Algorithmen und Datenstrukturen – Programmieren
Zusammenfassung

Prof. Dr. H. Kristl/ Prof. Dr. Thomas Rainer

von Matthias Markthaler

Kapitel 1: Daten und Algorithmen
1.1 Information, Nachricht, Daten

ASCII-Code = American Standard Code for Information Interchange

Escape Sequenzen (Graphisch nicht darstellbare Zeichen)

Zeichen	Darstellung	Zeichen	Darstellung
New Line	\n	Backslash	\\
Carriage Return	\r	Question Mark	\?
Form Feed	\f	Single Quote	\‘
Back Space	\b	Double Quote	\“
Horizontal Tab	\t		
Vertical Tab	\v		
Bell	\a		

Darstellung derZahlen in C

Bezeichnung	Darstellungs-länge	Länge der Mantisse +VZ	kleinster Zahlenwert	größter Zahlenwert
Ganze Zahlen				
Int	16		-32768	+32767
unsigned int	16		0	65535
char	8		-128	+127
unsigned char	8		0	255
long	32		-2^{31}	$+2^{31}-1$
unsigned long	32		0	$2^{32}-1$
Gleitpunktzahl				
float	8	24	$1.17e^{-38}$	$3.40e^{+38}$
double	11	53	$2.22e^{-208}$	$1.79e^{+308}$
long double	15	64	$3.36e^{-4932}$	$1.18e^{+4932}$

1.2 Schritte der Softwareerstellung

Algorithmus (= Kuchenrezept): Die Eindeutige und vollständige Beschreibung des Weges, auf dem ein gewünschtes Resultat gegebenen Voraussetzungen durch eine endliche Anzahl von Verarbeitungsschritten erreicht wird.

- Anweisungen werden nacheinander ausgeführt (Sequenz) sofern nicht anders formuliert

Programm(= Kochbuch)**:** Implementierung eines Algorithmus auf einem Datenverarbeitungssystem. Es steuert den Ablauf von Verarbeitungsprozessen, in deren Verlauf Resultate erarbeitet werden.

- Jeder Schritt festgelegt sein (Stelle) und ausführbar sein
- Verfahren muss endliche Länge besitzen

Codierung: Übertragung der abstrakten Beschreibung des Algorithmus in die Maschinensprache meist unter Anwendung einer höheren Programmiersprache oder Assemblersparche .

Daten: Informationen, die zur Verarbeitung durch ein Datenverarbeitungssystem bestimmt sind oder/und durch eine solche Verarbeitung entstanden sind.

Software: Gesamtheit aller Programme für den Betrieb von Systemen, die auf programmgesteuerten Rechnerbaugruppen basieren.

Schritte zur Softwareentwicklung
Schritt1 : Konzeption und Modularisierung
Aufgabenstellung(Lastenheft, Pflichtenheft), Zerlegung in einzelne Teilprobleme, Herarchie
Schritt 2: Modultest und Integration
Schrittweise Modulintegration (Tests) → Programmtest
Schritt 3: Abnahme, Einsatz und Optimierung
Dauernutzung, Wartung, Erweiterungen

Modularisierung: Ein Modul (Teilalgorithmus) ist in sich abgeschlossenes und bearbeitet ein geschlossenes Teilproblem. (IN-Black Box-OUT) Es besteht aus mehreren Blöcken.

1.3 Entwicklung und Darstellung von Algorithmen

- Einzelne Aktionen werden stets nacheinander ausgeführt (Sequenz)
- Struktogramme

1.4 Algorithmische Sprachen

Assembler = Maschienenorientiert
Höhrere Sprachen = problemorientiert → erlauben ausschließlich Lösung gestellter Probleme

Syntax: Der Streng festgelegte Aufbau von zulässigen sprachlichen Konstrukten
Semantik: Regeln der Semantik ist, dass Sätze zusätzlich einen Sinngehalt erfüllen müssen
Schlüsselwörter: z.B. int, for, char,…

1.5 Datenstrukturen

Variable: Variable sind Daten, die im Programmverlauf ihren Wert verändern können. → **Konstante**
Datentyp: Umfang und wertebereich werden im Datentyp festgelegt. unsigned int (Datentyp) iZahl, iZahl2; (Variablenname)

Datentypen: einfach (ganze Zahlen), skalar (Gleitpunktzahlen), char (Textzeichen + ganze), bool (true !=0, false ==0)

Felder -arrays: feste Anzahl von Komponenten des gleichen Datentyps
Verbunde -structures: feste Anzahl von Komponenten unterschiedlichen Datentyps
Datei –file: besteht aus linearen Folge von Komponenten gleichen Typs, die Anzahl der Komponenten ist nicht festgelegt. (FILE dient speziell zum Zugriff auf Massenspeicher)

Kapitel 2: Programmstrukturen in C

2.1 Grundelemente eines C-Programms

Struktur eines C-Programms

Ein C-Programm besteht aus mehreren (Programm-)Modulen, die getrennt zu übersetzen sind.
main() ist das Hauptprogramm

Ein C-Modul besteht aus einer beliebigen Folge von

- Vereinbarungen: - Deklarationen
  (Eigenschaften eines Objekts: Typ, Größe, Speicherklasse…)
  - Typdefinitionen
  - Variablendefinitionen
  - Funktionsdefinitionen
- Prozessoranweisungen

