Die Programmiersprache C

Copyright 2004 by Christian Bartl
bartl.christian@aon.at

Index

1. Einleitung 6
1.1. Entwicklungsgeschichte 6
1.2. Der Maschinencode 6
1.3. Der Assembler 6
1.4. Die höheren Programmiersprachen 6
1.5. Die Entstehung eines Programms 6
2. Entwicklungsumgebung 6
2.1. Der Compiler 6
2.1.1. Der C-Compiler 7
2.2. Der Interpreter 7
2.3. Der Debugger 8
3. Algorithmus 8
3.1. Vom Problem zur Lösung 9
3.2. Die Darstellung eines Algorithmus 9
3.2.1. Die verbale Beschreibung 9
3.2.2. Der Programmablaufplan (PAP) 10
3.2.3. Das Struktogramm 12
4. Datentypen 13
4.1. Integrale Datentypen 13
4.2. Gleitkomma Datentypen 13
4.3. Weitere Datentypen 13
4.4. Formatspezifizierer 14
4.5. Hinweise zur Typumwandlung 14
5. Grundlagen 15
5.1. Variablen 15
5.1.1. Referenz 15
5.1.2. Dereferenz 15
5.2. Standardbibliotheken 15
5.3. Der C-Zeichensatz 15
5.4. Zeichensätze 16
5.4.1. ASCII-CZeichensatz 16
5.4.2. ANSI-Zeichensatz 18
5.4.3. ISO Latin-1-Zeichensatz 19
5.5. Escape Sequenzen 19
5.6. Kommentare 20
5.7. C-Schlüsselwörter 20
6. Aufbau eines C-Programms 20
6.1. Verfügbarkeit und Lebensdauer von Namen 20
6.1.1. Verfügbarkeit 20
6.1.2. Lebensdauer 20
6.1.3. Gültigkeitsbereich 21
6.2. Der Aufbau eines C-Programms 21

6.2.1. „Hello World“ 21
7. Kontrollstrukturen 22
7.1. Verzweigungen 22
7.1.1. Einseitige Verzweigung 22
7.1.2. Zweiseitige Verzweigung 22
7.1.3. Mehrfachverzweigung 22
7.2. Wiederholungsstrukturen (Schleifen) 23
7.2.1. Abweisende Wiederholung 23
7.2.2. Nicht abweisende Wiederholung 23
7.2.3. Endloswiederholung 23
7.3. Unstrukturierte Kontrollanweisung 24
8. C-Präprozessor 24
9. Headerdatein 24
9.1. Der Aufbau einer Headerdatei 24
9.2. Einbinden einer Headerdatei in den Quellcode 24
10. Operatoren 25
10.1. Arithmetische Operatoren 25
10.2. Logische Operatoren 25
10.3. Vergleichsoperatoren 25
10.4. Bitmanipulation 25
10.5. Zusammengesetzte Operatoren 26
10.6. Größenoperatoren 26
10.7. Adressoperatoren 26
10.8. Cast Operatoren (Typcast) 26
10.8.1. Expliziter Typecast 26
10.8.2. Impliziter Typecast 27
10.9. Verschiedene Bedeutung der Operatoren 27
11. Ein- und Ausgabe 27
11.1. Eingabe 27
11.2. Ausgabe 27
11.3. Abfragen 27
12. Arrays (Felder) 28
12.1. Eindimensionale Felder 28
12.1.1. Syntax 28
12.1.2. Deklaration 28
12.1.3. Initialisierung 28
12.2. Zweidimensionale Felder 28
12.2.1. Syntax 28
12.2.2. Deklaration 28
12.2.3. Initialisierung 29
13. Strings (Zeichenketten) 29
13.1. Deklaration 29
13.2. Initialisierung 29
13.3. Ein- und Ausgabe 29

13.4. Stringkonstanten 29
13.5. Schreibweisen 30
13.6. Stringmanipulation 30
13.6.1. Strings kopieren 30
13.6.2. Strings anhängen 30
13.6.3. Strings vergleichen 30
13.6.4. Stringlänge ermitteln 30
13.6.5. Strings auf 0 initialisieren 30
14. Pointer (Zeiger) 30
14.1. Deklaration 31
14.2. Initialisierung 31
14.3. Zeiger auf 0 initialisieren 31
14.4. Zeigerarithmetik 31
14.5. Adressen eines Arrays ermitteln 31
14.6. Zeiger auf einen String 32
15. Dynamische Felder 32
16. Argumente der Kommandozeile 32
17. Funktion 33
17.1. Syntax 33
17.2. Funktionsprototyp 33
17.3. Funktionsaufruf 33
17.4. Funktionsdefinition 33
17.5. Call by Value 34
17.6. Call by Reference 34
17.7. Funktionszeiger 34
17.8. Array als Übergabeparameter 35
18. Suchen und Sortieren 35
18.1. Suchen 35
18.1.1. Lineare Suche 35
18.1.2. Binäre Suche 36
18.2. Sortieren 37
18.2.1. Bubblesort 37
18.2.2. Quicksort 38
18.3. Standardsuch- und Sortierfunktionen 39
18.3.1. Binäre Suche 39
18.3.2. Quicksort 39
19. Speicherklassen 40
20. Dateien 40
20.1. Standardoperationen auf Dateien 41
20.2. Standard Ein- und Ausgabe 41
20.3. Textdatei 41
20.4. Binärdatei 41
20.5. Beschreiben und Lesen von Dateien 42
20.5.1. Definieren einer Datei 42

20.5.2. Öffnen einer Datei 42
20.5.3. Formatierte Ausgabe auf Datei 43
20.5.4. Formatierte Eingabe von Datei 43
20.5.5. Dateiende abfragen 43
20.5.6. Schließen einer Datei 43
20.5.7. Beispiel 43
21. Strukturen 44
21.1. Deklaration 44
21.2. Initialisierung 45
21.3. Typedef 45
21.4. Union 45
21.5. Bitfelder 46
22. Komplexe Deklarationen 46
23. Wichtige Befehle 46
23.1. Der exit-Status 46
23.2. Bildschirminhalt löschen 47
23.3. Tastaturpuffer leeren 47
24. Wichtige Algorithmen 47
24.1. Vertauschalgorithmus 47
24.2. Vergleichsalgorithmus 47
24.3. Summenalgorithmus 48
24.4. Erzeugen von Zufallszahlen 48

1. Einleitung

1.1. Entwicklungsgeschichte

C erscheint 1977 als Programmiersprache und wurde von Kerneghie und Ritchie entwickelt. Diese entwickelten unter anderem auch das Betriebssystem Unix.

1.2. Der Maschinencode

Der Computer kann nur mit 2 Befehlen arbeiten. Diese sind 0 und 1. 1 bedeutet true, also wahr, und 0 bedeutet false, also falsch.

1.3. Der Assembler

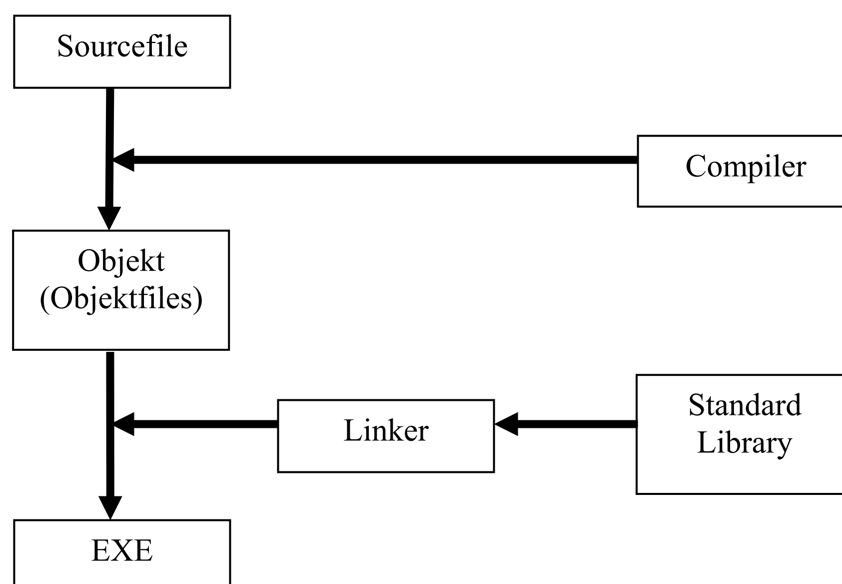
Der Assembler wandelt mit Hilfe eines Compilers die Befehle in den Maschinencode um. Typische Befehle sind: mov, push, pop, add, usw.

1.4. Die höheren Programmiersprachen

Höhere Programmiersprachen sind: C, C++. C#, Basic, Pascal, Cobold, Fortran, Algol, Java, usw.

1.5. Die Entstehung eines Programms

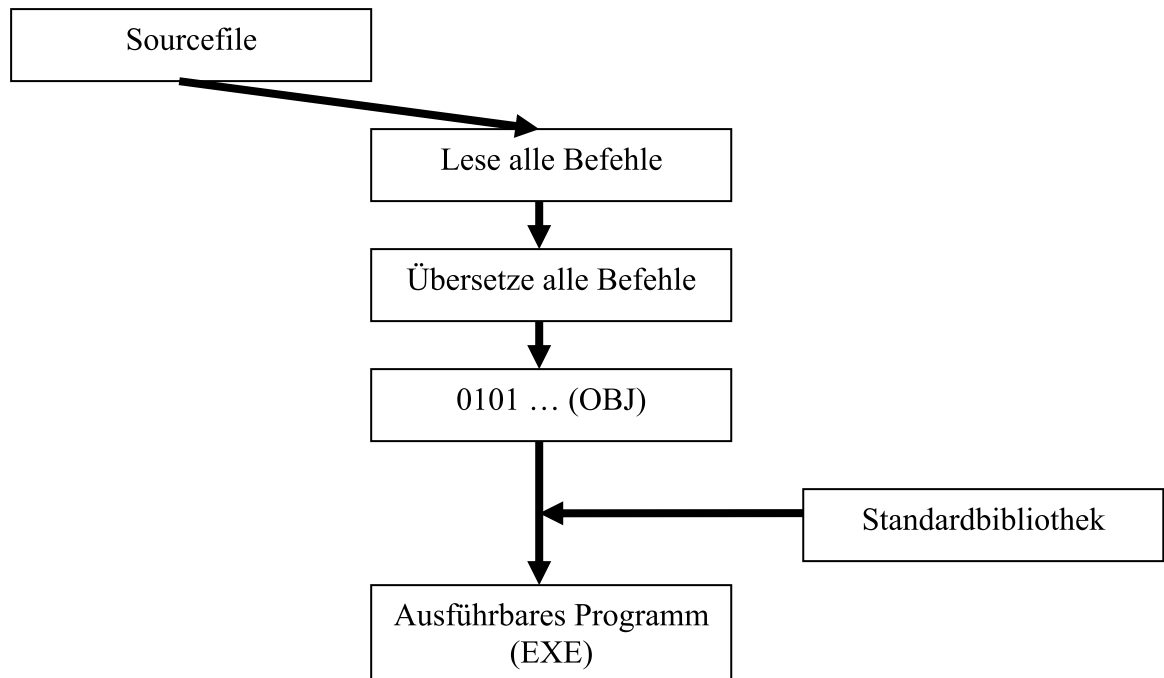
Um ein Programm entstehen zu lassen, benötigt man zuerst einmal einen Programmcode. Dieser wird in einem Editor oder einem Programmiertool (z.B. Visual Studio von Microsoft) geschrieben. Danach erzeugt der Compiler einen oder mehrere Objektfiles aus dem Programmcode. Der Linker bindet noch die Standard Library ein und fertig ist das EXE-File, also das fertige Programm. Exe bedeutet execute able, also ausführbar. Der EXE-File ist im Maschinencode verfasst.



2. Entwicklungsumgebung

2.1. Der Compiler

Der Compiler übersetzt den Programmcode (Sourcefile) in die Maschinensprache, die Objektfiles.



2.1.1. Der C-Compiler

Der C-Compiler ist ein 3-Phasen Compiler, das heißt, er compiliert ein Programm in 3 Schritten.

