointer auf double *cdp = 5.0 und cdp++ möglich*/
double *const cdp; /*konstanter Pointer, *cdp = 5.0 möglich, cdp++ nicht möglich*/
const double *dp; /*Pointer auf Konstante, dp++ möglich und *dp = 5.0 nicht möglich */
const double *const dp; /*Konstanter Pointer auf Konstante, dp++ und dp = 5.0 nicht
möglich*/
```

Es gibt auch die Möglichkeit, dass ein **Pointer void** ist. Dies wird häufig verwendet, wenn man einen Pointer auf einen Block von Daten zu referenzieren, ohne zu wissen, was in den Daten steht.

Bei Pointer arbeitet man mit zwei Operatoren:

- Der Referenzoperator oder Addressoperator **&** liefert die Adresse eines Objektes
- Der Dereferenzoperator oder Zugriffsoperator ***** kann auf Pointer angewendet werden und liefert die Daten des Objekts, auf welches der Pointer zeigt

7.2.2 Zuweisung eines Pointers

```
int i; /*eine Variable vom Typ int*/
int *ip; /*ein Pointer auf int*/
ip = & i; /*Zuweisung der Adresse von i; ip zeigt jetzt auf i */
*ip = 3; /*ip wird dereferenziert -
dem Objekt wird 3 zugewiesen; die Variable i ist jetzt als 3 */
```

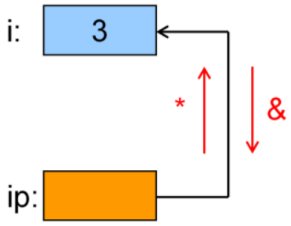


Abbildung 26 Funktionweise von Adress- und Zugriffsoperatoren

```
int var = 0;
int array[] = { 1, 2, 3, 4, 5 };
int *ptr;
ptr = NULL; // to NULL ( points to nothing, point to 0 (invalid)
ptr = &var; // assign to an address
ptr = array; // directly to an array
ptr = getPtr(); // from a function
```

7.2.3 Dereferenzieren von Pointers

```
int *ptr = &x; // ptr points to a integer value
*ptr = 50; // set the value of x to 50

int *ptr = array;
ptr[i] = 50; // [] dereferences the pointer, set the value from the i-element to 50

SomeStruct *ptr = &someStruct;
ptr->x = 50; // Set the struct property to 50
```

7.2.4 Value vs. Reference

```
void giveMeThings (int val, int *valRef, SomeStruct value, SomeStruct *ref, int arrayRef[], int *arrayRef2) {
// arrayRef[] and *arrayRef2 has the same type
// val, value are copied to the Function
// *ref, *valRef and arrays are passed as references
}
```

Abbildung 27 Value vs. References

7.3 Lesen von C Deklarationen

```
1.  int a; //declare a as int
2.  int (a); //declare a as int
3.  int a[]; //declare a as array of int
4.  int a[2]; //declare a as array 2 of int
5.  int *a[]; //declare a as array of pointer to int
6.  int (*a)[]; //declare a as pointer to array of int
7.  int a[2][3]; //declare a as array 2 of array 3 of int
8.  const int(*const a)[3] //declare a as const pointer to array 3 of const int
9.  int f(void); //declare f as function (void) returning int
10. int *f(int); //declare f as function (int) returning pointer to int
11. int (*f)(int); //declare f as pointer to function (int) returning int
```

- Gruppierung bindet am stärksten

```
int (*a)[2]; // declare a as pointer to ...
```

- Array Index und Funktionsparameter binden am zweitstärksten

```
int *a[2];     // declare a as array 2 of ...
int *f(void); // declare f as function (void) returning ...
```

- Pointer und const binden am drittstärksten

```
int * const a[2]; // declare a as array 2 of const pointer to ...
```

- Der Typ bindet am schwächsten

```
int * const a[2]; // declare a as array 2 of const pointer to int
```

- Der Typ kann nicht gruppiert werden, der Rest schon

```
int (* const a[2]); // declare a as array 2 of const pointer to int
```

```
// declare f as pointer to function (void) returning pointer to char
char *(*f)(void)
// declare tab as array 10 of pointer to function (int) returning int
int (*tab[10])(int)

// declare wrap as function (pointer to function (int) returning void)
// returning void