Anweisungen: Realisieren die Programmstrukturen Abschluss jeder Vereinbarung mit " ; "
Ausnahme: Verbundanweisungen
Block: Besteht aus Deklarationen, Typ- , Variablendefinitionen und Anweisungen jedoch nicht aus Funktionsdefinitionen. Entweder als Funktionstrump (Anweisung) oder Verbundanweisung
Prozessoranweisungen: " # " gefolgt von einem Schlüsselwort für die Bearbeitung durch Präprozessor
Kommentare: /* … */ oder //

Anweisungen in C

- Ausdrucksanweisung: → [Ausdruck] → ; →
- Leeranweisung: → ; →
- Verbundanweisung: → { → ↑Vereinbarung↓ → [Anweisung] } → !!! kein ;!!!
- while -Anweisung: while (…) [Anweisung]
- do -Anweisung: do [Anweisung] while (…);
- for -Anweisung: for(…;…;…) [Anweisung]
- break -Anweisung: break; (Sprung aus switch-Anweisung oder Schleifen beenden)
- continue -Anweisung: continue; (Sprung zum Ende des gegenwärtigen Schleifendurchlaufs)

```
while (Bedingung) {Anweisung 1; if (Bedingung2) continue; Anweisung2;}
```

- ~~goto -Anweisung~~: goto [Marke]; (Marke muss innerhalb der gleichen Funktion liegen)
  Sprünge zulässig: über Blockgrenzen hinweg, Aus schleifen raus und rein !!Schlecht!!
- if -Anweisung: if (Ausdruck) [Anweisung]; else if [Anweisung]; else [Anweisung];
- switch –Anweisung: switch (Ausdrick){case ‚1':…; case ‚d':…; default:…}

Kapitel 3: Ausdrücke und Operatoren

3.1 Arithmetische Operatoren

Arithmetische Operatoren Binäre: +, -, *, /, %

Unäre: + Identität, - Negation

++ Inkrement (Erhöhung des Operanden um 1)

-- Dekrement (Verminderung des Operanden um 1)

Prefix-Operator ++ davor, Postfix ++ danach `i++`

Vergleichsoperatoren Binäre: <, >, <=, >=, ==, !=

Logische Operatoren == 0 entspricht False

!=0 entspricht true

Binäre: && bzw. (AND), || bzw. (OR)

Negation: ! konvertiert Operanden

```
if(iWort ==0) ist wie if(!iWort)
```

Priorität	Operator	Operation
↓	( ) [ ] . ->	Argumentenliste Element (Array) Element (Structure) Element (Structure
	~ ! ++ -- - + * & (type) sizeof	Bitweises Komplement Komplement Increment/Decrement Negation (arithm.) Identität (arithm.) Pointer-Objekt Adresse Typkonvertierung Grösse in Bytes
	* / %	Mult/ Div/ Modulo
	+ -	Addition/Subtraktion
	<< >>	Verschiebung
	< > >= =>	Vergleich
	&	Bitweises UND
	^	Bitweises EXOR
	|	Bitweises ODER
	&&	UND
	||	ODER
	?:	Bedingte Auswerung
	= *= /= …	Zuweisungen
	,	sequentielle Auswerung

Bit-Operatoren

Binäre: & bitweise UND, | bitweise ODER ,^ bitweise EXOR

Unäre: ~ bitweise Negation (Einer-Komplement)

```
Löschen aller höherwertigen Bits, die 7niedrigsten sollen da bleiben
       Int c,n; … c = n & 0x7F; …
Umwandlung von Klein- in Groß-
if ( c >= 'a'&& c<= 'z') c = c & 0xDF;
Groß- zu Kleinbuchstaben
if ( c >= 'A'&& c<= 'Z') c = c | 0x20;
```

Schiebe-Operatoren

<< Linksschieben entspricht einer mnultiplikation `int x; x = x<<2; mit 4`

>> Rechtsschieben entspricht einer division

```
int x; x = x>>3; mit 8
```

Zusammengesetzte Zuweisungsoperatoren

+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=

Kapitel 4: Unterprogramme – Funktionen

4.1 Konzept

Funktionen sind statisch nebeneinander angeordnet →eine Funktion in einer Funktion nicht möglich

Funktionsdefinition:

$$\underbrace{Speicherklasse}_{\substack{\text{static: nur im Modul verwendbar}\\ \text{extern: auch in anderen Modulen}\\ \text{oder gar nix}}} \quad \underbrace{Funktionstyp}_{\text{double}} \; \underbrace{Funktionsname}_{\text{dMachWas}} \; \underbrace{Parameterliste}_{\text{(int iPi,double dB)}} \; \underbrace{Funktionsrumpf}_{\{\dots\}}$$

- ! als Datentypen sind keine „array"-Typen zulässig!