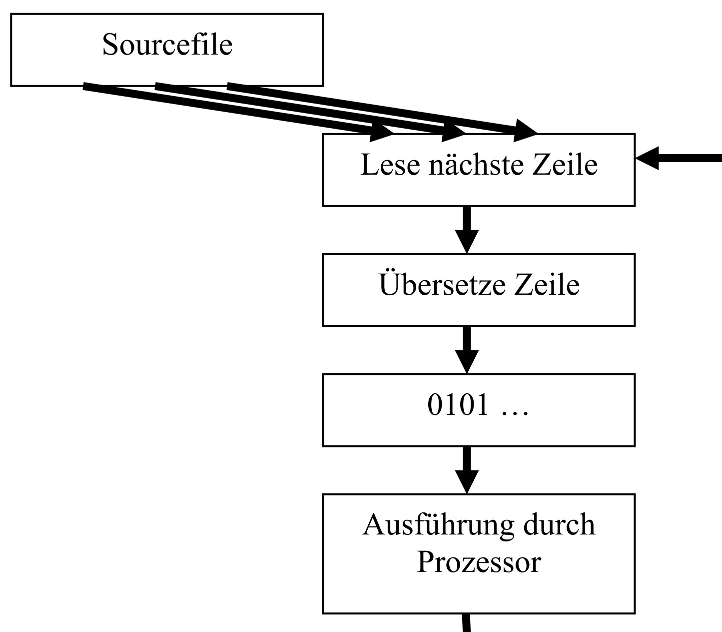
1.Phase: Die Präprozessorbefehle werden aufgelöst.

2.Phase: Der C-Code wird compeliert.

3.Phase: Der compilierte Code wird gelinkt.

2.2. Der Interpreter

Der Interpreter übersetzt Zeile für Zeile des Sourcefiles in den Maschinencode. Die übersetzte Zeile wird vom Prozessor sofort ausgeführt, danach wird die nächste Zeile übersetzt.



2.3. Der Debugger

Bei der Entwicklung einer Software unterscheidet man zwischen folgenden Fehlern:

- Compilerfehler (z.B. Rechtschreibfehler im Programmcode)
- Linkerfehler (z.B. fehlende Headerdateien)
- Laufzeitfehler (z.B. Zugriff auf falschen Speicherbereich)
- Logikfehler (z.B. Das Programm rechnet falsch)

Compilerfehler und Linkerfehler werden direkt als Fehlermeldung mit Zeilennummer beim Compilieren und Linken ausgegeben und müssen korrigiert werden. Erst danach ist das Programm lauffähig.

Laufzeitfehler und Logikfehler müssen während des Ausführens des Programms gefunden werden. Der Fachausdruck dafür lautet Debuggen oder zu Deutsch, entwanzen.

Dabei gibt es zwei Möglichkeiten:

1. Man arbeitet mit einer IDE (Integrated Development Environment), diese haben ein eingebautes Debugger Tool. Beispiel für IDE ist das Microsoft Visual Studio.

2. Besitzt die Programmierumgebung keinen Debug Tool, so kann man sich mit den Präprozessorbefehlen für die bedingte Compilierung abhelfen.
   Man definiert eine Konstante (einen Schalter) DEBUG nach dem Einbinden der Standardbibliotheken

```
#define DEBUG 1
```

wobei die 1 für DEBUG Modus EIN (Schalter gesetzt) steht und die 0 für DEBUG Modus AUS steht.
Der Code zum Debuggen wird in folgende Präprozessorbefehle eingeschlossen:

```
#if DEBUG == 1

    Code zum Debuggen

#endif
```

Das heißt, wenn DEBUG auf 1 steht, wird der Code mitcompeliert, steht DEBUG auf 0, wird der Code nicht mitcompeliert.

3. Algorithmus

Ein Algorithmus ist eine Vorschrift zur Lösung eines Problems.
(Suchen, Sortieren, Codieren, usw.)

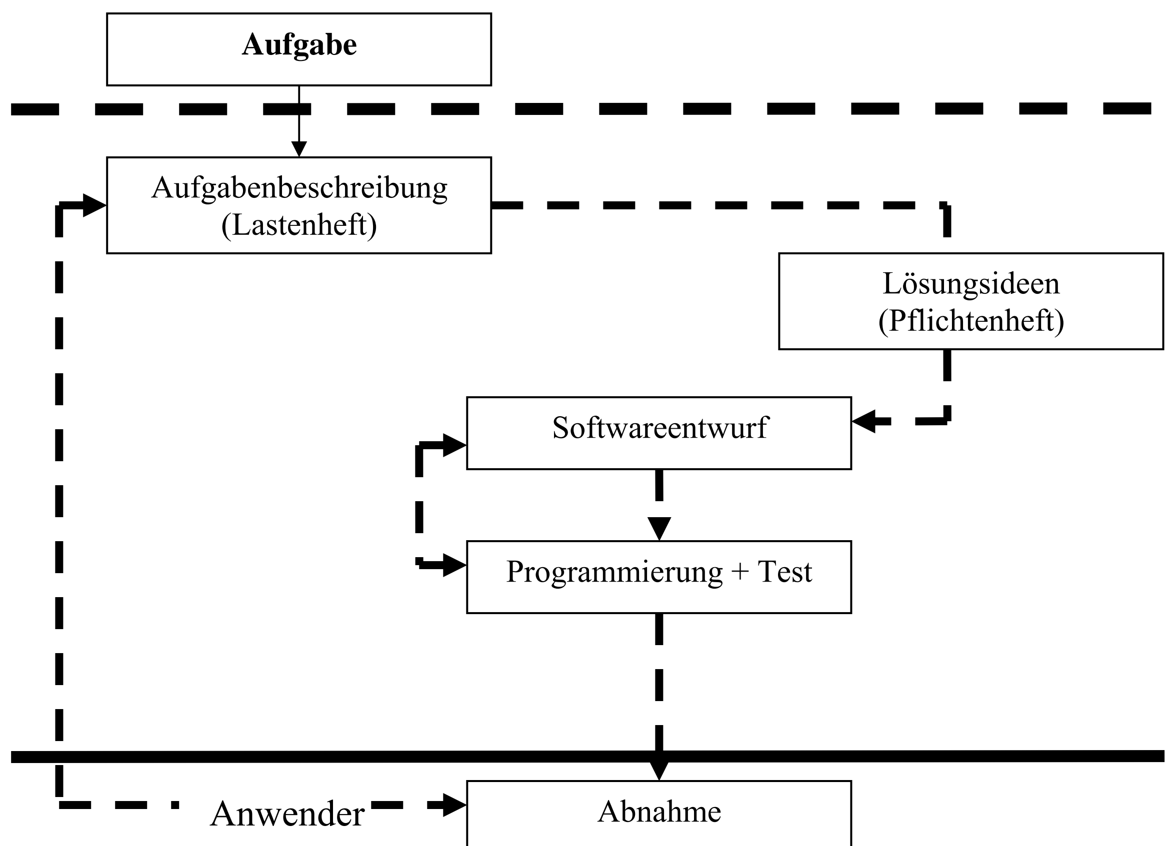
Die Korrektheit:
Die Korrektheit eines Algorithmus besteht in der Konsistenz (=Übereinstimmung) zwischen der in der Beschreibung gewünschten Problemlösung und seiner Realisierung.

Die Validation:
Die Validation ist die Prüfung, ob die Beschreibung eines Algorithmus mit dem zu lösenden Problem übereinstimmt.

Die Verifikation:
Unter Verifikation versteht man die Prüfung, ob ein Programm mit dem Algorithmus übereinstimmt.

3.1. Vom Problem zur Lösung

Bekommt ein Programmierer eine Aufgabe gestellt, geht er folgendermaßen vor: Zuerst erarbeitet er mit dem Auftraggeber (Anwender) eine Aufgabenbeschreibung, das Lastenheft. Danach sucht der Programmierer nach Lösungen, die er in das Pflichtenheft einträgt. Danach folgt der Softwareentwurf und danach die Programmierung und der Test der Software. Die beiden Vorgänge sind eng miteinander verknüpft. Nachdem der Programmierer den Test positiv abgeschlossen hat, erfolgt die Abnahme durch den Auftraggeber (Anwender).



3.2. Die Darstellung eines Algorithmus

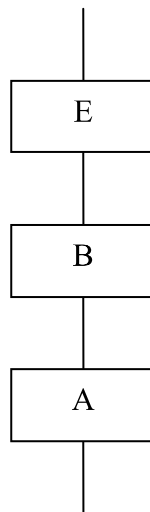
Um ein Programm zu entwickeln benötigt man einen Lösungsvorschlag, einen sogenannten Algorithmus. Um diesen Algorithmus nicht niederschreiben zu müssen, entwickelte man Verfahren um solche Lösungsvorschläge grafisch darzustellen. Und für jedes Konstrukt gibt es ein eigenes Plansymbol, in welches dann die wichtigsten Daten (Variable, Eingaben, Ausgaben, Berechnungen) eingetragen werden.

3.2.1. Die verbale Beschreibung

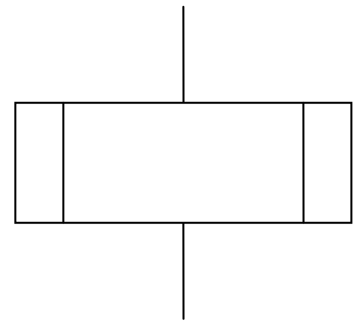
Wenn X>0 dann „Berechne“ sonst „Fehlermeldung“