// declare matrix as array 5 of array 5 of double


// declare argv as array of pointer to char


// declare p as pointer to array 5 of array 7 of char


// declare a as array of array 5 of array 7 of char
```

1) char * (*f)(void);

2) int (*(tab[10]))(int);

3) void wrap(void (*)(int));

4) double matrix[5][5];

5) char * argv[];

6) char (*p)[5][7];

7) char a[5][7];

Regeln:

- Gruppierungen (*a) bindet am stärksten
- ArrayIndex *a[2] und Funktionsparameter *f(void) binden am zweitstärksten
- Pointer und const *const binden am drittstärksten
- Der Typ int bindet am schwächsten
- Der Typ kann nicht gruppiert werden, der Rest schon

➔ Rote Farbe sind Beispiele

7.4 Zusammenhang Arrays und Pointer

Der Name eines Arrays stellt die fixe **Startadresse** des Arrays dar, diese Adresse kann nicht verändert werden. Wird ein Array in einem Ausdruck verwendet, so wird der vom Compiler immer impliziert in den entsprechenden Pointer auf das erste Element des Arrays konvertiert.

Speicheradresse	*3072*	*3076*	*3080*	*3084*
Value	2	4	6	8

In C wird ein Array immer linear im Speicher (sprich hintereinander) abgespeichert. Mit Pointer können auf die einzelnen Elemente zugegriffen werden → Pointerarithmetik.

7.5 Pointerarithmetik

Neben * und & sind noch weitere Operatoren auf Pointer anwendbar (==, !=, <, >, <=, >=, +, -, ++, --, +=, -=)

Zeigt der Pointer pa auf das erste Element, so zeigt (pa + i) auf das i-te Element

Pointer	*pa*	*pa + 1*	*pa + 2*	*pa + 3*
Value	2	4	6	8

a[2] == *(pa+2) == *(a+2) == pa[2] → Alle Ausdrücke sind äquivalent

Die Adressen einzelner Elemente können im Array auch mit dem Referenzoperator direkt einem Pointer zugewiesen werden

```
1. int a[5] = {2, 3, 6, 8, 10};
2. int *pa = & a[2];
3. *(pa + 2) = 13; /* a[4] ist jetzt = 13 */
4. *(pa - 2) = 20; /* a[0] ist jetzt = 20 */
```

Folgerung: sizeof spielt effektiv eine Rolle im Hintergrund. Würde statt int* z.B. double* verwendet werden, so wirkt sich eine Operation +1 auf den Pointer effektiv um +8 aus (Annahme: ein double braucht 8 Byte Speicherplatz). Bei char* ist es effektig ein +1 auf der Adresse, da ein char 1 Byte beansprucht.

7.5.1 String Literals und Pointer

Man kann Arrays und Pointer mit einem String Literal initialisieren:

```
1. char a[] = "hello, world";
2. char *pa = "hello, world";
```

Dabei sind die beiden Ausdrücke sehr ähnlich und doch gibt es einen wichtigen Unterschied:

- **a** ist ein **char-Array** und stellt eine **fixe Startadresse** dar → a kann nicht auf einen anderen Speicherbereich zeigen
- **pa** ist eine **Pointervariable**, die die **Startadresse des char-Arrays** enthält → kann später auch andere Werte enthalten

7.5.2 Mehrdimensionale Arrays und Pointers

Es gilt weiterhin der Name des Arrays stellt die fixe Startadresse des Arrays dar, diese Adresse kann nicht verändert werden. Wird ein Array in einem Ausdruck verwendet, so wird er implizit in den Pointer auf das erste Element (der **ersten Dimension**) konvertiert.

```
1. //1D
2. double values[10] = {0.0, 5.0};
3. double(*pValues) = values; //Pointer to double
4. //2D
5. int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
6. int(*pMatrix)[3] = matrix //Pointer to array 3 of int
7. //3D
8. char board[8][8][3] = {0};
9. char(*pBoard)[8][3] = board //Pointer to array 8 of array 3 of char
```

Die Pointerarithmetik funktioniert analog zur ein-Dimension

```
1. int p[10] = {0}; /*p[2] == *(p+2)*/
2.
```

```
3.  int q[5][8]; /*q[2][3] == (q[2])[3] == *(*(q+2)+3)*/
```

7.5.3 Jagged-Arrays

Jagged Arrays sind ein-dimensionale Arrays von Pointern. Die Element Pointer zeigen dann auf andere Speicherbereiche wo die «nächste» Arrays Dimension abgelegt ist.

```
1.  char * jagged[] = {
2.           "Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag", "Samstag", "Sonntag"
3.      }
4.  ...
5.  if(jagged[3][9] == 'g') {...}
6.  //jagged[3] ist Donnerstag
7.  //jagged[3][9] ist der letzt Character von Donnerstag
```

7.5.4 Struct und Pointer

Oft verwendet man Pointer auf Strukturen. Dabei gibt es eine abgekürzte Schreibweise (**->**), um auf Elemente in der Struktur zuzugreifen

```
1.  struct student { /*Deklaration struct Student*/
2.      int legiNr;
3.      char name[30];
4.  }
5.  struct student *sp, s; /*sp ist Pointer auf eine Strutktur Student*/
6.  s.legiNr = 999; /*Initialisierung von s*/
7.  (void) strcpy(s.name, "Mueller");
8.  sp = & s; /*sp zeigt auf Struktur s*/
9.  (void) printf("Student: %s, Nr: %d", (*sp).name, (*sp).legiNr);
10. /*bessere Lesbarkeit*/
11. (void) printf("Student: %s, Nr: %d", sp - > name, sp - > legiNr);
```

7.6 Häufige Fehler mit Pointers