Funktionsdeklaration: *Speicherklasse Funktionstyp Funktionsname Parameterliste;*

Funktionsaufruf: *Funktionsname (aktuelle Parameter);*

printf() („Formatstring",Argument1,…ArgumentN);

scanf() („Formatstring", $\underbrace{\&Argument1}_{\text{bzw.Adresse von Argument 1}}$,…) EOF wird bei Fehlern zurückgegeben

Ein-/Ausgabeformate

Allgemeine Form: *% [Flags] [Breite] [.Präzision] [Argumentengröße] Typ*

Flags - linksbündige Ausgabe
+ Ausgabe des „+"-Vorzeichens bei numerischen Daten
, , ein führendes Leerzeichen wird bei positiven Zahlen ausgegeben

Breite normal Nummer mit 0en aufgefüllt, wenn mit 0 beginnt. Sonst mit „blanks"
* Feldlänge verweist auf die Angabe als Parameter vor dem Ausgabeargument

Präzision .0 Der Dezimalpunkt & überflüssige Nullen werden nicht ausgegeben (bei Typen e,E,f)
.n Gleitkommawert wird mit n Nachkommastellen ausgegeben, ein String mit max n Stellen
.* Der nächste Ganzzahlige Parameter wird als Genauigkeit interpretiert

Argumentengröße h Ganzzahlige Typen werden als short interpretiert (bzw. uh für unsigned)
l Ganzzahlige Typen als long (ul für unsigned), Gleitkommtypen als double
L Gleitkommastellen werden als long double interpretiert

Typ			
	d, i	Ganzzahlige Dezimalzahl	`printf(„:+10d:",i); →:  +1:`
	o	Ganzzahlige Oktalzahl	`printf(„:10.2lf:",d);→:  1.20:`
	x, X	Ganzzahlige Hexadezimalzahl	`printf(„:010.2lf:",d)→:0000000001.20:`
	f	Vorzeichenbehaftete Gleitkommazahl	`printf(„:- 10.2lf:",d)→:1.20  :`
	e, E	VZG, wissenschaftliche Darstellung	`printf(„:- 10.2le:",d)→:  1.2e+00:`
	g, G	VZG, längenoptimiert zw. e/f-Typ	
	c	Zeichen	
	s	Stringbezeichner -genauer: Zeiger auf einen String	

```
char *pStr; printf(„%s", pStr);
```

Zeichenweise Ein/-Ausgabe

Headerdatei: stdio.h

getchar() Zeichenweise Eingabe von stdin(Tastatur)
List das nächste Zeichen von stdin
Fkt.wert: Gelesenes Zeichen bzw. EOF(=-1) bei Eingabe des Fileendzeichen (Ctrl Z)

putchar() Zeichenweise Ausgabe nach stdout(Bildschirm)
Fkt.wert: Ausgegebenes Zeichen, EOF (==-1) im Fehlerfall

4.2 Der C-Präprozessor

Präprozessoranweisungen beginnen mir #

Einbinden von Textdateien: `#include“dateibezeichnung“` “ “ aktuelle Directory

`# include<stdio.h>` <> wird in voreingestellten Directory gesucht

Definitionen eines Namens für die Zeichenfolge: #define NAME zeichenfolge

→ NAME wird beim Lesen durch „zeichenfolge“ ersetz

Parametrisierte Makros: #define(parameterliste) zeichenfolge

Die Parameterliste enthält durch „ ,“ getrennte Parameter, die in der zeichenfolge sein müssen

#define sqr(x) ((x)*(x)) → sqr(a+b) ist beim Aufruf ((a+b)*(a+b))

Steuerung des bedingten Preprocessing und der bedingten Kompilierung:

#if	konst_ausdruck			`#ifdef PI`
#ifdef	name	(#if defined	name)	`#define PI (2*acos(0))`
#ifndef	name	(#if ! defined	name)	`#endif`
#elif	konst_ausdruck			`#if (PROZ > P2)`
#else				`printf (“schnellstes System”);`
#endif				`#else … #endif`

Weitere Präprozessoranweisungen:

#line	konstante [“dateiname”]	Zeilennummer der Quelle festlegen
#pragma	zeichenfolge [parameter]	Frei für Implementierung
#error	zeichenfolge	Eigene Fehlermeldung formulieren

Kapitel 5: Nähere Betrachtung der Datentypen in C

5.1 Felder (Arrays)

Arraysyntax

$$\textit{Komponententyp Variablenname}\ \underbrace{[\textit{kost. Ausdruck}]}_{\substack{\textit{untere Grenze } 0 \\ \textit{obere Grenze } n-1}} = \begin{array}{lll} iFeld & \boxed{1.\textit{Komponente}} & [0] \\ & \vdots & \vdots \\ & \boxed{n-1\ \textit{Komponente}} & [n] \end{array}$$

Initialisierung von Arrays

Arrayvariablendefinition = {…}; `int iFeld[]={2, 5, 2*76, -3};`

Zeichenketten in C

Zeichenketten – Strings – in C sind Arrays mit dem Elementtyp char →“ \0 “ markiert das Stringende

→ Die char-Array-Variable, die einen String aufnimmt, muss mind um eine Komponente länger sein

`Char caWort[]={‘d‘, ‘e‘, ‘r‘, ‘\0‘} kann auch char caWort[]= “der“ geschrieben werden`

Eingabe mit scanf() oder gets() Ausgabe mit printf() oder puts()

Headerdatei stdio.h

gets() Einlesen eines Strings *char *gets(char*str);*

Liest Zeichen von stdin bis das nächste Zeilenende-Zeichen '\n' oder Dateiende EOF auftritt. Der String wird mit '\0' abgelegt (nicht mit \n !!!)

Parameter: Zeiger auf char-Array – char *str

Funktionswert: übergebener Zeiger oder Null-Pointer falls Dateiende ohne vorheriges Lesen erreicht wird (bzw. ein Lesefehler vorliegt

`scanf("%s", caWort);` ist nicht `gets(caWort);`

liest nur bis zum white space character und bricht also bei blanks und tabs ab — liest bis zum naechsten '\n'

puts() Ausgeben eines Strings. *int puts(const char * str);*

Der durch das char-Array - char * str - referenzierte String wird an stdout ausgegeben. Das abschließende '\0'-Zeichen wird dabei durch ein '\n'-Zeichen ersetzt.