3.2.2. Der Programmablaufplan (PAP)

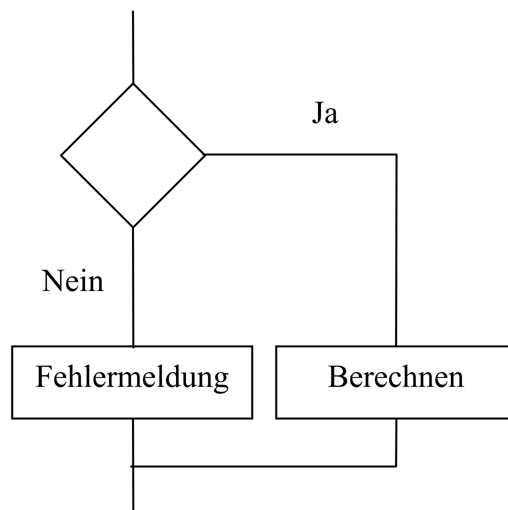
Sequenz



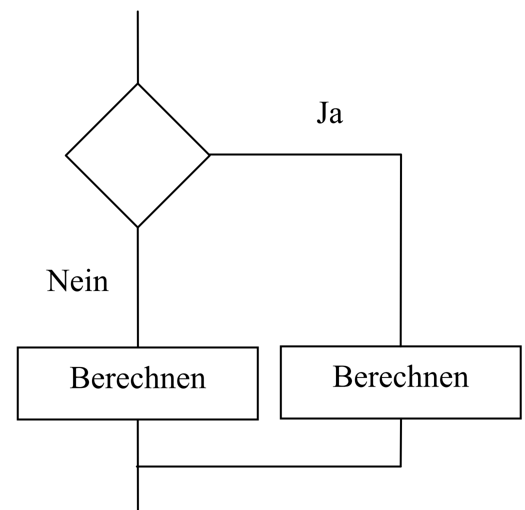
Unterprogramm (Funktion, Prozedur



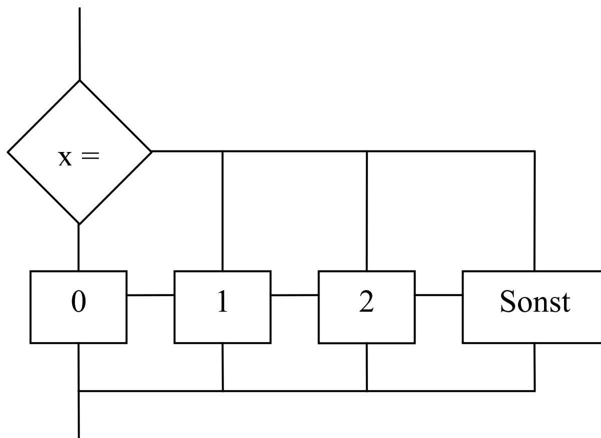
Einseitige Auswahl



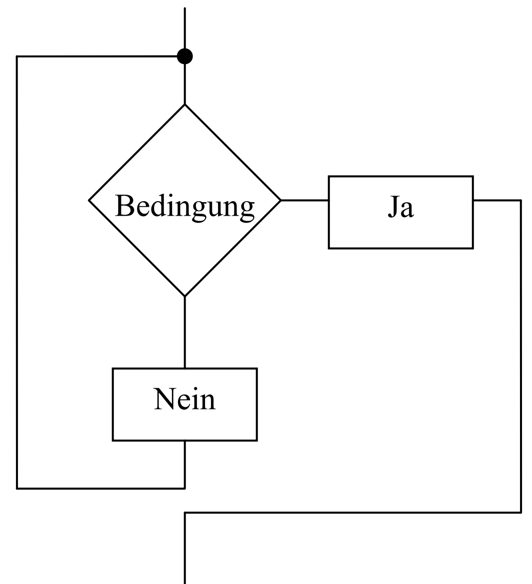
Zweiseitige Auswahl



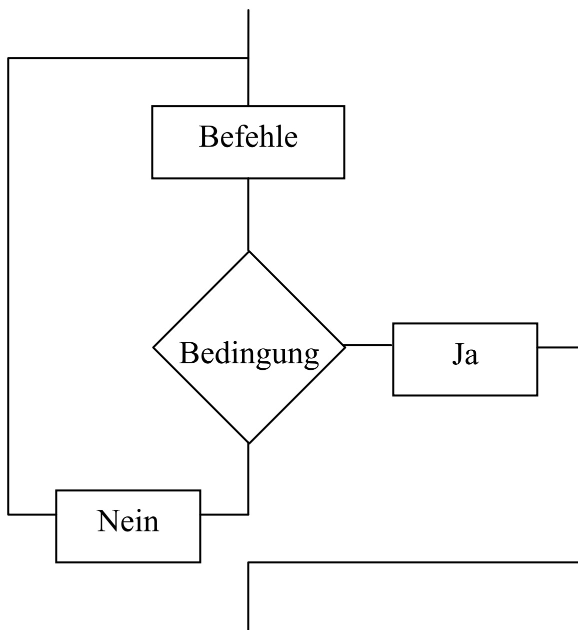
Mehrfache Auswahl



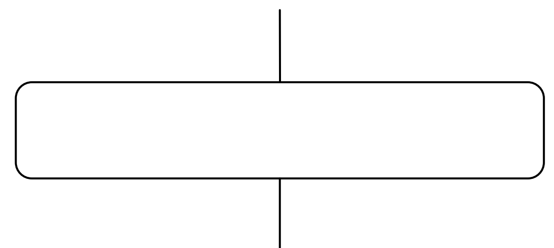
Abweisende Wiederholung



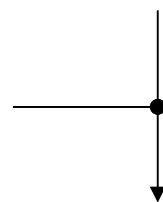
Nicht Abweisende Wiederholung



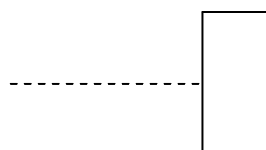
Grenzstelle



Zusammenführung



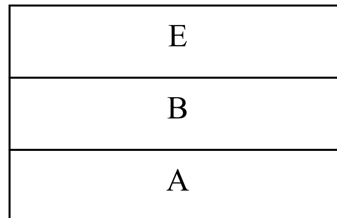
Kommentar



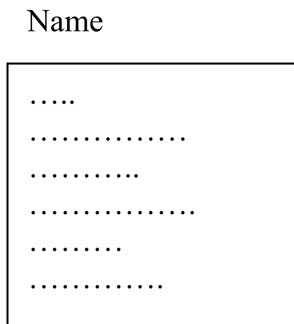
3.2.3. Das Struktogramm

Das Struktogramm ist die beliebteste Art ein Programm grafisch darzustellen. Das Struktogramm wurde von Nassi und Schneidermann erfunden.

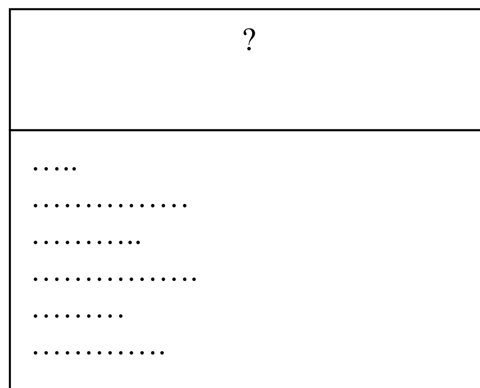
Sequenz



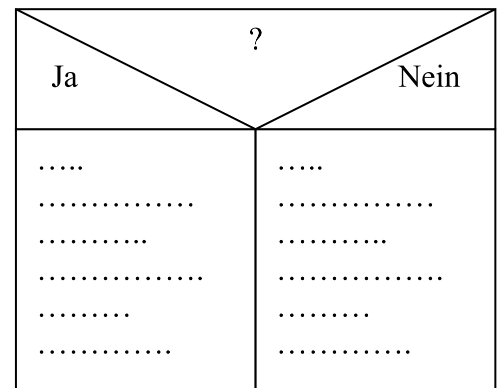
Unterprogramm
(Funktion, Prozedur)



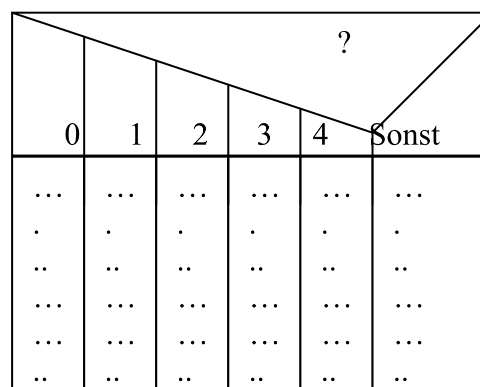
Einseitige Auswahl

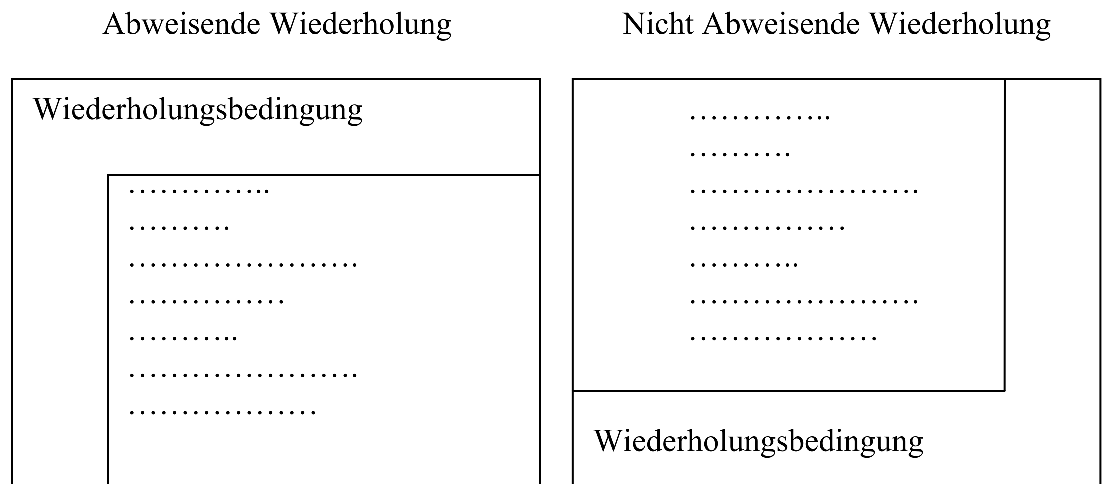


Zweiseitige Auswahl



Mehrfache Auswahl





4. Datentypen

Es gibt verschiedene Typen von Zahlengruppen, um Zahlen in einen gewissen Bereich einzuordnen und damit Speicherplatz zu sparen. Dabei unterscheidet man zwischen signed und unsigned, das heißt Variablen mit und ohne Vorzeichen. Außerdem gibt es Integrale Datentypen, ganzzahlige Werte, und Gleitkomma Datentypen, Werte mit Kommastellen. Dann gibt es noch Datentypen, mit denen man Zeichen, also zum Beispiel Buchstaben, abspeichern kann.

4.1. Integrale Datentypen

int	Integer	(-32768 … … 32767) 2 Byte
short int	Short Integer	(-32768 … … 32767) 2 Byte
long int	Long Integer	(-2147483648 ……214783647) 4 Byte

4.2. Gleitkomma Datentypen

float	Float	($3,4*10^{-38}$ …… $3,4*10^{38}$) 4 Byte
double	Double	($1,7*10^{-308}$ …… $1,7*10^{308}$) 8 Byte
long double	Long Double	($1,7*10^{-4932}$ …… $1,7*10^{4932}$) 10 Byte

4.3. Weitere Datentypen

char	Charaktar	(-127 …… 128) 1 Byte (256 Zeichen)
unsigned char	unsigned Charaktar	(0 …… 255) 1 Byte (256 Zeichen)

4.4. Formatspezifizierer

Formatspezifizierer werden benötigt, um Variablen mit Werten zu belegen, scanf(), oder auszugeben, printf(). Dabei muss immer der Datentyp angegeben werden.

%d	int
%f	float
%lf	double
%c	char
%s	Zeichenkette (String) → siehe Strings
%p	Zeiger (Pointer) → siehe Pointer

4.5. Hinweise zur Typumwandlung

Bei Umwandlungen von Datentypen mit *Vorzeichen (signed)* in solche *ohne Vorzeichen* (unsigned) bzw. umgekehrt, verliert bzw. erhält das äußerste linke Bit die Funktion des Vorzeichenbits.

Umwandlung	Begleiterscheinungen
double -> float	event. Genauigkeitsverlust (Rundungsfehler)
float -> double	keine Wertänderung
float -> long **float -> int** **float -> short** **float -> char**	zunächst Umwandlung in **long,** danach - falls erforderlich - in den entsprechenden "kleineren" ganzzahligen Datentyp. Nachkommastellen gehen verloren. Ergebnis undefiniert, wenn der umgewandelte Gleitkommawert zu groß für den Datentyp long ist.
long -> float **int -> float** **short -> float** **char -> float**	bei **int, short, char** zunächst Umwandlung in **long,** eventuell Genauigkeitsverlust falls der long - Wert zu groß für die Mantisse des float - Wertes ist.
long -> int **long -> short** **long -> char** **int -> short** **int -> char** **short -> char**	linke überzählige Bits, die nicht vom neuen Datentyp aufgenommen werden können, werden abgeschnitten
int -> long **short -> long** **char -> long** **short -> int** **char -> int** **char -> short**	keine Wertänderung

5. Grundlagen

5.1. Variablen

Variablen sind Platzhalter im Speicher und können mit Informationen unterschiedlicher Datentypen belegt werden. Variablen müssen immer am Anfang eines C-Programmes deklariert werden.

5.1.1. Referenz

Die Referenz ist die Adresse einer Variable im Speicher. Zum Beispiel A1FC wäre eine Angabe zu einem Speicherort im Hexadezimalsystem. Um später mit Zeigern arbeiten zu können ist es wichtig, den Unterschied zwieschen Referenz und Dereferenz zu kennen.

5.1.2. Dereferenz

Die Dereferenz ist der Wert einer Variable, der im Speicher abgelegt ist. Zum Beispiel a = 5, wobei 5 die Dereferenz ist.