```
1.  int *ptr = NULL; // Not possible to dereference null
2.  *ptr = 0; // crash
3.  ptr[i] = 0; // crash
4.
5.  Man *man; // Not possible to dereference uninitialized 'wild' pointers
6.  man-> x = 50; // crash
7.  man-> name = NULL; // crash
8.
9.  Man men[10]; // Not dereference outside the bounds of an array
10. men[500].x = 50; // crash (because men is an array '->' not necessary
11. men[10].x = 50; // crash
12.
13. char *str1 = "a string"; // not modify constants
14. char str2[1024] = "a string";
15. str1[0] = 'b'; // crash
16. str2[0] = 'b'; // ok (Will copy the string)
```

8 Dynamische Allozierung

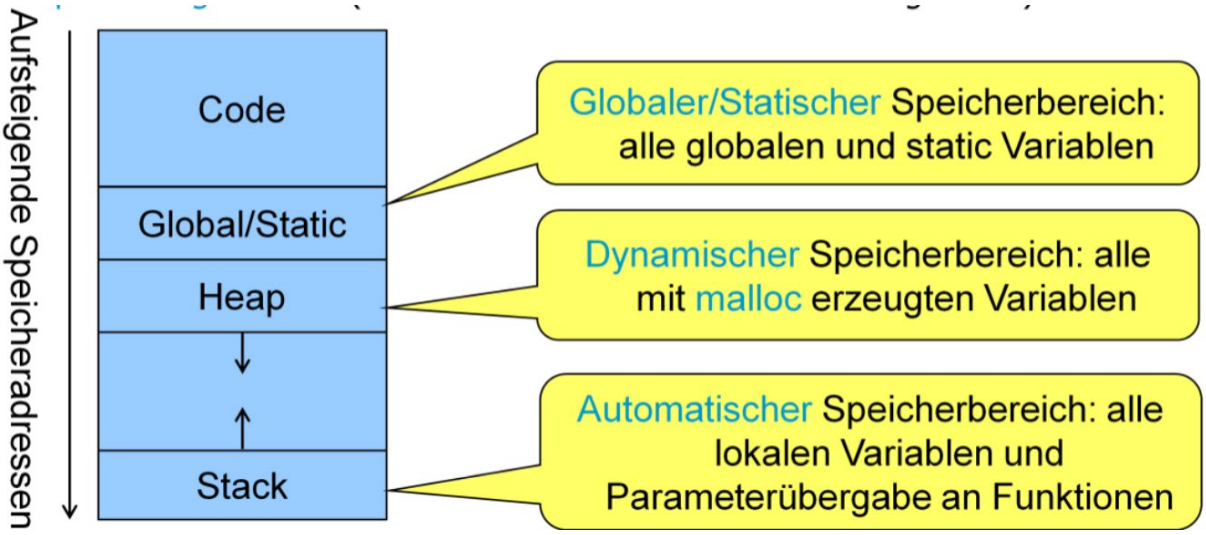


Abbildung 28 Speicherorganisation aus Sicht eines ausführenden Programms

8.1 Stack (automatischer Speicher)

Der Stack hat eine einfache Struktur nach dem LIFO (last-in-first-out) Prinzip. Speicheradressen verändert sich dauernd und kann auch überlaufen. Grosse Datenmengen sollten nicht auf dem Stack definiert werden.

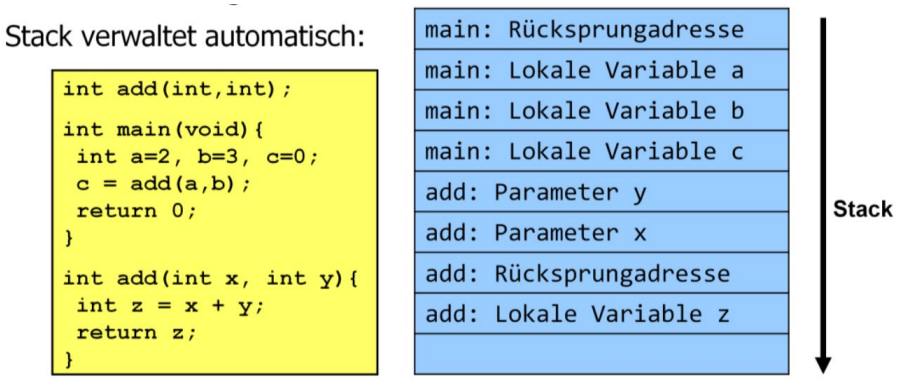


Abbildung 29 Verwaltung von Stack

8.2 HEAP (Dynamischer Speicher)

- Dynamischer Speicher kann man mit malloc() erzeugen
- Die Grösse des HEAPS ist dynamisch (wächst nach Bedarf)
- HEAP-Memory ist langsamer als STACK-Memory
- Einzige Einschränkung der Grösse durch Hardware
- Keine garantierte Effizienz der Speicherausnützung → Fragementierung, dies bedeutet, dass im einzelnen zwar noch Speicherplatz vorhanden ist, dieser aber so verstückelt ist, dass keine brauchbaren Daten mehr abgespeichert werden können
- Funktionen:
    - Malloc → alloziert Speicher dynamisch (Grösse angeben)
    - Calloc → alloziert und **initialisiert** Speicher dynamisch
    - Realloc → vergrössert / verkleinert einen dynamischen Speicherbereich

8.2.1 Dynamischer Speicher allozieren – malloc()

Der Speicher wird zur Laufzeit auf dem Heap mit malloc (stdlib.h) erzeugt. Dies ist vor allem hilfreich, wenn zur Kompilierzeit noch nicht bekannt ist, wie viel Speicher benötigt wird.

Malloc macht zwei Dinge:

- Alloziert Speicher der Grösse size Bytes auf dem Heap
- Gibt die Adresse (also einen Pointer) dieses Speicherplatzes zurück (void *)

Wichtig! Es soll jeweils überprüft werden, ob der Speicher auch entsprechend alloziert werden konnte. Falls kein Speicher alloziert wurde, so wird NULL zurückgegeben.