Parameter: str Zeiger auf char-Array.

Funktionswert: nicht negativer Wert bei fehlerfreier Ausgabe oder EOF (-1) im Fehlerfall.

`puts(caWort);` ≈ `while (caWort[i] != '\0') { printf("%c = %02x\n",caWort[i]"); i++; }`

ersetzt '\0' durch '\n'

sscanf( ) Formatierte Eingabe aus einem String.

*int sscanf (char * str, "Formatstring", &Argument1,..., &ArgumentN);*

Aus dem String str werden die in „Formatstring" festgelegten Werte entnommen und den Argumenten zugewiesen (exakt wie bei „scanf( )").

Parameter: str: Pointer auf zu bearbeitenden String,
Formatstring: Enthält die Formatierung für jedes Argument,
Argument: Variable dessen Typ und Format in „Formatstring"festgelegt ist.

Funktionswert: Anzahl der erfolgreich konvertierten Eingabefelder oder EOF im Fehlerfall.

sprintf( ) Formatierte Ausgabe in einen String

*int sprintf (char * str, "Formatstring", Argument1, Argument2,..., ArgumentN);*

Die in „Formatstring" festgelegten Formate werden den Argumenten zugewiesen und in dem String str hinterlegt(exakt wie bei „printf( )"), abgeschlossen mit dem '\0'-Zeichen.

Parameter: str: Pointer auf zu bearbeitenden String,
Formatstring: Enthält die Formatierung für jedes Argument (vgl. „printf()"),
Argument: Variable dessen Typ und Format in „Formatstring"festgelegt ist.

Funktionswert: Anzahl der Zeichen, die in str abgelegt wurden (ohne '\0') oder EOF im Fehlerfall.

Headerdatei string.h

strcpy( ) String kopieren *char * strcpy(char * s1, const char * s2);*

Kopiert String s2 (Quellstring) in den String s1 (Zielstring).

Parameter: Pointer auf den Zielstring s1, Pointer auf den Quellstring s2.

Funktionswert: Pointer auf den Zielstring s1.

strcat( ) Anhängen eines Strings *char * strcat(char * s1, const char * s2);*

Der String s2 wird an den String s1 angehängt.

Parameter: Pointer auf den Zielstring s1, Pointer auf den Quellstring s2.

Funktionswert: Pointer auf den resultierenden String, d.h. s1.

strcmp() Vergleich zweier Strings *int strcmp(const char * s1, const char * s2);*

Die Strings s1 und s2 werden lexigrafisch verglichen.
Parameter: Pointer auf den ersten String s1, Pointer auf den zweiten String s2.
Funktionswert: < 0 , wenn s1 „kleiner“ als s2,
0 , wenn s1 „gleich“ s2,
> 0 , wenn s1 „größer“ s2.

strlen() Länge eines Strings ermitteln *size_t strlen(const char * s);*
Ermittelt die Länge des Strings s in Anzahl von Bytes, ohne abschließendes ‘\0’-Zeichen.
Parameter: String s
Funktionswert: Anzahl der Bytes des Strings s
Anmerkung: „size_t“ ist der „natürliche Ganzzahltyp“ und in <string.h> definiert.

strchr( ) String nach Zeichen durchsuchen *char * strchr(const char *str1, int c);*
strstr( ) String nach Teilstring durchsuchen *char * strchr(const char *str1, const char * str2);*
Durchsucht den String str auf das erste Auftreten des Zeichens c bzw. des Strings str2.
Parameter: str1 zu durchsuchender String, c/str2 zu suchendes/r Zeichen/Teilstring
Funktionswert: Pointer auf die Stelle im String str1, an der c bzw. str2 erstmals auftritt,
NULL-Pointer falls c bzw. str2 nicht in str1 enthalten ist.

Mehrdemensionale Arrays

Speicherung erfolgt Zeilenweise

☐	[0]	[1]		☐	[n]	[1]
☐	[0]	[2]	⋯	☐	[n]	[2]
☐	[0]	[3]		☐	[n]	[3]
1.Zeile bzw.Zeile 0				n−te Zeile		

Sind Mehrdemensionale Arrays Funktionsparameter, so darf in der Parameterdeklaration die Größenangabe für die 1. Dimension („Zeile“) weggelassen werden.

Initialisierung: `int math[3][4] = { {1, 4, 3, -4}, {2, 0, -3, 1},{0, 0, 0, 1};` oder
`int math[][4] = { {1, 4, 3, -4, 2, 0, -3, 1, 0, 0, 0, 1};` → 3*4 Array

Aber, wenn fehlende Angaben mit 0 initialisiert werden sollen

`int math[3][4] = { {1, 4, 3, -4}, {2, 0, -3, 1}};` oder
`int math[3][4] = { {1, 4, 3, -4, 2, 0, -3, 1 };` → Ein 3*4 Array
bzw. `int math[][4] = { {1, 4, 3}, {2, 0, -3}, {0, 0, 0}};`

5.2 Typumwandlung

Implizite Typumwandlung

aVar		bVar	Durchgeführte Aktion
int	←	float	Gebrochenen Anteil weggelassen
int	←	double	
int	←	long	Höherwertiges Bit weggelassen
char	←	int	
char	←	short	
float	←	double	Runden oder Abschneiden
float	←	long, int, short, char	Wenn keine exakte Darstellung möglich
double	←	long, int, short, char	Runden oder Abschneiden