5.2. Standardbibliotheken

Standardbibliotheken enthalten alle Befehle die man in einem C-Programm verwendet (Beispiel: printf()). Damit der Compiler die Befehle erkennt, müssen die Standardbibliotheken eingefügt werden.

stdlib.h	Standard Library
stdio.h	Standard Input-Output Library
conio.h	Standard Library
math.h	Library mit Mathematischen Funktionen
time.h	Library mit Zeitfunktionen

5.3. Der C-Zeichensatz

Großbuchstaben:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Kleinbuchstaben:

a b c d e f g h i j k l m n o p q r s t u v w x y z

Dezimalziffern:

0 1 2 3 4 5 6 7 8 9

Unterstreichungszeichen:

_

Nicht - sichtbare Zeichen:

Leerzeichen, Tabulatorzeichen, Wagenrücklaufzeichen, Zeilenvorschubzeichen

Satz- und Sonderzeichen:

, Komma	! Ausrufungszeichen
. Punkt		Vertikaler Balken
; Semilkon	/ Schrägstrich (slash)
: Doppelpunkt	\ Backslash
? Fragezeichen	~ Tilde

‘ Apostroph
“ Anführungszeichen
(Linke Klammer
) Rechte Klammer
[Linke eckige Klammer
] Rechte eckige Klammer
{ Linke geschweifte Klammer
} Rechte geschweifte Klammer
< Kleiner- Zeichen
+ Pluszeichen
# Nummernzeichen
% Prozentzeichen
& Ampersand
^ Caret
* Stern
- Minuszeichen
= Gleichheitszeichen
> Größer- Zeichen

5.4. Zeichensätze

5.4.1. ASCII-CZeichensatz

Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen
0	00		32	20		64	40	@	96	60	`
1	01	☺	33	21	!	65	41	A	97	61	a
2	02	☻	34	22	"	66	42	B	98	62	b
3	03	♥	35	23	#	67	43	C	99	63	c
4	04	♦	36	24	$	68	44	D	100	64	d
5	05	♣	37	25	%	69	45	E	101	65	e
6	06	♠	38	26	&	70	46	F	102	66	f
7	07	•	39	27	'	71	47	G	103	67	g
8	08	◘	40	28	(	72	48	H	104	68	h
9	09	○	41	29	)	73	49	I	105	69	i
10	0A	◙	42	2A	*	74	4A	J	106	6A	j
11	0B	♂	43	2B	+	75	4B	K	107	6B	k
12	0C	♀	44	2C	,	76	4C	L	108	6C	l
13	0D	♪	45	2D	-	77	4D	M	109	6D	m
14	0E	♫	46	2E	.	78	4E	N	110	6E	n
15	0F	—	47	2F	/	79	4F	O	111	6F	o
16	10	►	48	30	0	80	50	P	112	70	p
17	11	◄	49	31	1	81	51	Q	113	71	q
18	12	↕	50	32	2	82	52	R	114	72	r
19	13	‼	51	33	3	83	53	S	115	73	s
20	14	¶	52	34	4	84	54	T	116	74	t
21	15	§	53	35	5	85	55	U	117	75	u
22	16	▬	54	36	6	86	56	V	118	76	v
23	17	↨	55	37	7	87	57	W	119	77	w
24	18	↑	56	38	8	88	58	X	120	78	x
25	19	↓	57	39	9	89	59	Y	121	79	y
26	1A	→	58	3A	:	90	5A	Z	122	7A	z
27	1B	←	59	3B	;	91	5B	[	123	7B	{
28	1C	∟	60	3C	<	92	5C	\	124	7C	¦
29	1D	↔	61	3D	=	93	5D	]	125	7D	}
30	1E	▲	62	3E	>	94	5E	^	126	7E	~
31	1F	▼	63	3F	?	95	5F	_	127	7F	▒

ASCII-Code bedeutet American Standard Code for Information Interchanging. Um ein ASCII-Code-Zeichen über die Tastatur einzugeben drücken Sie die Alt-Taste und geben die Dezimalzahl des ASCII-Zeichens über den Nummernblock der Tastatur ein. Danach lassen Sie die Alt-Taste los und das ASCII-Zeichen erscheint auf dem Bildschirm.

Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	•
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	η
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

5.4.2. ANSI-Zeichensatz

Windows verwendet den ANSI-Zeichensatz, der sich in den ersten 128 Zeichen mit dem ASCII-Zeichensatz deckt.

Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen
128	80	€	160	A0		192	C0	À	224	E0	à
129	81		161	A1	¡	193	C1	Á	225	E1	á
130	82	‚	162	A2	¢	194	C2	Â	226	E2	â
131	83	ƒ	163	A3	£	195	C3	Ã	227	E3	ã
132	84	„	164	A4	¤	196	C4	Ä	228	E4	ä
133	85	…	165	A5	¥	197	C5	Å	229	E5	å
134	86	†	166	A6	¦	198	C6	Æ	230	E6	æ
135	87	‡	167	A7	§	199	C7	Ç	231	E7	ç
136	88	ˆ	168	A8	¨	200	C8	È	232	E8	è
137	89	‰	169	A9	©	201	C9	É	233	E9	é
138	8A	Š	170	AA	ª	202	CA	Ê	234	EA	ê
139	8B	‹	171	AB	«	203	CB	Ë	235	EB	ë
140	8C	Œ	172	AC	¬	204	CC	Ì	236	EC	ì
141	8D		173	AD	-	205	CD	Í	237	ED	í
142	8E	Ž	174	AE	®	206	CE	Î	238	EE	î
143	8F		175	AF	¯	207	CF	Ï	239	EF	ï
144	90		176	B0	°	208	D0	Ð	240	F0	ð
145	91	‘	177	B1	±	209	D1	Ñ	241	F1	ñ
146	92	’	178	B2	²	210	D2	Ò	242	F2	ò
147	93	“	179	B3	³	211	D3	Ó	243	F3	ó
148	94	”	180	B4	´	212	D4	Ô	244	F4	ô
149	95	•	181	B5	µ	213	D5	Õ	245	F5	õ
150	96	–	182	B6	¶	214	D6	Ö	246	F6	ö
151	97	—	183	B7	·	215	D7	×	247	F7	÷
152	98	˜	184	B8	¸	216	D8	Ø	248	F8	ø
153	99	™	185	B9	¹	217	D9	Ù	249	F9	ù
154	9A	š	186	BA	º	218	DA	Ú	250	FA	ú
155	9B	›	187	BB	»	219	DB	Û	251	FB	û
156	9C	œ	188	BC	¼	220	DC	Ü	252	FC	ü
157	9D		189	BD	½	221	DD	Ý	253	FD	ý
158	9E	ž	190	BE	¾	222	DE	Þ	254	FE	þ
159	9F	Ÿ	191	BF	¿	223	DF	ß	255	FF	ÿ

5.4.3. ISO Latin-1-Zeichensatz

Die Betriebssysteme Unix und Linux verwenden den ISO Latin-1, der in den ersten 128 zeichen dem ASCII-Zeichensatz entspricht.

Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen
128	80		160	A0		192	C0	À	224	E0	à
129	81		161	A1	¡	193	C1	Á	225	E1	á
130	82		162	A2	¢	194	C2	Â	226	E2	â
131	83		163	A3	£	195	C3	Ã	227	E3	ã
132	84		164	A4	¤	196	C4	Ä	228	E4	ä
133	85		165	A5	¥	197	C5	Å	229	E5	å
134	86		166	A6	¦	198	C6	Æ	230	E6	æ
135	87		167	A7	§	199	C7	Ç	231	E7	ç
136	88		168	A8	¨	200	C8	È	232	E8	è
137	89		169	A9	©	201	C9	É	233	E9	é
138	8A		170	AA	ª	202	CA	Ê	234	EA	ê
139	8B		171	AB	«	203	CB	Ë	235	EB	ë
140	8C		172	AC	¬	204	CC	Ì	236	EC	ì
141	8D		173	AD	-	205	CD	Í	237	ED	í
142	8E		174	AE	®	206	CE	Î	238	EE	î
143	8F		175	AF	¯	207	CF	Ï	239	EF	ï
144	90		176	B0	°	208	D0	Ð	240	F0	ð
145	91		177	B1	±	209	D1	Ñ	241	F1	ñ
146	92		178	B2	²	210	D2	Ò	242	F2	ò
147	93		179	B3	³	211	D3	Ó	243	F3	ó
148	94		180	B4	´	212	D4	Ô	244	F4	ô
149	95		181	B5	µ	213	D5	Õ	245	F5	õ
150	96		182	B6	¶	214	D6	Ö	246	F6	ö
151	97		183	B7	·	215	D7	×	247	F7	÷
152	98		184	B8	¸	216	D8	Ø	248	F8	ø
153	99		185	B9	¹	217	D9	Ù	249	F9	ù
154	9A		186	BA	º	218	DA	Ú	250	FA	ú
155	9B		187	BB	»	219	DB	Û	251	FB	û
156	9C		188	BC	¼	220	DC	Ü	252	FC	ü
157	9D		189	BD	½	221	DD	Ý	253	FD	ý
158	9E		190	BE	¾	222.	DE	Þ	254	FE	þ
159	9F		191	BF	¿	223	DF	ß	255	FF	ÿ

5.5. Escape Sequenzen

\n	führt Zeilenvorschub durch (new line)
\t	setzt Horizontaltabulator
\v	setzt Vertikaltabulator
\b	geht ein Zeichen zurück (backspace)
\r	führt Wagenrücklauf durch (carriage return)
\f	führt Seitenvorschub durch (form feed)
\a	löst Klingelzeichen aus (Alarm)
\’	Hochkomma
\”	Anführungszeichen
\\	Umgekehrter Schrägstrich (backslash)
\ddd	ASCII – Zeichen in Oktalnatation
\xddd	ASCII – Zeichen in Hexadezimalnatation

5.6. Kommentare

Wenn man Kommentare in einen Quellcode einfügt, dürfen diese beim Compelieren nicht beachtet werden. Dafür gibt es in C und C++ zwei Arten um Kommentare zu kennzeichnen.

/* …………..*/	Kommentar in C++
// …………….	Kommentar in C

5.7. C-Schlüsselwörter

Schlüsselwörter sind reservierte Namen und dürfen somit nicht als Variablennamen, Funktionsname, Strukturname, usw. dienen.

auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, while, volatile, void, unsigned, union, typedef, switch, struct, long, register, return, short, signed, sizeof, static

6. Aufbau eines C-Programms

6.1. Verfügbarkeit und Lebensdauer von Namen

Jedes Programm besitzt eine Struktur. Für Namen von Variablen, Konstanten und Funktionen ist die Verfügbarkeit und Lebensdauer von Wichtigkeit.

6.1.1. Verfügbarkeit

Unter Verfügbarkeit versteht man, ob man auf ein Objekt zugreifen kann oder nicht. Es gibt mehrere Bereiche eines Programms, mit denen auf ein Objekt zugegriffen werden kann. In C unterscheidet man zwischen 4 verschiedenen Verfügbarkeitsbereichen.

1. Projekt (Programm)
2. Datei
3. Funktion
4. Block (Verbundanweisung { …. })

Objekte die in einem Programm oder in einer Datei definiert werden, sind lokale Objekte. Funktionen sind in C immer Global. Objekte, die innerhalb eines Blocks oder einer Funktion definiert werden, sind Lokale Objekte. Lokale Objekte können nur Variablen sein.

6.1.2. Lebensdauer

Mit Lebensdauer ist die Zeit der Zuordnung von Speicherplatz zu einem Objekt gemeint. In C unterscheidet man zwischen zwei Arten.

Static Duration:
Hier ist die Lebenszeit auf die Programmlaufzeit ausgedehnt. Der Speicherbereich wird dem Objekt bei Programmbeginn zugeordnet und bei Beendigung wieder aufgehoben.

Automatic Duration:
Hier ist die Lebenszeit auf einen Block eingeschränkt. Dem Objekt wird mit Eintritt des Programms in einen Block automatisch Speicher zugewiesen und mit Verlassen des Blockes wieder aufgehoben.

6.1.3. Gültigkeitsbereich

Hier geht es um die Überlagerung von Variablennamen innerhalb eines Programms. So werden Globale Variablen während des Ausführens einer Funktion von gleichnamigen lokalen Variablen überdeckt und sind somit nicht verfügbar. Der Gültigkeitsbereich eines Namens, welcher innerhalb einer Funktion vereinbart wurde, ist demzufolge auf die Funktion begrenzt. Dies gilt auch für die Parameter der Funktion.

6.2. Der Aufbau eines C-Programms

Am Anfang jedes C-Programms müssen die Standardbibliotheken eingefügt werden. Danach kommen die Funktionsprototypen und die Globalen Variablen. Das eigentliche Programm fängt erst mit der Funktion main an. Der Programmcode einer Funktion muss immer in geschwungenen Klammern stehen. Das void steht für keinen Rückgabewert der Funktion main und ist eine sehr hilfreiche Formsache, die aber weggelassen werden kann. Am Schluss kommen dann noch die eigentlichen Funktionen.

```
# include <Standardbibliothek.h>

  Funktionsprototypen
     Globale Variablen

void main (void)
     {
       Lokale Variablen
     Programmcode
     }

     Funktionen
```

6.2.1. „Hello World“

Das wohl bekannteste Programmbeispiel (in jeder Programmiersprache) ist das „Hello World“-Programm.

```
# include <stdlib.h>

void main (void)
        {
                printf („Hello World“);
        }
```

Dieses Programm gibt den Text „Hello World“ am Bildschirm aus. Wie man bei diesem Beispiel sieht, muss man in C nach jeder Programmzeile einen Strichpunkt setzen. Diesen darf man nicht vergessen, da der Compiler sonst nicht weiß, wo eine Programmzeile zu Ende ist. Häufig wird auch eine Klammer oder das Einfügen der Standardbibliotheken vergessen. Darum ist es sinnvoll, Klammern sofort wieder zu schließen und dann erst den Programmcode einzufügen. Bei unauffindbaren Fehlern ist es oftmals ratsam, eine 2.Person den Quellcode durchlesen zu lassen, da man vor allem

vergessene Strichpunkte und seine eigenen Rechtschreib- und Tippfehler gerne überliest.