```
1. int *ip;
2. ip = malloc(sizeof(int)); //fordert Speicher an
3. if (ip) { //überprüft, ob erfolgreich Speicher alloziert wurde
4.     *ip = 5;
5. }
```

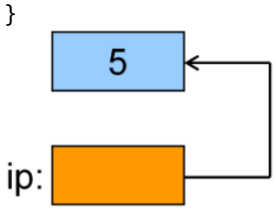


Abbildung 30 allozierter Speicherzuweisung

Ebenfalls wichtig ist, dass man den Pointer auf den Speicherbereich nicht verliert. Denn wenn dieser nicht mehr bekannt ist, kann der Speicher auch nicht mehr freigegeben werden.

```
1. int *ip;
2. ip = malloc(sizeof(int)); //Speicherbereich 1 fordert Speicher an
3. ip = malloc(sizeof(int)); //Speicherbereich 2 fordert Speicher an, der Pointer auf 1 geht v
   erloren!
4. if (ip) { //überprüft, ob erfolgreich Speicher alloziert wurde
5.     *ip = 5;
6. }
```

Natürlich kann man auch Speicherplatz für ein Array anfordern.

```
1. int * ap;
2. int n = 250;
3. ap = malloc(n * sizeof(int));
```

In diesem Beispiel werden nun 1000 Bytes alloziert (angenommen sizeof(int) = 4Bytes. Diese 1000 Bytes liegen nun **hintereinander** auf dem Heap

8.2.2 Dynamischer Speicher freigeben – free()

Im Gegensatzu zu Java, hat C keinen Garbage Collector und somit ist der Entwickler selbst für den Speicher verantwortlich. Wenn ein gewisser Speicherplatz nicht mehr benötigt wird, muss er diesen wieder mit free() freigeben. Mit dieser Freigabe werden die Daten auf dem Speicher nicht gelöscht, sondern man sagt dem Betriebssystem, dass diese Daten überschrieben werden dürfen. Dies ist vergleichbar, wenn ein Dozent eine Wandtafel dem nächsten Dozenten übergibt. Der neue Dozent kann selbst entscheiden, ob er die Daten auf der Wandtafel überschreiben möchte oder ob er dies stehen lassen will.

➔ Ein Unterlassen der Freigabe, kann langfrist das Programm zum Absturz bringen (Memory Leaks)