Typ-Hierarchie	
long double	↓
double	↓
float	↓
unsigned long int	↓
long int	↓
unsigned int	↓
int	↓
niedrigster Typ	

Bei 2-stelligen arithmethischen Operationen wird bei unterschiedlichen Datentyp der niedrigere Datentyp in den höherwertigen umgewandelt (siehe Hierarchie)

Explizite Typumwandlung erfolgt mittels Cast-Operator, der unmittelbar vor dem Ausdruck steht
Unärer Operator: (typangabe) → Wert wird umgewandelt, Datentyp der Variablen bleibt unversehrt

```
int j, i; float f;  f =   (float)   i/j;
                          Cast-Operator
```
= gecastet → echtes Ergebnis z.B.1,5

5.3 Verfügbarkeit und Lebensdauer - Speicherklassen

Verfügbarkeit und Lebensdauer

(1)gesamtes Programm (2)Modul (File)	→ **globale Objekte:**	Definition außerhalb jeder Funktion Funktionen sind immer global!!!
(3)Funktion (4)Verbundanweisung	→ **lokale Objekte:**	Definition innerhalb eines Blockes Lokale Objekte können nur Variablen sein

Speicherklassen **extern – static – auto – register**

auto Speicherort: Stack
Alle Variablen (bzw. formalen Funktionsparameter), die innerhalb eines Blockes definiert sind und nicht explizit „static“ oder „register“ vereinbart sind

register wie Speicherklasse (auto) ABER die Variable in einem lokalen CPU-Speicher (Register) angelegt werden soll (Zugriffszeit)

extern Speicherort: Datensegment
Alle Funktionen und alle außerhalb jeder Funktion definierten Variablen, die nicht eplizit „static“ angegeben sind
Default-Initialisierung mit 0 und explizite Initialisierung durch konstanten Ausdruck

static Speicherort: Datensegment
static local: Anwendung auf lokales Objekt
behalten Werte zwischen Ausführungen eines Blockes. Interessant, wenn zwischen zwei Aufrufen der gleichen Funktion die Variable den zuletzt ermittelten Wert beibehalten soll. Initialisierung nur beim ersten Eintritt in diesen Block!
Static global: (Variable und Funktionen) Ermöglicht Globalität auf betrachtetes Modul zu beschränken. Sie können von anderen Modulen desselben Programmes nicht „gesehen“ werden. Interessant: Sollen also einzelne Funktionen und globale Variable nur im eigenen Quellmodul sichtbar sein!
Default-Initialisierung mit 0 und explizite Initialisierung durch konstanten Ausdruck!

Ebene	Vereinbarung	Speicherklasse	Lebensdauer	Verfügbarkeitsbereich
Modul	Variablendefinition	static	gesamte Programmlaufzeit	Rest des Moduls
	Variablendeklaration	extern		
	Funktions-Definition oder Deklaration	static		Beschränkt auf das aktuelle Modul
	Funktionsdeklaration	extern		Rest des Moduls
Block	Variablendeklaration	extern		Block
	Variablendefinition	static		
	Variablendefinition	auto / register	Blockausführung	

5.4 Zeiger

- Zeiger (Pointer) ist eine (Zeiger-)Variable oder eine (Zeiger-)Konstante und benötigt wie eine Variable Speicherplatz
- Der Wert der Zeigervariablen (bzw. -konstante) ist eine Speicheradresse
  → Ein Zeiger zeigt auf eine andere Variable oder auf eine Funktion (indirekt angesprochen)
- Objekt-Typen (an die Zeiger gebunden sind)
  Ein Zeiger auf int-Werte, kann nur Adressen von int-Variablen annehmen
  Ausnahme: *Generic pointer:* *void** kann auf beliebige Objekt-Typen zeigen
- Im Zusammenhang mit Zeigern existieren zwei - zueinander inverse unäre Operatoren:
  & Adressoperator liefert die Adresse eines Objektes `&x ⇒ Adresse der Variablen x`

  Adressen können Zeigern zugewiesen werden:
  ```
  px sei Zeigervariable (auf int)
  x sei int-Variable
  → px = &x;
  ```

  * Objektoperator liefert das Objekt, auf das ein Zeiger zeigt → Dereferenzierung

  `*px ⇒ Objekt, auf das der Zeiger px zeigt, bzw. Variable deren Adresse in px steht.`

  Zeigerobjekte können wie unter Namen referenzierte Variable verwendet werden:
  ```
  px  sei Zeigervariable (auf int)
  y  sei int-Variable
  → y = *px;
  ```
- int *px; *px wird als Zeigervariable vereinbart, die auf int-Werte zeigt→px zeigt auf int*

Zeigerarithmetik in C

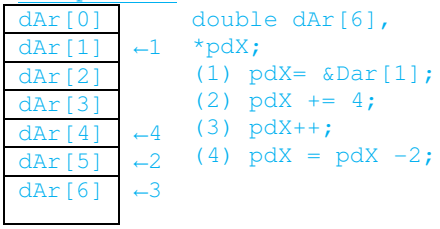
Auf Zeiger in C sind die folgenden Operationen definiert:

- Zuweisung des Wertes einer anderen Zeigervariablen (desselben Typs).
- Addition und Subtraktion eines ganzzahligen Wertes (einschließlich Inkrement und Dekrement).
- Die Veränderung erfolgt entsprechend der Größe des Objekttyps.
- Subtraktion von zwei Zeigern, die auf Objekte des gleichen Typs zeigen.
  Die Operation liefert die Anzahl der Objekte, um die die Werte der beiden Zeiger voneinander entfernt sind.
- Vergleichsoperationen (==, !=, >, <, >=, <=) zwischen Zeigern gleichen Typs.
- Zeigern, die keinen definierten Wert enthalten, sollte der Wert NULL (==0, NULL-Pointer) zugewiesen werden.