7. Kontrollstrukturen

7.1. Verzweigungen

Verzweigungen werden benötigt um einen Programmablauf durch das Erfüllen oder Nichterfüllen von Bedingungen zu steuern. Verzweigungen können einen, zwei oder mehrere Zweige besitzen. Für die Zwei- und Mehrseitige Verzweigung gibt es zwei Arten diese auszuprogrammieren. Es erfüllen aber beide den selben Zweck.

7.1.1. Einseitige Verzweigung

```
if (Bedingungsausdruck)
{
  Anweisungen;
}
```

7.1.2. Zweiseitige Verzweigung

1.Möglichkeit:

```
if (Bedienungsausdruck)
{
        Anweisungen 1;
}
else
{
        Anweisungen 2;
}
```

2.Möglichkeit:

```
Bedienungsaudruck ? Anweisung 1 : Anweisung 2
```

7.1.3. Mehrfachverzweigung

1.Möglichkeit:

```
if (Bedienungsausdruck 1)
{
        Anweisungen 1;
}
else if (Bedienungsausdruck 2)
{
        Anweisungen 2;
}
else
{
        Anweisungen n;
}
```

2.Möglichkeit:

```
switch (Bedienungsausdruck)
{
        case F1: Anweisungen 1; break;
        case F2: Anweisungen 2; break;
```

```
        case Fn: Anweisungen n; break;
        default: Anweisungen;
}
```

7.2. Wiederholungsstrukturen (Schleifen)

Wiederholungsstrukturen benötigt man, um Programmteile oder ganze Programme so oft zu wiederholen, bis eine Bedienung erfüllt ist. Es gibt auch sogenannte Endlosschleifen, die einfach immer wieder von vorne beginnen bis das Programm abgebrochen wird. Dabei gibt es kopf- und fußgesteuerte Schleifen. Bei kopfgesteuerten Schleifen wird zuerst eine Bedingung überprüft und dann erst die Anweisungen ausgeführt. Bei fußgesteuerten ist es genau umgekehrt, hier wird zuerst die Anweisung ausgeführt und dann wird erst die Bedingung überprüft.

7.2.1. Abweisende Wiederholung

1.Möglichkeit:

```
for (Einstiegsbedienung; Ausstiegsbedienung; Schleifenzähler)
{
        Anweisungen;
}
```

Einstiegsbedienung = Initialisierung der Laufvariable
Ausstiegsbedienung = Laufbedienung
Schleifenzähler = Erhöhung/Erniedrigung der Laufvariable

2.Möglichkeit:

```
while (Laufbedienung)
{
        Anweisungen;
}
```

7.2.2. Nicht abweisende Wiederholung

```
do
{
  Anweisungen;
}
while (Laufbedienung);
```

7.2.3. Endloswiederholung

1.Möglichkeit:

```
for (;;)
{
        Anweisungen;
}
```

2.Möglichkeit:

```
while (0)
{
        Anweisungen;
}
```

7.3. Unstrukturierte Kontrollanweisung

marke, goto marke:

Der Befehl marke wird verwendet, um innerhalb eines Programms zu gewissen Abschnitten springen zu können. Diesen Befehl sollte man aber möglichst vermeiden und solche Probleme mit Schleifen lösen.

break;

Der Befehl break bricht eine Verzweigung ab und geht heraus.

continue;

Der Befehl continue bewirkt einen Sprung zum Ende einer Schleife.

8. C-Präprozessor

# include < … >	Bewirkt das Einbinden von Standardbibliotheken.
# include “ … “	Bewirkt das Einbinden von eigenen Bibliotheken.
# define	Mit define definiert man Konstanten. Bsp.: # define MAX 1000
# undef	Mit undefine löscht man Konstanten. Bsp.: # undefine MAX
# if # else # endif	Diese Befehle werden für Bedingte Compelierung verwendet.

Bsp.:

```
# if MAX > 1000
# define MIN 100
# else
# define MIN 200
# endif
```

9. Headerdatein

Headerdateien sind durch die Endung “.h” erkennbar. Headerdateien werden immer am Beginn des Quelltextes eingefügt. In Headerdateien findet der Compiler: Funktionsprototypen, Konstanten, Präprozessor Makros

9.1. Der Aufbau einer Headerdatei

```
#ifndef _NAME_DER_HEADERDATEI_H
#define _NAME_DER_HEADERDATEI_H
    :
    :
    Inhalt der Headerdatei: Konstanten, Präprozessor Makros, Funktionsprototypen
    :
    :
#endif
```

9.2. Einbinden einer Headerdatei in den Quellcode

```
#ifndef _NAME_DER_HEADERDATEI_H
#include „NAME.H“
#endif
```

10. Operatoren

10.1. Arithmetische Operatoren

+ Addition
- Subtraktion
- Vorzeichen
* Multiplikation
/ Division
% Divisionsrest (Modulo)

Beispiel :

```
int h ,i ,k ;
h=6 ;
i=2;

    k=h+i       //k=8
    k=h-i       //k=4
    k=h*i       //k=12
    k=h/i       //k=3
    k=h%i       //k=0
```

10.2. Logische Operatoren

&& logisches und
|| logisches oder
! logische Negation (logisches nicht)

Aussage1 (A1)	**Aussage 2 (A2)**	**A1 && A2**	*A1 || A2*	**! A1**
falsch	falsch	falsch	falsch	wahr
falsch	wahr	falsch	wahr	wahr
wahr	falsch	falsch	wahr	falsch
wahr	wahr	wahr	wahr	falsch

10.3. Vergleichsoperatoren

== gleich (identisch)
< kleiner
> größer
<= kleiner gleich
>= größer gleich
!= ungleich

10.4. Bitmanipulation

~ Komplement („Tilde“)
<< Linksshift
>> Rechtsshift
& bitweise UND („Ampersand“)
| bitweise ODER
^ bitweise EXCLUSIV ODER (EXOR)

Bespiel:

```
        00001010
~       11110101

i       00001111
i<<4    11110000
i>>4    00000000

        00001101
        11101010

&       00001000
|       11101111
^       11100111
```

10.5. Zusammengesetzte Operatoren

+=
i += m → i= i+m

-=
i -= m → i= i-m

***=**
i *= m → i= i*m

/=
i /= m → i= i/m

%=
i %= m → i= i%m

++ :
i++ → i wird abgefragt und dann um 1 erhöht
++i → i wird um 1 erhöht und dann abgefragt

-- :
i-- → i wird abgefragt und dann um 1 erniedrigt
--i → i wird um 1 erniedrigt und dann abgefragt

10.6. Größenoperatoren

sizeof ( ) Der Operator sizeof gibt die Größe eines Datentyps, einer Variable oder eines Feldes als Integer-Wert zurück.

10.7. Adressoperatoren

& Der Adressoperator gibt die Adresse eine Variable zurück.

10.8. Cast Operatoren (Typcast)

10.8.1. Expliziter Typecast

( ) Wandelt einen Datentyp in einen anderen um

Beispiel:

```
int i, j, k;
i=5;
j=4;
```

```
k=i/j;   → k= (double) i/j;
```

10.8.2. Impliziter Typecast

Unter implizitem Typcasting versteht man das Speichern eines datenreichen Datentyps (Ergebnisse von Rechnungen) auf einen datenärmeren Datentyp. Zum Beispiel eine Division vom Typ double wird auf die Variable i vom Typ int gespeichert. Achtung dabei entsteht ein Datenverlust!

```
int i;
double a, b;
i=a/b;
```

10.9. Verschiedene Bedeutung der Operatoren

( )	Klammerung und Casting
*	Multiplikation und Zeiger
-	Vorzeichen und Subtraktion
&	Adresse und Bitweises UND

11. Ein- und Ausgabe

11.1. Eingabe

scanf Scanf ist eine formatierte Eingabe und muss mit Enter bestätigt werden. Scanf ist in der Standardbibliothek stdio.h zu finden.

getchar Getchar ist eine zeichenweise Eingabe und muss mit Enter bestätigt werden. Getchar ist in der Standardbibliothek stdio.h zu finden.

getch Getch ist eine zeichenweise Eingabe und muss nicht mit Enter bestätigt werden. Getch ist in der Standardbibliothek conio.h zu finden.

getche Getche ist eine zeichenweise Eingabe und muss nicht mit Enter bestätigt werden und das eingegebene Zeichen wird sofort wieder ausgegeben. Getche ist in der Standardbibliothek conio.h zu finden.

11.2. Ausgabe

printf Printf ist eine formatierte Ausgabe. Printf ist in der Standardbibliothek stdio.h zu finden.

putchar Putchar ist eine zeichenweise Ausgabe. Putchar ist in der Standardbibliothek stdio.h zu finden.

11.3. Abfragen

EOF EOF bedeutet End of File, das heißt ein Programm kann mit der Tastenkombination „Strg+Z“ beendet werden. EOF ist in der Standardbibliothek conio.h zu finden.

12. Arrays (Felder)

Achtung in Microsoft Visual Studio beginnen die Felder bei 0 und enden einen Wert früher. Das heißt die Größe des Feldes ist immer gleich der Anzahl der benötigten Felder.

12.1. Eindimensionale Felder

12.1.1. Syntax

Datentyp *Name* [Größe des Feldes]

12.1.2. Deklaration

```
1)iaArray[3]={Wert, Wert, Wert}

2) iaArray [0]=Wert;
   iaArray [1]=Wert;
           iaArray [2]=Wert;

3) for(i=0; i<3;i++)
        {
        iaArray [i]=Wert;
        }
```

12.1.3. Initialisierung

```
1) iaArray[3]={Wert, Wert, Wert}

2) iaArray [0]=Wert;
   iaArray [1]=Wert;
   iaArray [2]=Wert;

3) for(i=0; i<3;i++)
   {
        iaArray [i]=Wert;
   }
```

12.2. Zweidimensionale Felder

12.2.1. Syntax

Datentyp *Name* [Größe 1] [Größe 2]

12.2.2. Deklaration

```
1) iaArray [3] [3] = {
  {3,4,5}
          {6,7,8}
          {3,5,7}
          }

2) for(i=0; i<3;i++)
   {
    for(j=0;j<3;j++)
    {
```

```
iaArray[i] [j]=0;
  }
}
```

12.2.3. Initialisierung

```
1) iaArray [3] [3] = {
                       {3,4,5}
                       {6,7,8}
                       {3,5,7}
                      }
2) for(i=0; i<3;i++)
   {
    for(j=0;j<3;j++)
    {
      iaArray[i] [j]=0;
    }
   }
```

13. Strings (Zeichenketten)

Strings benutzt man um einzelne Zeichen oder Wörter zu speichern. Der Datentyp ist natürlich Charakter und jeder String wird mit der Binären Null abgeschlossen. Die Binäre Null sieht so aus: '/0' und der Computer benötigt sie, damit dieser weiß, wann der jeweilige String zu Ende ist. Außerdem ermöglicht die Binäre Null die Ermittlung der Länge eines Strings mit Hilfe einer Schleife. Die Stringvariable sieht so aus: sString [13] . Die eckigen Klammern können auch leer gelassen werden, dann ordnet der Computer die benötigte Größe des Strings automatisch zu. Dabei muss der Wert des Strings aber schon bei der Deklaration initialisiert werden. Achtung bei der Deklaration muss man darauf achten, dass die Binäre Null auch ein Feld benötigt.