```
1. int * ip;
```

```
ip = malloc(sizeof(int)); //fordert Speicher an
if (ip) { //überprüft, ob erfolgreich Speicher alloziert wurde
    * ip = 5;
}
...
free(ip) //der allozierte Speicher wird wieder dem System freigegeben
```

Sämtlicher Speicher wird freigegeben, sobald das Programm terminiert.

8.3 Stack Overflow (sicheres Programmieren)

Als Programmieren muss man absichern, dass jeweils auch nur das eingelesen wird wo man möchte, bzw. nicht das etwas zu viel überschrieben wird.

- Buffer Overflow = zu wenig Platz für die Daten → Sicherstellen, dass der Buffer jeweils geleert wird
- Missachten der Buffer-Grenzen (Bsp. Array)
- Zu wenig reservierter Speicherplatz für die geplante Datenmenge
- Keine integrierten Mechanismen für Out-Of-Range → Danach werden einfach die Speicherplätze überschrieben
- Der Programmablauf kann beeinträchtigt werden

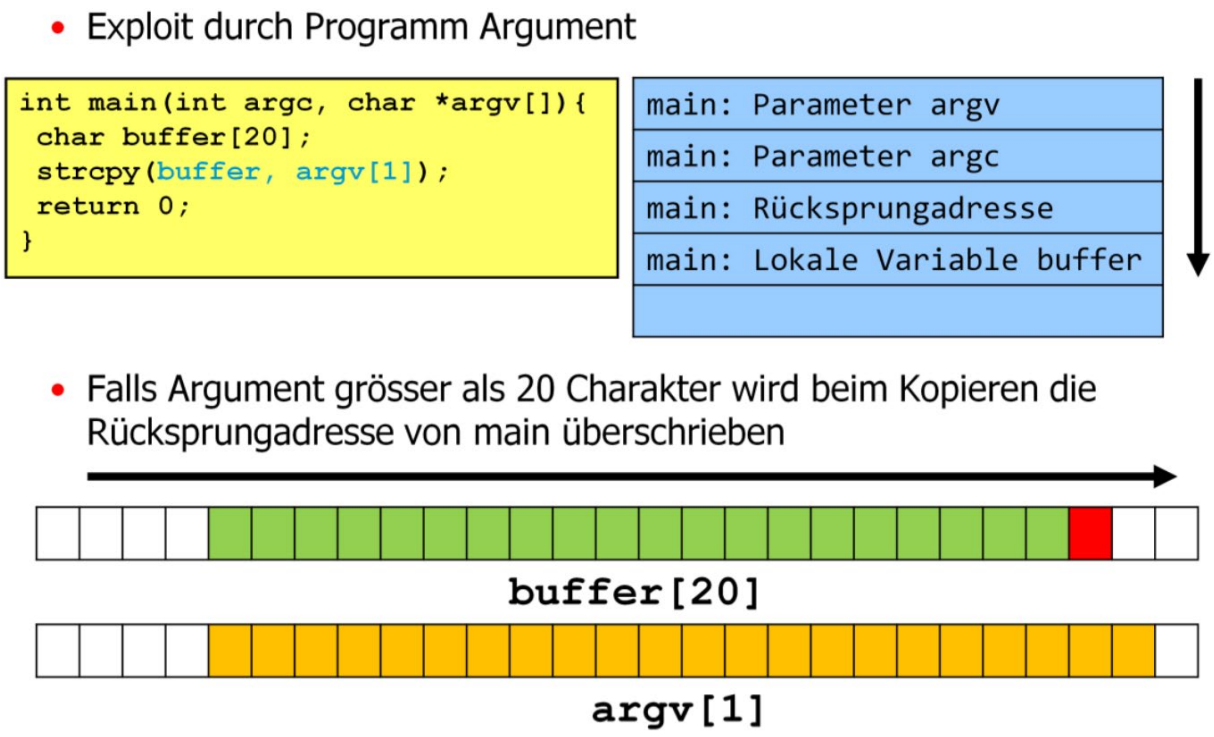


Abbildung 31 Stack-Overflow (typisches Beispiel)

Massnahmen:

- man sollte keine unsicheren Funktionen verwenden

unsicher	besser
gets	fgets (liest immer \n, informiert falls buffer zu klein)
strcpy / strcat	strncpy / strncat (n = Anzahl Zeichen)
sprintf / scanf	immer mit Precision Specifiers verwenden (z.B. "%.100s")

Abbildung 32 sichere Funktionen verwenden

- Benutzereingaben prüfen

- Testen mit Tools um Speicher-Leaks aufzuzeigen

8.4 Heap-Overflow (sicheres Programmieren)

- Ursachen oft identisch zu Stack-Overflow
- Heap-Overflow erfolgt durch die Verwendung von malloc, calloc und realloc
- Dynamische Allokierung kann Kontrollstrukturen für free() überschreiben
- Free() ist nicht manipulationssicher, sicherstellen dass free() nur einmalig aufgerufen wird

```
int main(int argc, char *argv[]){
 char* a = malloc((strlen(argv[1]) * sizeof(char));
```

unsicher	**besser**
malloc, calloc, realloc	Längenprüfung bei Benutzerangaben
free	nur ein mal verwenden

Abbildung 33 Heap-Overflow

9 Code Beispiele

9.1 Address as a parameter

```
1.  void calculateCircleArea(int *radius) { // Address as a parameter
2.          int area = ((*radius) * (*radius) * PI); //Dereference parameter
3.      }
```

9.2 Swap two pointers

```
1.  // Swap the memory place from two pointers
2.  void swap(void * a, void * b, size_t bytes) {
3.      char * left = a;
4.      char * right = b;
5.      char tmp;
6.      if (bytes == 1) {
7.          tmp = * left; // Dereference left
8.          * left = * right; // Swap pointer
9.          * right = tmp; // Save dereferenced left to dereferenced right
10.     } else {
11.         for (int i = 0; i < bytes; i++) {
12.             swap( & left[i], & right[i], 1);
13.         }
14.     }
15. }
```

9.3 Inverse / Reverse

```
void inverse(char * const pc) {
  int i = 0, len = 0, temp = 0;
  for(i = 0; *(pc + i); i++, len++);
  for(i = 0; i < len/2; i++) {
    temp = *(pc + i);
    *(pc + i) = *(pc + len - i - 1);
    *(pc + len - i - 1) = temp;
  }
```

9.4 Bubble sort