Beispiel 1:

```
int x=1;
*px,*py;          px zeigt auf  x
px =&x;           Die Adresse x bekommt den Wert
*px=5;            5
(*px)++;          „x“ wird um 1 erhöht
py=px;            py zeigt auf dasselbe objekt
*py +=3;          wie px
                  „x“ wird um 3 erhöht
```

Beispiel 2:




```
double dAr[6],
*pdX;
(1) pdX= &Dar[1];
(2) pdX += 4;
(3) pdX++;
(4) pdX = pdX -2;
```

Zeiger und Arrays

- pdX = &a[0]; pdX zeigt auf das erste Element von a
  → *pa ist gleich mit a[0] und *(pa+1) ist gleich mit a[1]
- Array-Namen werden durch den Compiler wie Zeiger auf das erste Arrayelement behandelt:
  → a ist äquvaläntzu &a[0] → statt pa=&[0] lässt sich auch pa =a formulieren
  → statt a[i] lässt sich auch *(a+i) formulieren
  → statt *(pa+i) lässt sich auch pa[i] formulieren
- **!!!UNTERSCHIED!!!**
  Eine *Zeiger*(-Variable) ist eine *Variable*, deren Wert jederzeit geändert werden darf.
  ```
  int     x, *px;     px = &x;
  ```
  ist zulässig
  Ein Array-Name ist eine Zeiger-Konstante, ihr Wert ist nicht veränderbar!
  ```
  int z, y[5];   y =&z;
  ```
  UNZULÄSSIG!!

Mehrdemensionale Arrays

char caF [2] [4]; caF → *caF →

	[0] [0]
	[0] [1]
	[0] [2]
	[0] [3]
	[1] [0]
	[1] [1]
	[1] [2]
	[1] [3]

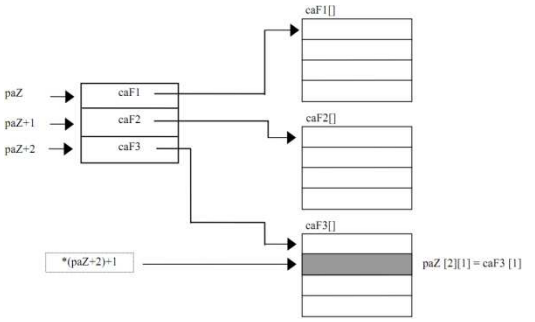
(caF+1) → *(caF+1) →
caF[1] + 1 (*caF+1)+1 →

→ *(*(caF + i) + j) = *(caF[i] + j) = caF[i] [j] = (*(caF + i)) [j]
char (*caMat)[4] = cahr ca Mat[] [4]

Zeiger Arrays

Zeiger - Arrays sind Arrays deren Komponenten Zeiger sind. Die Komponenten können auch Adressen von Arrays sein und es ergibt sich eine Ähnlichkeit zu mehrdimensionalen Arrays.

```
char * paZ[3];          Definition und damit Speicherplatzreservierung für ein
                        Array mit 3 Elementen, die jeweils Zeiger auf
                        char-Typen sind.
                        paZ ist ein Zeiger auf ein Array mit 3 char-
                        Zeigern
```




```
                        char    caF1 [4],
                                caF2 [4],
                                caF3 [4];
                        Definition und damit Speicherplatzreservierung
                        für 3 char-Arrays mit jeweils der Länge 4.
```

Programm-Parameter in C

- Programm beim Start, String als Parameter zu übergeben
- ***argc (= argument count)*** *und* ***argv (=argument vector)***
- bzw. *int main (int argc, char *argv[])* bzw. *int main (int argc, char **argv[])*
  Im ersten Parameter (argc) wird die Anzahl der Programmparameter übergeben, dabei zählt der Programmname immer als erster Parameter (==1, wenn es keine Programmparameter gibt)
  Der zweite argv ist ein Pointer auf ein chr-Pointer-Array, dessen Elemente auf die einzelnen Programm-Parameter-Strings verweisen.
  Das erste Element zeigt immer auf den Programmnamen und abgeschlossen ist das Array mit dem NULL-Pointer

```
argv[0]             ist ein Pointer auf den Programmnamen
argv[1]      ist ein Pointer auf den ersten echten Programm-Parameter
argv[argc-1] ist ein Pointer auf den letzten Programm-Parameter
argv[argc]   ist der NuLL-Pointer
```

Zeiger als Funktionsparameter und Funktionswert

Die Übergabe aktueller Werte für Funktionsparameter wird in C stets als Kopie durchgeführt. Für Zeiger als Parameter bedeutet dies, dass sie durch den Parameter übergebene Adresse als lokale Kopie in der Funktion zur Verfügung steht.
Parameter existieren - wie lokale "auto"-Variable - nur, solange die Funktion durchlaufen wird.