13.1. Deklaration

1) char sString[13];
2) char sString[]={’Z’, ’e’, ’i’, ’c’, ’h’, ’e’, ’n’, ’k’, ’e’, ’t’,’t’,’e’,’\0’,}
3) char sString[]=“Zeichenkette“;

13.2. Initialisierung

1) sString[13]={’Z’, ’e’, ’i’, ’c’, ’h’, ’e’, ’n’, ’k’, ’e’,’t’, ’t’, ’e’, ’\0’,};
2) sString[13]=“Zeichenkette“;

13.3. Ein- und Ausgabe

Liest man einen String über den Befehl scanf ein werden alle Zeichen nach einem Leerzeichen ignoriert und gehen verloren. Über gets kann man aber auch Leerzeichen einlesen.

gets (Stringname);	Bei gets werden alle Zeichen eingelesen. Eine Bestätigung mit Enter ist erforderlich.
puts (Stringname);	Puts gibt einen String am Bildschirm aus.

13.4. Stringkonstanten

Bei Stringkonstanten wird der Wert bei der Deklaration angegeben. Eine Stringkonstante ist innerhalb eines Programms nicht mehr veränderbar.

char* Name="Wert";

13.5. Schreibweisen

char *Name[4];

Deklariert man einen String auf diese Art, hängt er die eingegebenen Werte im Speicher aneinander. Diese Methode spart viel Speicherplatz.

char Name [4] [20];

Bei dieser Schreibweise wird ein 2-dimensionales Array deklariert. Dabei werden 4 Zeilen zu je 20 Zeichen reserviert. [4] geht also in die y-Richtung während [20] in die x-Richtung geht.

13.6. Stringmanipulation

Unter Stringmanipulation versteht man das Kopieren eines Strings, das Vergleichen eines Strings oder ähnliches. Alle diese Funktionen findet man in der Bibliothek: string.h .

13.6.1. Strings kopieren

strcpy(Adresse des neuen Strings, Adresse des alten Strings);

strncpy(Adresse des neuen Strings, Adresse des alten Strings, Anzahl der Zeichen in int);

13.6.2. Strings anhängen

strcat(Adresse des ersten Strings, Adresse des anzuhängenden Strings);

13.6.3. Strings vergleichen

Rückgabewert vom Typ int = strcmp(Adresse des 1.Strings, Adresse des 2.Strings);

Beim Vergleichen von zwei Strings können folgende Rückgabewerte zurückgegeben werden:

<0	1. String ist kleiner als der 2. String
>0	1. String ist größer als der 2. String
=0	1. String und 2. String sind identisch

13.6.4. Stringlänge ermitteln

Rückgabewert vom Typ int = strlen(Adresse des Strings);

13.6.5. Strings auf 0 initialisieren

Stringname[0]='\0';

14. Pointer (Zeiger)

Ein Zeiger ist eine Variable die eine Adresse speichert. Zeiger benötigt man um direkt auf den Adress- oder Datenteil des Speichers zuzugreifen. Schreibt man vor eine Zeigervariable einen Pointer „*", so greift man auf den Datenteil zu. Schreibt man

keinen Pointer davor, greift man auf den Adressteil zu. Die Formatspezifikation für einen Pointer ist %p.

Beispiel:

piVariable // Adresse
*piVariable // Daten

◄1000	5	i
1001	8	
1002	10	
1003	12	

```
int i, *pi;        //Variable und Pointer *pi deklarieren
pi= &i;            //pi wird die Adresse von i zugewiesen
pi++;              //im Speicher wird um die Größe des
                     Datentyps weitergegangen (hier 2 Byte)
printf(„%p“, pi);  //Ausgabe    1002
printf(„%d“,*pi);  //Ausgabe    10
```

14.1. Deklaration

Datentyp *Variablenname

Beispiel: int *piZeiger

14.2. Initialisierung

*Variablenname = Wert;

Beispiel: *piZeiger = 0;

14.3. Zeiger auf 0 initialisieren

piZeiger = NULL;

14.4. Zeigerarithmetik

„++“, „--“

addieren von Zeigern

Das Addieren von Zeigern ist nur sinnvoll, wenn beide Zeiger auf das gleiche Array zeigen.

subtrahieren von Zeigern

Das Subtrahieren von Zeigern ist nur sinnvoll, wenn beide Zeiger auf das gleiche Array zeigen.

„>“, „<“, „>=“, „<=“, „!=“, „==“,

Diese Operatoren sind nur bei Arrays sinnvoll.

14.5. Adressen eines Arrays ermitteln

a[i] [j]= Anfangsadresse + i * Elementzahl der 2. Dimension * Größe des Datentyps eines Arrayelements (in Byte) + j * Größe des Datentyps eines Arrayelements (in Byte)

1 .Dimension in y-Richtung: 1.Index[i]
2. Dimension in x-Richtung: 2.Index[j]

```
a[0] [0] = 5 ;    →    *(*(a))=5 ;
a[0] [1] = 6 ;    →    *(*(a)+1)=6 ;
a[1] [0] = 7 ;    →    *(*(a+1))=7 ;
a[1] [1] = 8 ;    →    *(*(a+1)+1)=8 ;
```

14.6. Zeiger auf einen String

piZeiger = Stringname;

15. Dynamische Felder

Dynamische Felder benutzt man um Speicherplatz variabel zu halten. Im Gegensatz zu einem statischen Feld (int ia[5]) benötigt man für dynamische Felder eigene Befehle. Diese Befehle findet man in der Bibliothek: stdlib.h . Schlägt die Speicherreservierung fehl kommt ein Pointer auf NULL zurück.

- Pointer auf Speicherblock= **malloc** [Größe des Speicherblocks in Byte]
  Der Befehl malloc reserviert einen Speicherblock und gibt die Anfangsadresse als Pointer zurück.
- Pointer auf Speicherblock = **calloc** [Anzahl der Elemente, Größe eines Elementes in Byte]
  Der Befehl calloc reserviert einen Speicherblock und gibt die Anfangsadresse als Pointer zurück und initialisiert jedes Element mit 0.
- Pointer auf Speicherblock = **realloc** [Pointer von malloc oder calloc, neue Größe des Speicherblockes in Bytes]
  Der Befehl realloc verändert die Speichergröße eines Speicherblockes.
- **free** (Pointer auf Speicherblock)
  Der Befehl free gibt den zuerst reservierten Speicher wieder frei.

Dynamische Speicherreservierung:

```
Deklaration:     Zeiger = malloc (sizeof(Datentyp));
Initialisierung:                 * Zeiger =0;
  oder           Zeiger [0]=0;
Einlesen:        for(int i; …)
  {
  scanf(„%d“, *( Zeiger +i))
          oder
  scanf(„%d“, Zeiger [i]);
}
```

Realloc: Zeiger = realloc (Zeiger, (i+1)*sizeof(Datentyp));

16. Argumente der Kommandozeile

Argumente verwendet man, um einem Programm beim Starten im DOS-Modus Wörter mitzugeben. Diese können dann später im Programm aufgerufen und ausgegeben werden, sind aber nicht veränderbar. Das 0. Element ist der Pfad und Dateiname.

Aufruf im DOS-Modus: Programmname Argument 1 Argument n

Um diese Befehle nutzen zu können, benötigt man die Standardbibliothek stdio.h .

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

  int argc           //Anzahl der eingegebenen Elemente
  char*argv[]        //Elemente
```

```
char* argv="Argument";   //Diese beiden Argumente sind
char* argv[];            //Stringkonstanten und somit nicht veränderbar.
```

17. Funktion

Funktionen sind Hilfsmittel, um Probleme in kleine Teilprobleme zerlegen zu können. Sie dienen damit einer strukturierten Programmierung. Eine typische Anwendung für Funktionen ist die Erledigung immer wieder kommender Aufgaben, innerhalb eines Programms. Für die wichtigsten Aufgaben gibt es bereits vordefinierte Funktionen, wie zum Beispiel printf oder scanf. Diese Funktionen sind in den C-Standardbibliotheken bereits ausprogrammiert. Eine Funktion gibt normalerweise einen Wert über den Befehl "return x;" zurück. Gibt eine Funktion keinen Wert zurück benötigt man diesen Befehl nicht, muss aber beim Rückgabewert "void" angeben. Das selbe gilt für die Übergabeparameter. Gibt man keine Übergabeparameter mit trägt man ebenfalls "void" ein.

17.1. Syntax

```
Standardbibliotheken

Funktionsprototypen                 // Vorwärtsdeklaration der Funktion

main ( )
{
      Funktionsaufruf
}
Funktionsdefinitionen               // Die Funktion wird ausprogrammiert
```

17.2. Funktionsprototyp

Der Funktionsprototyp wird benötigt um dem Compiler mitzuteilen, dass es eine Funktion gibt und welche Variablen diese Funktion verwendet. Dieses Mitteilen an den Compiler nennt man Vorwärtsdeklaration.

```
Rückgabedatentyp Funktionsname (Übergabeparameter);

z.B.: int Beispiel(int, double);
```

17.3. Funktionsaufruf

Der Funktionsaufruf befindet sich innerhalb der main-Funktion und dient zum Ausführen der Funktion.

```
Funktionsname (Übergabeparameter);

z.B.:   Beispiel (iA, dB);
        oder
        iErgebniss=Beispiel (iA, dB);
```

17.4. Funktionsdefinition

Der Funktionsdefinition dient zur Ausprogrammierung der Funktion. Erst am Ende eines Programmes werden die verwendeten Funktionen ausprogrammiert.

```
Rückgabedatentyp Funktionsname (Übergabeparameter)
{
```

```
        lokale Variablen
        Anweisungen

        return (Rückgabewerte);
}

z.B.:   int Beispiel (int iA, double dB);
        {
          int iSumme;
          iSumme= iA + (int) dB;

          return(iSumme);
        }
```

17.5. Call by Value

Unter Call by Value versteht man das Übergeben von Werten an eine Funktion. Dabei können die Werte in der Funktion verändert werden, ohne, dass man in der Hauptfunktion davon etwas merkt. Gibt man die Werte zurück, verändert man sie auch in der Hauptfunktion.

17.6. Call by Reference

Unter Call by Reference versteht man das Übergeben von Pointer an eine Funktion. Verändert man in der Funktion jetzt den Wert dieser Variablen, macht sich das auch im Hauptprogramm bemerkbar. Das heißt, die Funktion greift über Pointer direkt auf den Speicher der Variable zu und verändert diese somit.

```
Funktionsprototyp:
            Rückgabedatentyp Name(Datentyp *);

Funktionsaufruf:
            Name (&Variable);

Funktionsdefinition:
            Rückgabedatentyp Name (Datentyp *Pointer);
```

17.7. Funktionszeiger

Um eine Funktion durch das Beenden einer anderen Funktion aufzurufen übergibt man Funktionszeiger. Die Funktionszeiger werden dann in der entsprechenden Funktion in das return() eingebunden und rufen dann die gewünschte Funktion auf.