```
1.  // Bubble sort algorithm
2.  void sortList(char * wordlist[], int listLength) {
3.      int i, j;
4.      char * temp;
5.      for (i = 0; i < listLength - 1; i++) { // -1 because counter starts at 0
6.          for (j = 0; j < listLength - i - 1; j++) {
7.              if (strcmp(wordlist[j], wordlist[j + 1]) > 0) {
8.  // strmp gives back greater than 0 when word1 is longer than word2 // if yes, swap them
9.                  temp = wordlist[j];
10.                 wordlist[j] = wordlist[j + 1];
11.                 wordlist[j + 1] = temp;
12.             }
13.         }
14.     }
15. }
```

9.5 Rekursiver Aufruf

```
1.  int fakultaet(int n) {
2.      if (n < 2) {
3.          return 1;
4.      } else {
5.          return n * fakultaet(n - 1); // Rekursiver Aufruf
6.      }
7.  }
```

9.6 CompareTo

```
/* Compare to */
int compareTo(Person p1, Person p2) {
    int result;
    result = strcmp(p1.lastname, p2.lastname);
       // == 0 if p1.lastname == p2.lastname
       // == <0 if ASCII p1.lastname < ASCII p2.lastname
       // == >0 if ASCII p1.lastname > ASCII p2.lastname
    if (result != 0) { // --> not the same String
        return result; // any number != 0
    }
    result = strcmp(p1.firstname, p2.firstname); // Executed in case p1.lastname == p2.last
name
    if (result != 0) { // --> not the same String
        return result;
    }
    result = p1.age - p2.age; // Executed in case p1.firstname == p2.firstname
    return result; // Difference between the two ages
}
```

9.7 Summe von int Array

```
int sumOfArray(const int * array, int arrayLength) {
    int sum = 0;
    for (int i = 0; i < arrayLength; i++) {
        sum += *(array + i);
    }
    return sum;
}
int main() {
    int superArray[] = { 1, 5, 8, 10, 14 };
    int length = (sizeof(superArray) / sizeof(*superArray));  // == 5
    printf("%d (arraylength = %d)\n", sumOfArray(superArray, length), length);
    return 0;
}
```

Oder:

```
int sumOfArray(const int *array, int arrayLength) {
    int sum = 0;
    const int *p;
    for (p = array; p < (array + arrayLength; p++) {
        sum += *p;
    }
    return sum;
}
```

9.8 String auf heap kopieren

```
#include < stdlib.h >
#include < string.h >
char *copyStringToHeap(const char *sourceString) {
    int length;
    char *stringOnHeap;
    length = strlen(sourceString);
    stringOnHeap = (char *)malloc((length + 1) * sizeof(char));
    if (NULL == stringOnHeap) {
        exit(1);
    }
    return strncpy(stringOnHeap, sourceString, length);
}
```

10 Anhang

10.1 FAQ

10.2 Beispielaufgaben

```
6   void printStdout(void (*printOut)(int i), int outVal);
```

- Funktionsdeklaration der Funktion printStdout
- Rückgabewert void
- Parameter 1: Ein Pointer auf eine Funktion mit dem Parameter int i und einem Rückgabewert void
- Parameter 2: outVal vom typ Integer

```
8   static int counter;
```

- Definition einer globalen Variable
- Statisch, das heisst sie ist nur innerhalb des Files sichtbar

```
11      const int test[] = {9,8,7,6,5,4};
```

- Allozieren von Speicherplatz auf dem Stack für 6 konstante ints welche mit Werten initialisiert werden

10.2.1 Pointers-Value berechnen

```
/* pointers.c */
int main(void) {
  int *ap, *bp;
  int c = 5, d;

  ap = &c;          /*   ap = 2289596      *ap = 5          */
  c++;              /*    c = 6            *ap = 6          */
  *ap = -10;        /*    c = -10          *ap = -10        */
  bp = &c;
  c = 15;           /*  *ap = 15           *bp = 15         */
  *bp = *ap / 2;    /*    c = 7                             */

  ap = &d;          /*  *ap = 232482034    *bp = 7          */
  d = 3;            /*  *ap = 3            *bp = 7          */
}
```

10.2.2 Berechnungen mit Arrays und Jagged-Arrays

```c
char days[7][10] = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};
// unchecked array
int array[7][10] = {1,2,3,4,5,6,7};
char *pdays[7] =  {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};
// checked array
(void)printf("%ld %ld\n", sizeof(days), sizeof(pdays));
// sizeof --> anzahl Byte. (7*10=70), (7*8=56) (8 weil Pointer in 64 Bit umgebung)
(void)printf("%ld\n", sizeof(array));
// --> (70*4 = 280) wegen int
(void)printf("%ld %ld\n", sizeof(days+1), sizeof(pdays+1));
// --> gibt 2x 8 aus weil bei beiden die grösse eines Pointers abgefragt wird
(void)printf("%ld %ld\n", sizeof(days[1]), sizeof(pdays[1]));
// days[1] zeigt auf Tuesday mit länge 10 --> 10, zweites: 8 weil pointer
(void)printf("%s %s\n", days[4], pdays[4]);
// beides gibt Friday aus, %p beim zweiten gibt die Adresse
(void)printf("%d\n", days[4] - days[1]);
// 4 - 1 = 3 --> 3*10 --> 30
(void)printf("%d\n", pdays[4] - pdays[1]);