Konsequenzen :

- Als aktuelle Parameter übergebene Variable werden durch die in der Funktion ausgeführten Aktionen nicht verändert, da nur ihr jeweiliger Wert als Kopie übergeben wird (→ Wertübergabe).
- Soll eine außerhalb der Funktion existierende Variable durch die Funktion modifiziert werden, so muss ein Zeiger auf diese Variable als Parameter übergeben werden.
  → Die Übergabe eines Zeigers (== Adresse !) als Parameter wirkt wie eine Referenzübergabe für die Variable.
- Variable eines zusammengesetzten Datentyps, sollten im allgemeinen stets mittels eines Zeigers übergeben werden (schnellere Parameterübergabe, Einsparung von Speicherplatz !)
  Gegebenenfalls ist von Veränderungen einzelner Komponenten durch Anweisungen innerhalb der Funktion abzusehen.
- Arrays lassen sich nur per Zeiger übergeben (Array-Name == Zeiger-Konstante

Funktionswert (Rückgabewert) :

```
int *tuwas(int *, int *); Funktionswert ist ein Zeiger auf ein int-Objekt
```

Es ist dabei besonders darauf zu achten, dass der Rückgabewert auf noch existierende Variable verweist. Innerhalb der Funktion verwendete lokale Variable dürfen deshalb nicht referiert werden !

Zeiger auf Funktionen

- Eine Zeigervariable kann als Wert auch die Anfangsadresse einer Funktion besitzen → Zeiger auf Funktion
- Bei der Vereinbarung von Funktionszeigervariablen ist anzugeben:

  Funktionswerts der Objekt−Funktion Anzahl der Parameter der Objekt−Funktion

```
double        (* pfFunc)                  (int)
pfFunc ist ein Zeiger auf eine Funktion, die einen int-Parameter erwartet und
einen double-Wert zurückliefert
```

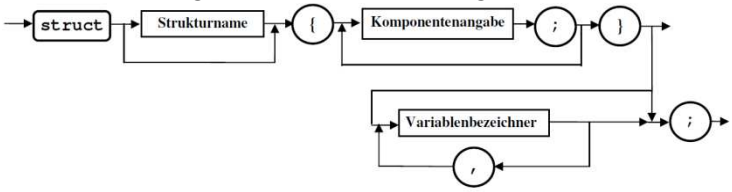
- Wertzuweisung an eine Funktionszeigervariable
  Wie bei den Arrays, so ist auch bei Funktionen der Name allein als Zeiger auf die Funktion festgelegt (keinen &-Operator anwenden, Funktionsname ist Adresskonstante !).

```
double dF1 (int);          Deklaration der Funktion dF1()
double (*pfFunc)(int);     Definition einer Funktionszeigervariablen
pfFunc = dF1;              Zuweisung der Adresse der Funktion dF1 als
                           Wert an die Variable pfFunc
```

5.5 Structuren

- structures in C sind selbstdefinierte zusammengesetzte Datentypen
  → mehrere Variablen sind somit zu einer einzigen Variable zusammengefasst

```
struct person
{ char name[20],
vname[20];
struct datum geburtstag;
char familienstand;
    }
```



- Variablenvereinbarung mit
  zuvor definierten Strukturnamen

```
struct person sStudent, sDiplomand, *psPersZeig;
```

Operationen mit Strukturen

- Adressoperator & `psPersZeig = &sStudent;`
- Zugriff zu Strukturkomponenten oder mit Objektelement-Operator "->"

```
scanf("%s", sStudent.name);
sStudent.vorname[0] = 'F';
sStudent.geburtstag.jahr = 2001;
```

```
scanf("%s", psPersZeig->name);
psPersZeig->vorname[0] = 'F';
psPersZeig->geburtstag.jahr = 2001;
```

- Wertzuweisung an Strukturvariable
  Geschlossene Zuweisung zwischen Strukturvariablen möglich `sDiplomand = sStudent;`
- Initialisierung von Strukturvariablen

```
struct datum geburtstag = {23, "Mai", 1983 };
```

- Strukturen als Funktionsparameter und Rückgabewerte von Funktionen
  Strukturen sind als Funktionsparameter und Rückgabewerte von Funktionen zulässig.
  ACHTUNG : Bei größeren Strukturen ist es meist effizienter Zeiger auf Strukturen als Parameter zu verwenden, um umfangreiches Kopieren beim Funktionsaufruf zu vermeiden!

5.6 Festlegung von Dateinamen mittels “typedef“

- Es wird kein neuer Typ, sondern nur ein neuer Name für einen existierenden Typ definiert.
- Anwendung: Parametrisierung eines Programmes gegen Portabilitätsprobleme durch Definition von Typnamen für maschinenabhängige Datentypen. Diese werden meist in eigenen Header-Dateien aufgeführt.

```
typedef unsigned int WORD;
typedef unsigned char BYTE;
```

  Erhöhung der Übersichtlichkeit und Lesbarkeit eines Quelltextes durch die Wahl von problemangepaßten und treffenden Typnamen → selbstdokumentierend!