```
Funktionsprototyp:
            Rückgabedatentyp Name (Datentyp (*Name der Funktion)
                        (Datentyp der Übergabevariable));

Funktionsaufruf:
            Name (&Funktionsname);

Funktionsdefinition:
            Rückgabedatentyp Name (Datentyp (*Name der Funktion)
                        (Datentyp der Übergabevariable));
```

Funktionsaufruf über den Funktionszeiger:
return (Name der Funktion (Variable));

17.8. Array als Übergabeparameter

Da der Name eines Feldes ein Zeiger auf das 1. Element des Feldes ist (Adresse des 1. Elementes) wird bei Feldern immer eine Adresse übergeben Das bedeutet, dass die Funktion mit dem Originalfeld arbeitet. Also ist das ganze Call by Reference. Call by Value ist bei Feldern nicht möglich.

Funktionsprototyp:
Rückgabedatentyp Name (Datentyp []);
oder
Rückgabedatentyp Name (Datentyp*);

Funktionsaufruf:
Name (& Arrayname[]);
oder
Name (Arrayname);

Funktionsdefinition:
Rückgabedatentyp Name (Datentyp Arrayname[])
oder
Rückgabedatentyp Name (Datentyp *Arrayname)

18. Suchen und Sortieren

Das Suchen in Datenbeständen ist ein häufiges und immer wieder vorkommendes Programmierproblem. Eng damit verbunden ist auch das Sortieren, denn normalerweise sortiert man Datenbestände um ein späteres Suchen zu erleichtern. Beim Suchen muss man sich im Klaren sein, welches Feld als Such- und welches als Sortierschlüssel dienen soll.

18.1. Suchen

18.1.1. Lineare Suche

Bei der Linearen Suche ist die Tabelle nicht sortiert und es bleibt daher nichts anderes übrig als von vorne bis hinten alles zu durchsuchen. Bei einer Feldgröße n benötigt man im Mittel n/2 Suchschritte. Im günstigsten Fall 1 Suchschritt, im schlechtesten Fall n Suchschritte.

Headerdatei:
```
#define TABLESIZE x
```

Code:
```
int Table [TABLESIZE +1]
void main (void)
{
    int iIndex, iKey;
    iIndex=iKey=0;

    printf("Geben Sie die Tabellenwerte ein");
```

```
            for(iIndex=0; iIndex<TABLESIZE; iIndex++)
            {
                    scanf("%d", &Table[iIndex]);
            }
            printf("Geben sie den Suchschlüssel (Key) ein");
            scanf("%d", &iKey);

            Table[TABLESIZE]= iKey;
            iIndex=0;
            while(Table[iIndex] != iKey)
            {
                    iIndex++;
            }
            if(iIndex == TABLESIZE)
            {
                    printf("Error");
            }
            printf("%d", iIndex);
    }
```

18.1.2. Binäre Suche

Um die Suche zu beschleunigen müssen die Elemente in der Tabelle sortiert vorliegen. Dies kann man durch ein Sortierverfahren (siehe: 24.3) erreichen oder in dem man die Werte sortiert in die Tabelle eingibt. Beides bedeutet mehr Aufwand, spart aber Zeit bei der Suche. Bei der Binären Suche sucht man den mittleren Wert der Tabelle und prüft, ob dieser größer oder kleiner ist als der Suchschlüssel (Key). Ist dieser kleiner oder gleich muss man nur noch in der unteren Hälfte suchen, ist dieser größer sucht man nur noch in der oberen Hälfte. Ist das Suchintervall auf 1 geschrumpft hat man das Element gefunden oder es ist nicht vorhanden. Im Mittel benötigt man bei der Binären Suche $lg_2 (n)$ Suchschritte.

Headerdatei:

```
#define TABLESIZE x
```

Code:

```
int Table [TABLESIZE]
void main (void)
  {
        int iRight, iLeft, iMiddle, iKey, iIndex;
        iRight=iLeft=iMiddle=iKey=iIndex=0;

        printf("Geben Sie die Tabellenwerte ein");
        for(iIndex=0; iIndex<TABLESIZE; iIndex++)
        {
                scanf("%d", &Table[iIndex]);
        }
        printf("Geben sie den Suchschlüssel (Key) ein");
        scanf("%d", &iKey);
```

```
            iLeft=1;
            iRight= TABLESIZE+1;
            while(iLeft<iRight)
            {
            iMiddle=(iRight+iLeft)/2;
            if(Table[iMiddle]<iKey)
            {
                    iLeft=iMiddle+1;
            }
            else
            {
                    iRight=iMiddle;
            }
            }
            if(Table[iRight] == iKey)
            {
                    printf("%d", iRight);
            }
            else
            {
                printf("Fehler, Suchelement wurde nicht gefunden");
            }
    }
```

18.2. Sortieren

18.2.1. Bubblesort

Bubblesort ist einer der einfachsten Sortieralgorithmen. Im ersten Durchgang wird das Array vom Anfang bis zum Ende bearbeitet und bei jedem Schritt die aktuelle Komponente mit der nächsten verglichen. Ist die untere Komponente kleiner als die obere Komponente werden die beiden Komponenten vertauscht. Die größere Komponente behält also ihren Platz und die jeweils kleinere Komponente wandert nach oben. Im nächsten Durchlauf wird die zweitkleinste Komponente gesucht und nach oben durchgereiht. Logischerweise muss dieser Durchlauf nicht bis zum Anfang gehen, sondern nur bis zum ersten Index. Bubblesort ist sehr langsam und wird daher in der Praxis nicht eingesetzt.

Headerdatei:

```
    #define TABLESIZE x
```

Code:

```
    int Table [TABLESIZE]
    void main (void)
      {
            int iIndex1, iIndex2, iTemp;
            iIndex1=iIndex2=iTemp=0;
            for(iIndex1=0; iIndex1<TABLESIZE; iIndex1++)
            {
```

```
                for(iIndex2=TABLESIZE-1; iIndex2>iIndex1;
                iIndex2--)
                {
                        if(Table[iIndex2]<Table[iIndex2-1])
                        {
                                iTemp=Table[iIndex2];
                                Table[iIndex2]=Table[iIndex2-1];
                                Table[iIndex2-1]=iTemp;
                        }
                }
        }
}
```

18.2.2. Quicksort

Quicksort ist in den meisten Fällen der schnellste Algorithmus. Man greift sich eine beliebige Komponente des Arrays heraus, beispielsweise die Mittlere, und teilt das Array in zwei Gruppen auf. Eine Gruppe mit den Komponenten die größer sind als die Mittlere und eine die kleiner gleich sind als die Mittlere. Diese beiden Gruppen übergibt man an Quicksort. Dies geschieht so oft bis das Array nur noch 1 Element groß ist. Daraus folgt, dass das Array sortiert ist. Das heißt die Funktion Quicksort ruft sich immer wieder selbst auf, dies nennt man einen rekursiven Aufruf.

Quellcode:

```
void Qsort (int, int, int*)

void main (void)
{
        :
}

void Qsort (int iLinks, int iRechts, int* Table)
{
        int i, j, x, temp;
        i=iLinks;
        j=iRechts;
        x=Table[(i+j)/2];

        do
        {
                while (Table[i]<x)  i++;
                while (Table[j]>x)  j--;

                if(i<=j)
                {
                        temp=Table[i];
                        Table[i]=Table[j];
                        Table[j]=temp;
                        i++;
```

```
                j--;
            }
        }
        while (i<=j);

        if (iLinks<j)   Qsort(iLinks, j, Table);
        if (iRechts>i)  Qsort (i, iRechts, Table);
}
```

18.3. Standardsuch- und Sortierfunktionen

18.3.1. Binäre Suche

Funktionsname: bsearch()

Funktionsprototyp: void* bsearch(const void* key, const void* base, size_t num, size_t size, int(*cmp)(void* elem1, void* elem2));

key	Zeiger auf das gesuchte Element
base	Zeiger auf das 1. Element des Arrays
num	Anzahl der Komponenten des Arrays
size	Größe einer Komponente des Arrays in Byte
cmp	Zeiger auf die Vergleichsfunktion
void*	Dieser Ausdruck wird verwendet, wenn man vorher nicht weiß welcher Datentyp verwendet wird. Während der Programmausführung wird der verwendete Datentyp ermittelt und an die Stelle des void* gesetzt.
const	Const (=Konstante) bedeutet, dass die Zeiger, hier „base“ und „key“, nicht verändert werden können.

Rückgabewert der Funktion cmp:

<0	Parameter 1 ist kleiner als Parameter 2
=0	Parameter 1 ist gleich Parameter 2
>0	Parameter 1 ist größer als Parameter 2

Rückgabewert der Funktion bsearch:
Falls das gesuchte Element gefunden wird, wird ein Zeiger auf dieses zurückgegeben. Ansonsten wird der NULL Zeiger zurückgegeben.

num = sizeof (Array) / sizeof (Array[0]);
size = sizeof (Array [0]);

18.3.2. Quicksort

Funktionsprototyp:
void qsort(void* base, size_t num, size_t size, int(*cmp)(void* elem1, void* elem2));

base	Zeiger auf das 1. Element des Arrays
num	Anzahl der Komponenten des Arrays
size	Größe einer Komponente des Arrays in Byte
cmp	Zeiger auf die Vergleichsfunktion

19. Speicherklassen

Speicherklassen werden verwendet um Variablen ein Verhalten im Speicher zuzuweisen.

Syntax:

Speicherklasse Datentyp Variablenname;

Speicherklassen:

auto	Auto-Variablen sind „normale" lokale Variablen. Die auto-Speicherklasse wird auch Standard- oder Default-Speicherklasse genannt. Allen Variablen denen nicht explizit, also ausdrücklich, eine Speicherklasse zugewiesen wird, und die nicht außerhalb von Funktionen vereinbart werden, wird automatisch die Speicherklasse auto zugewiesen. Automatische Variablen werden bei jedem Funktionsaufruf neu erzeugt und beim Verlassen der Funktion wieder zerstört. Daher ist der Geltungsbereich auf die lokale Funktion, in der die Variable vereinbart wurde, beschränkt.
register	Diese Variablen werden direkt im CPU-Register gespeichert. Falls kein Register mehr zu Verfügung steht, wird die Variable wie vom Typ auto behandelt. Register-Variablen sind entweder vom Typ int oder vom Typ char.
extern	Externe Speicherklassen sind die Lokale Deklaration von globalen Variablen. Alle Objekte die außerhalb von Funktionen vereinbart werden sind von der Speicherklasse extern. Auch Funktionsnamen sind von der Speicherklasse extern. Externe Objekte sind ab ihrer Deklaration bis zum Ende des Quellcodes und in anderen Quellcodedateien bekannt und verfügbar.
static	Bei statischen (static) Variablen bleibt der Inhalt der Variable, zwischen Funktionsaufrufen, bekannt. Static-Variablen können sowohl interne als auch externe Variablen sein. Der Compiler ergänzt zu static automatisch die Schlüsselwörter auto, register oder extern. Es gibt natürlich auch reine static-Variablen. Statische Objekte innerhalb von Funktionen sind nur lokal bekannt. Behalten aber im Gegensatz zu auto-Objekten ihre Werte zwischen den Funktionsaufrufen bei.

20. Dateien

Dateien werden für die permanente Speicherung von Daten bzw. zur Eingabe und Ausgabe von Daten benötigt. Es gibt transiente Daten, diese sind nach dem Ausschalten des Computers weg, da sie nur im Arbeitsspeicher gespeichert werden. Es gibt aber auch persistente Daten, dies sind gespeicherte Daten in Dateien und Datenbanken (auf Festplatte, CD, usw.), diese bleiben auch nach dem Ausschalten des Computers erhalten. Dateien bestehen aus beliebig vielen Komponenten eines bestimmten Datentyps. Am Ende einer Datei können weitere Komponenten angefügt werden. Die Abbildung der abstrakten Dateistrukturen auf reale Speichermedien (Festplatte, CD, usw.) erfolgt durch das Betriebssystem.

20.1. Standardoperationen auf Dateien

Open	Eröffnen einer Datei zum Lesen einer Datei oder zum Anfügen neuer Komponenten einer Datei.
Close	Beenden des Zugriffs auf eine Datei.
Read, Write	Auswahl ob die Datei gelesen oder beschrieben werden soll.

20.2. Standard Ein- und Ausgabe

Ausgabe	Über den Bildschirm mit stdout für die normale Ausgabe oder mit stderr für die Fehlerausgabe am Bildschirm
Eingabe	Über die Tastatur mit stdin

Um anzugeben welches Gerät angesprochen werden soll, muss ein File - Pointer deklariert werden.

20.3. Textdatei

Textdateien sind Dateien der Komponenten aus Schriftzeichen-Strings bestehen.

- fprintf("…");
- fscanf("…");
- int fputc(int, FILE*);
  Schreibt 1 Zeichen in eine Datei, wobei int eine Ascii-Code ist
- int fgetc(FILE*);
  Liest 1 Zeichen aus einer Datei
- int fputs (const char*, FILE*);
  Schreibt einen String in eine Datei
- fgets char*, int, FILE*);
  Liest einen String aus einer Datei

20.4. Binärdatei

Bei einer Binärdatei werden die Daten in einer nicht lesbaren Form abgelegt. Die Daten werden byteweise in die Datei übertragen.

- Binärdatei schreiben
  size_t fwrite(const void*, size_t, size_t, FILE*);

const void*	Adresse des ersten Datenelements
size_t	Größe eines einzelnen Datenelements in Byte
size_t	Anzahl der Datenelemente
FILE*	Filepointer auf Datei

- Binärdatei lesen
  size_t fwrite(const void*, size_t, size_t, FILE*);

const void*	Adresse des ersten Datenelements
size_t	Größe eines einzelnen Datenelements in Byte

size_t Anzahl der Datenelemente
FILE* Filepointer auf Datei

20.5. Beschreiben und Lesen von Dateien

20.5.1. Definieren einer Datei

Syntax: FILE* <Dateizeiger>

FILE ist eine spezielle Stream-Struktur, der Datentyp für Dateien. Er ist in der Standardbibliothek <stdio.h> als Struktur festgelegt, die Informationen über die Datei enthält. Mit obiger Syntax wird ein Zeiger auf den Datentyp FILE definiert.

20.5.2. Öffnen einer Datei

Syntax: <Dateizeiger> = fopen(<Dateiname>,
<Zugriffsmodus>);

Die Funktion fopen verbindet den externen Namen der Datei mit dem Programm und liefert als Ergebnis den Zeiger auf die Beschreibung der Datei. Im Fehlerfall wird der NULL-Zeiger zurückgeliefert. Die Funktion ist definiert als:
FILE *fopen(const char * filename, const char *modus)

Als Zugriffsmodus steht eine Kombination von „a“, „r“, „w“ und „+“:

- r Lesen (read)
- w Schreiben (write)
- a Anhängen (append)
- r+ Lesen und Schreiben
- w+ Schreiben und Lesen
- a+ Lesen an beliebigen Positionen, Schreiben am Dateiende

Durch Anhängen eines Zusatzes kann festgelegt werden, ob es sich bei der zu bearbeitenden Datei um eine Binär- oder Textdatei handelt:

- t für Textdatei
- b für Binärdatei

Die Funktion fopen reagiert folgendermaßen:

- Beim Öffen einer existierenden Datei:
  - Zum Lesen: keine Probleme
  - Zum Anhängen: keine Probleme
  - Zum Schreiben: Inhalt der Datei geht verloren
- Beim Öffnen einer nicht existierenden Datei:
  - Zum Lesen: Fehler, Ergebnis ist NULL-Zeiger
  - Zum Anhängen: neue Datei wird angelegt
  - Zum Schreiben: neue Datei wird angelegt

Maximal fopen_max Dateien können gleichzeitig geöffnet werden, maximale Dateinamenlänge filename_max.

Zwischen Lesen und Schreiben ist ein Aufruf von fflush() oder ein Positionierungsvorgang nötig.

20.5.3. Formatierte Ausgabe auf Datei

Syntax: fprintf(<Dateizeiger>, "<Format>", <Werte>);

Entspricht der Funktion „printf" und schreibt die angegebenen Werte im angegebenen Format auf die Datei. Der Dateizeiger verweist auf die Datei, auf die geschrieben wird. „fprintf" ist definiert als:

int fprintf(FILE*, const char *format, …)

20.5.4. Formatierte Eingabe von Datei

Syntax: fscanf(<Dateizeiger>, "<Format>", <Werte>);

Entspricht der Funktion „scanf" und liest die angegebenen Werte im vereinbarten Format der Datei ein. „fscanf" ist definiert als:

int fscanf (FILE*, const char *format, …)

20.5.5. Dateiende abfragen

Syntax: feof(<dateizeiger>);

Die Funktion „feof" liefert den Integerwert 1, wenn das Dateiende gelesen wurde, sonst 0. „feof" ist definiert als:

int feof (FILE*)

20.5.6. Schließen einer Datei

Syntax: fclose(<Dateizeiger>);

Die Datei wird geschlossen, vom Programm abgehängt und der Platz für den Filebeschreibungsblock wieder freigegeben. Beim Schreiben auf Dateien sollte die Datei geschlossen werden, sobald alle Schreiboperationen abgeschlossen sind, da erst beim Schließen die Dateipuffer auf die Platte geschrieben und die Informationen über die Datei in der Dateiverwaltung aktualisiert werden. „fclose" ist definiert als:

int fclose (FILE*)

20.5.7. Beispiel