(void)printf("%d\n", &days[2][3] - &days[0][0]);
// Adresse von n - Adresse von M = 23
(void)printf("%d\n", &pdays[2][3] - &pdays[0][0]);
// Adresse von n - Adresse von M = 23
```

10.3 Make

Make generiert ein Programm aus mehreren Modulen. Dabei werden nur diese Dinge neu gebildet die nicht aktuell sind.

10.3.1 Makefile

Enthält Regeln wie und was auszuführen ist. Eine Regel hat folgende Form:

```
1. target: dependecies
2. <TAB> command
```

Target = was zu erstellen ist

Dependencies = Wovon das target abhängig ist

Command = Befehle die immer dann abgearbeitet werden wenn eine der dependencies ein jüngeres Datum haben als das target.

<TAB> Ist notwendig

- Beispiel: Programm `rechner` aus Modulen: `main.c`, `add.c`, `sub.c`, `mul.c` und `div.c`
- Alle Quellcodedateien nutzen die gemeinsame Headerdatei `def.h`
- Entsprechendes Makefile (Aufruf: `make [all]`, `make rechner` oder `make clean`):

```
# rechner wird aus allen Objektdateien (.o) zusammengebaut; clean dient
# dazu, alle Objektdateien zu entfernen;
all: rechner
rechner: add.o sub.o mul.o div.o main.o
        gcc -o rechner add.o sub.o mul.o div.o main.o
clean:
        rm -f *.o rechner

# so entstehen die Objektdateien (def.h und Makefile werden angegeben, um
# auch bei Veraenderungen dieser Dateien die Kompilierung neu zu starten)
add.o: add.c def.h Makefile
        gcc -c add.c
sub.o: sub.c def.h Makefile
        gcc -c sub.c
mul.o: mul.c def.h Makefile
        gcc -c mul.c
div.o: div.c def.h Makefile
        gcc -c div.c
main.o: main.c def.h Makefile
        gcc -c main.c
```

- Der Einstiegspunkt des Makefiles ist das target all, dort steht, all hängt von rechner ab.
- Man springt zum target rechner, rechner hängt von add.o, sub.o etc. ab
- Man springt zum target add.o, add.o hängt von add.c, def.h, Makefile ab. Ist eines davon jünger als add.o, so wird add.o gemäss gcc –c add.c kompiliert
- Dasselbe geschieht mit sub.o, mul.o etc.
- Man kehrt wieder zum target rechner zurück. Wurde eine der add.o, sub.o etc. neu kompiliert, so wird gcc –o rechner add.o... ausgeführt
- Man kehrt zu all zurück, dort gibt es kein command, womit das Makefile verlassen und make beendet wird
- .PHONY wird für die targets verwendet die kein File produzieren sollen (default, all, clean, test). Wenn .PHONY nicht definiert wären, würde beim Befehl make clean ein File mit dem Namen clean.o erstellt.
- Das make tool schreibt jedes Kommando das ausgeführt wird auf die Konsole. Wenn ein Kommando nicht vor dem Ausführen ausgegeben werden soll, wird ein @ hinzugefügt.

```
#Target to produce
TARGET        := bin/convert

# implementation files
SOURCES       := src/main.c src/calculate.c src/countWords.c

# test implementations
TSTSOURCES    := tests/test.c

# directories to create (and remove upon cleanup)
CREATEDIRS    := bin doc

# list of derived file names from the source names
OBJECTS       := $(SOURCES:%.c=%.o)    # list of gcc -c  ... produced *.o files
DEPS          := $(SOURCES:%.c=%.d)    # list of gcc -MD ... produced *.d files
TSTOBJECTS    := $(TSTSOURCES:%.c=%.o) # list of gcc -c  ... produced *.o files
TSTDEPS       := $(TSTSOURCES:%.c=%.d) # list of gcc -MD ... produced *.d files
TSTTARGET     := $(CURDIR)/tests/runtest

# libraries
CUNITINCDIR   := $(CURDIR)/../../../CUnit/include
CUNITLIBDIR   := $(CURDIR)/../../../CUnit/lib
TSTINCDIR     := $(CURDIR)/../../../include
TSTLIBDIR     := $(CURDIR)/../../../lib

# full path to the target
FULLTARGET    := $(CURDIR)/$(TARGET)

# commands and flags
CC            = gcc
CFLAGS        = -std=c99 -Wall -g
CPPFLAGS      = -MD -Isrc -Itests -I$(TSTINCDIR) -I$(CUNITINCDIR) -DTARGET=$(FULLTARGET)
LDFLAGS       = -static -z muldefs
```

$(CURDIR) = Aktueller Pfad.