```
#include <stdio.h>

int main (void)
{
        fILE *fp;
        int i, xi;
        static char dateiname[]="daten.datei";
        char text[80];
```

```
// Beschreiben der Datei
fp=fopen(dateiname,“w“);
if(fp==NULL)
{
        fprintf(stderr, “Datei %s kann nicht zum
                        Schreiben geoeffnet werden\n“,
                        dateiname);
        exit(1);
}

for(i=0; i<10; i=i+2)
{
        fprintf(fp, “%d\n“, i);
}
fclose(fp)                      // Schließen der Datei nicht
                                   vergessen!

// Lesen der Datei
fp=fopen(„dateiname“, „“r“);
if(fp==NULL)
{
        fprintf(stderr, “Datei %s kann nicht zum Lesen
                        geoeffnet werden\n“, dateiname);
        exit(2);
}

while(feof(fp) == 0)
{
        fscanf(fp, „“%d“, &xi);
        printf(„“%d“, xi);
}
fclose (fp);                    // Schließen der Datei nicht
                                   vergessen!
exit(0);
}
```

21. Strukturen

Ein Array ist eine Zusammenfassung von Objekten gleichen Typs. Bei Strukturen handelt es sich um eine Zusammenfassung von Objekten möglicherweise verschiedenen Typs zu einer Einheit. Die Verwendung von Strukturen bietet Vorteile bei der Organisation komplexer Daten (Bsp.: Telefonliste -> Es werden Vorname, Nachname, Adresse, Telefonnummer zu einer Einheit zusammengefügt.)

21.1. Deklaration

```
Struct Strukturname
{
      Datentyp Variablenname;
};
```

Bsp.:

```
Struct Telefonliste
{
        char Vorname [40];
        char Nachname [40];
        int Telefonnummer;
};
```

21.2. Initialisierung

Bsp.:

```
Struct Datum
{
        int Tag;
        int Monat;
        int Jahr;
        char Monatsname [4];
};

struct Datum heute={12, 5, 2003, "Mai"};
```

Für den Elementzugriff gibt es zwei eigene Operatoren. Der direkt Zugriff wird dabei mit dem „.“-Operator durchgeführt.

```
Heute.Tag=12;
Heute.Monat=5;
Heute.Jahr=2003;
Heute.Monatsname="Mai";
```

21.3. Typedef

Mit typedef kann man neue Datentypnamen definieren. Nicht aber neue Datentypen. Typdef ist dem „# define“ ähnlich , aber weitergehend. Da der Befehl typedefm erst vom Compiler verarbeitet wird und nicht einfacher Textersatz ist. Die Anwendungen von Typedef liegen darin, ein Programm gegen Portabilitätsprobleme abzuschirmen und für eine bessere interne Dokumentation zu sorgen. Durch typedef wird aber auch der Aufbau von Strukturen verschleiert, sodass viele Programmierer keinen Gebrach davon machen.

Syntax: typdef Datentyp Aliasname;

Beispiel: typedef unsigned long int bigint;

21.4. Union

Während eine Struktur mehrere Variablen verschiedenen Typs enthält, ist die Union eine Variante einer Struktur, die ebenfalls Variablen verschiedenen Typs speichern kann, aber nicht gleichzeitig. Verschiedene Arten von Datenobjekten können so in einem einzigen Speicherbereich maschinenunabhängig manipuliert werden. Syntaktisch sind union und struct gleich. Der wesentliche Unterschied ist, dass eine Variable vom Typ Union zu einer Zeit immer nur eines der angegebenen Elemente enthalten kann.

Syntax:

```
union Name
{
        Variablen;
};
```

21.5. Bitfelder

Strukturen können auch Elemente haben, die vom Typ signed oder unsigned int, aber nur wenige Bit lang sind. Solche Elemente bezeichnet man als Bitfelder.

22. Komplexe Deklarationen

Komplexe Deklarationen sind manchmal schwer aufzulösen. Dafür gibt es aber ein Rezept welches wir vereinfacht benutzen werden. Man löst die Deklaration vom Namen her auf und schaut dann nach rechts und nach links in Abhängigkeit vom Vorrang der Syntaxelemente. Beginnend bei wichtigen Elementen ist die Reihenfolge folgende:

- Runde Klammer(),die Teile der Deklaration zusammenfasst oder gruppiert
- Runde Klammer(), die eine Funktion anzeigt
- Eckige Klammer[], die ein Array oder Feld anzeigt
- Sternsymbol *, das einen Zeiger anzeigt

Beispiel:

```
Char*(*(*seltsam) (double, int) [3];
```

*seltsam	seltsam ist ein Zeiger.
*seltsam(double, int	seltsam ist ein Zeiger auf eine Funktion mit den Argumenten double und int.
*(*seltsam)(double, int)	seltsam ist ein Zeiger auf eine Funktion mit den Argumenten double und int, welche einen Zeiger zurückgibt.
char*(*(*seltsam) (double, int) [3];	seltsam ist ein Zeiger auf eine Funktion mit den Argumenten double und int, welcher einen Zeiger zurückgibt auf ein Feld mit der Größe 3 dessen Elemente vom Typ char sind.

Komplizierte Deklarationen sollten der Lesbarkeit halber vermieden werden. Manchmal sind Sie aber unumgänglich. Für diesen Fall bietet das Schlüsselwort typedef die Möglichkeit die Lesbarkeit durch Strukturierung der Namensgebung zu verbessern.

Beispiel:

```
typedef char* Arrayvon3charZeigern [3];
Arrayvon3charZeigern A; = char* A [3];
```

23. Wichtige Befehle

23.1. Der exit-Status

Wenn ein Programm seinen Ablauf beendet hat, sollte es dem System mitteilen, auf welche Weise diese Beendigung erfolgt ist. Dazu dient der exit-Status. Der exit-Status ist eine ganze Zahl, die im System unmittelbar nach Programmbeendigung in

irgendeiner Form zurückbleibt. Die Beachtung des exit-Status wird spätestens dann wichtig, wenn Programme sich gegenseitig aufrufen und abhängig vom Ausgang Entscheidungen treffen müssen. Der exit-Status ist ein Rückgabewert der Funktion main. Also im Allgemeinen der Wert der in main mit Return zurückgegeben wird.

```
exit(0);        Alles OK !
exit(>0);       Fehler !
```

23.2. Bildschirminhalt löschen

system(„cls“); aus stdlib.h löscht den aktuellen Bildschirminhalt. Ideal für Menüstrukturen mit vielen Untermenüs.

23.3. Tastaturpuffer leeren

fflush(stdin); aus stdio.h leert den Tastaturpuffer. Ideal vor Abfragen auf Enter.

24. Wichtige Algorithmen

24.1. Vertauschalgorithmus

Den Vertauschalgorithmus benötigt man um die Werte zweier Variablen zu vertauschen. Dafür benötigt man aber eine Hilfsvariable temp.

```
int a=5;            // Variable 1
int b=6;            // Variable 2
int temp=0;         // Hilfsvariable

temp=a;             // a wird auf temp gelegt
a=b                 // a wird der Wert b zugewiesen
b=temp              // b wird der Wert temp zugewiesen also der
                    // Wert vom ehemaligen a
```

24.2. Vergleichsalgorithmus

Den Vergleichsalgorithmus benötigt man, um die Werte zweier Variablen zu vergleichen und dann auszugeben, welcher Wert welcher Variable größer ist bzw. ob die Werte beider Variablen gleich groß sind

```
if (a<b)
        {
        printf(a);
        }
else if (a>b)
        {
        printf(b);
        }
else
        {
        printf(a=b);
        }
```

24.3. Summenalgorithmus

Der Summenalgorithmus wird verwendet, um die Summe aller Zahlen von einem gewissen Zahlenbereich zu bilden.

```
int iSumme;
iSumme=0;

for(int i=y; i<=x; i++)  //y Anfangswert, x Endwert der Summenbildung
  {
  iSumme= iSumme +i;
  }
printf("%d",iSumme);
```

24.4. Erzeugen von Zufallszahlen

Um Zufallszahlen zu generieren benötigt man Funktionen aus der Bibliothek time.h . Außerdem benötigt man die Systemzeit und den Datentyp time_t sowie die Funktionen srand( ) und rand( ).

```
#include<time.h>
#include<stdio.h>

void main (void)
        {
        int iWert;
        iWert=0;

        time_t tZeit;                  // Datentyp: time_t
        time(&tZeit);                  // Lesen der Systemzeit
        srand((unsigned int)tZeit);    // generieren der Zufallszahlen

        iWert= rand() %6+1             // Ausgeben von Zufallszahlen
        printf("%d", iWert);           // zwischen 6 und 1
        }
```