```
# targets which get always visited (without checking any up-to-date state)
.PHONY: default clean test doc mkdir

# targets
default: $(FULLTARGET)
	@echo "#### $< built ####"

$(FULLTARGET): mkdir $(OBJECTS) Makefile
	$(LINK.c) -o $@ $(OBJECTS)

clean:
	$(RM) $(TARGET) $(OBJECTS) $(DEPS) $(TSTTARGET) $(TSTOBJECTS) $(TSTDEPS) $(wildcard */*~ *~ tests/*.txt)
	$(RM) -r $(CREATEDIRS)
	@echo "#### $@ done ####"

doc:
	doxygen ../Doxyfile > /dev/null
	@echo "#### $@ done ####"

test: $(TSTTARGET)
	(cd tests; $(TSTTARGET))
	@echo "#### $< executed ####"

$(TSTTARGET): $(FULLTARGET) $(TSTOBJECTS)
	$(LINK.c) -o $(TSTTARGET) $(TSTOBJECTS) $(OBJECTS) -L$(CUNITLIBDIR) -lcunit -L$(TSTLIBDIR) -lprogctest
	@echo "#### $@ built ####"


# create needed directories (ignoring any error)
mkdir:
	-mkdir -p $(CREATEDIRS)

# read in the gcc -MD ... produced dependencies (ignoring any error)
-include $(DEPS) $(TSTDEPS)
```

10.4 Unit testing

Zusichern von Preconditions

Wenn im Code gewisse Vorbedingungen wahr sein müssen (z.B. eine Parametereingabe die das Programm abstürzen lassen würde) kann man durch Precondition das Programm an einer definierten Stelle abstürzen lassen (antstatt unkontrolliert)

→ #include <assert.h>

```
1. #include < assert.h >
2. void swap_int(int *a, int *b) {
3.     assert(a); // programming error if a == null
4.     assert(b): // programming error if b == null
5. }
```

Assert violations können durch den Debugger analysiert werden. Dazu müssen core-dumps eingeschaltet sein.

Unit testing

Unit tests testen einzelne Funktionen oder Module. Durch testen kann man die Korrektheit eines Programmes nicht beweisen, aber es dient dazu Fehler früh zu finden und später bei Änderungen oder Erweiterungen die Funktionalität bei zu behalten.

Test driven development

- Definition Test sowie ein Stub das mit dem Test bricht.
- Implementieren einer Funktion bis der Test erfolgreich durchlaufen wird.

CUnit

```c
#include "CUnit/Basic.h" //Cunit file
#include "test_utils.h" //Cunit file
#include "list.h" // Header file from the programm
#include "person.h" // Header file from the programm

#ifndef TARGET // must be given by the make file --> see test target
#error missing TARGET define
#endif

/// @brief The name of the STDOUT text file.
#define OUTFILE "stdout.txt"
/// @brief The name of the STDERR text file.
#define ERRFILE "stderr.txt"

static Person testperson1 = {"Tesla", "Nikola", 86};
static Person testperson2 = {"Edison", "Thomas Alva", 84};
static Person testperson3 = {"Tesla", "Nikola", 86};

// for output testing see SHOW_USECASE
#define SHOW_USECASE "show.input" ///< Input file for show usecase

// setup & cleanup
static int setup(void) {
        remove_file_if_exists(OUTFILE);
        remove_file_if_exists(ERRFILE);
        return 0; // success
}

static int teardown(void) {
        // Do nothing.
        // Especially: do not remove result files - they are removed in int setup(void) *before* running a test.
        return 0; // success
}
```

```c
static void test_comparePerson() {
    CU_ASSERT_TRUE(comparePerson(testperson1, testperson2));
    CU_ASSERT_TRUE(comparePerson(testperson2, testperson3));
    CU_ASSERT_EQUAL(comparePerson(testperson1, testperson3), 0);
}
```

```c
static void test_show() { // arrange
    const char *out_txt[] = {
        "Program name: /home/vele/progc/PROGC/Selbststudium/05/bin/linkedList\n", "Hi man! Great to see you, this is a very awesome program!\n", "Please enter the function that you wanna use! I(nsert), R(emove), S(how), C(lear), E(nd)\n", "Please enter lastname, firstname, age from your person (space seperated)\n", "Please enter the function that you wanna use! I(nsert), R(emove), S(how), C(lear), E(nd)\n", "I am the, head node, 1000\n", "Talamona, Colin, 9\n", "Please enter the function that you wanna use! I(nsert), R(emove), S(how), C(lear), E(nd)\n",
    }; // act
    int exit_code = system(XSTR(TARGET)
        " 1>"
        OUTFILE " 2>"
        ERRFILE " <"
        SHOW_USECASE); // assert
    CU_ASSERT_EQUAL(exit_code, 0);
    assert_lines(OUTFILE, out_txt, sizeof(out_txt) / sizeof( *out_txt));
}
```

```c
int main(void) { // setup, run, teardown
```

```
2.      TestMainBasic("Hello World", setup, teardown, test_comparePerson, test_dispose, test_cr
    eate, test_insertAlphabetical, test_removeAny, test_show);
3.  }
```

CU_ASSERT_PTR_NULL(head->next);

CU_ASSERT_STRING_EQUAL(testNode->person.lastname, testperson1.lastname);

CU_ASSERT_EQUAL(testNode->person.age, testperson1.age);

10.5 Bit-Shifting