Statt	kann man auch formulieren:

`struct datum`
`{ int tag;`
`char monat[4];`
`int jahr;`
`};`
`struct datum geb_datum;`
	`typedef struct`
	`{ int tag;`
	`char monat[4];`
	`int jahr;`
	`} datum; /* Typname !! */`
	`datum geb_tag;`

```
typedef char wort[30];     wort name;
typedef float REAL;        REAL a;
```

5.7 Aufzählungstypen und Aufzählungskonstante

- Alternative zur Festlegung symbolischer Konst. mit Hilfe der Preprozessor-Direktive #define
- Aufzählungskonstante sind immer vom Typ int und werden immer gleichzeitig mit der Definition eines Aufzählungstyps festgelegt.
- Wird ein Konstantenname mittels "=" mit einem konstanten Ausdruck gleichgesetzt, so wird ihm der Wert dieses Ausdrucks zugeordnet. Die nachfolgenden Konstantennamen erhalten dann in der Reihenfolge ihrer Auflistung den jeweils nächsten int-Wert zugeordnet, wenn sie nicht selbst explizit mit einem "=" in ihrem Wert festgelegt werden.
- Wird kein Typname angegeben, so handelt es sich um die Definition eines namenlosen Typs.

```
enum { MON, DIE, MIT, DON, FRE, SAM, SON };

enum MONAT { JAN = 1, FEB, MAR, APR, MAI, JUN,…};    Typdefinit.: enum MONAT
enum MONAT MyMonat         Definition der Variablen MyMonat
typedef enum MONAT MONTH   Definition des neuen Typnamnes MONTH
MONTH Wonnemonat = MAI;    Definition und Initialisierung einer Variablen
```

5.8 Der sizeof-Operator in C

- unärer Operator
- Die Auswertung erfolgt zur Compilezeit! Der Operator dient zur Ermittlung der Größe des Speicherplatzes in Bytes, den ein Objekt des Typs seines Operanden belegt.

```
sizeof 3            sizeof (3)              sizeof (a+b)
sizeof (int)        sizeof (double)         sizeof (char *)
```

- Ergebnis der Operation ist vom Typ size_t
- Anwendung :
  - Ermittlung des Platzbedarfs zusammengesetzter Datentypen, wie "structures" und "arrays"
  - zur portablen Angabe des Platzbedarfs der Standard-Datentypen

```
int i=6, aiM[17];
printf("int %2u Bytes\n", sizeof i);                          --> 4
printf("long double %2u Bytes\n", sizeof(long double));       --> 12
printf("Arraygroesse%2u Bytes\n", sizeof aiM                  --> 68
printf("Arraygroesse%2u Elemente\n", sizeof aiM/sizeof aiM[0]);  --> 17
```

Kapitel 6: Dateibearbeitung in C

6.1 Dateibearbeitungskonzept von ANSI-C

- Jegliche Ein- und Ausgabe in C geschieht mittels Bibliotheksfunktionen. Ihre Deklarationen sind in der Headerdatei <stdio.h> enthalten.
- Jede zu bearbeitende Datei bzw. jedes Gerät wird logisch als Stream – als kontinuierliche geordnete Folge von Bytes - betrachtet.
- Referiert wird ein Stream über einen Filepointer dem Typnamen FILE.
- Der genaue Aufbau des Datentyps FILE ist betriebssystem- und implementierungsabhängig. Er ist in <stdio.h> definiert.
- ANSI-C unterscheidet 2 Dateiformate (können Text- oder im Binär-Modus geöffnet werden)
  Textdateien: bestehen aus einer Folge von Zeilen. Jede Zeile ist logisch mit einem Newline-Zeichen abgeschlossen.
  Binärdateien: bestehen aus einer Folge von Bytes und besitzen keine weitere Struktur.
- Dateien können im Text- oder im Binär-Modus geöffnet

6.2 Grundlegende Funktionen zur Dateibearbeitung

Headerdatei stdio.h

fopen() öffnen einer Datei `FILE *fopen(const char *path, const char *mode);`

öffnet die durch den *path* bezeichnete Datei und liefert einen Pointer das File-Arrays zurück

mode:			
	"r"	Lesen	(Datei muß existieren)
	"w"	Schreiben	(evtl. exist. Datei wird gelöscht)
	"a"	Anhaengen	(evtl. exist. Datei wird nicht gelöscht)
	"r+"	Lesen und Schreiben	(Datei muß existieren) *)
	"w+"	Schreiben und Lesen	(evtl. exist. Datei wird gelöscht) *)
	"a+"	Lesen	(an beliebiger Position) und Schreiben am Dateiende

*) Zwischen Lesen und Schreiben muß jeweils ein explizites Herausschreiben der Dateipuffer erfolgen (Aufruf von fflush).
Durch Anhängen der Zeichen 'b' oder 't' kann zusätzlich angegeben werden, ob es sich um eine Binär- oder Textdatei handelt.

Funktionswert : Filepointer bzw. NULL-Pointer im Fehlerfall.

fclose() schließen einer Datei `int fclose(FILE *fp);`

schreibt noch nicht herausgeschriebene Filebuffer in die durch "fp" referierte Datei, gibt die Buffer und die FILE-Array-Komponente, auf die fp zeigt, frei und schließt die Datei.
Übergabeparameter: fp Filepointer
Funktionswert : im Erfolgsfall der Wert "0" bzw. im Fehlerfall der Wert EOF.

fseek() Verändern der Bearbeitungsposition `int fseek(FILE *fp, long offset, int origin);`

verändert die aktuelle Bearbeitungsposition der durch "fp" referierten Datei um "offset" Bytes gegenüber der Bezugsposition "origin" (→wahlfreier Dateizugriff !)

Übergabeparameter: fp Filepointer
offset Anzahl Bytes relativ zur Bezugsposition
origin Bezugsposition

Funktionswert: "0", wenn die Bearbeitungsposition wie angegeben verändert werden konnte
bzw. ein Wert ungleich "0" im Fehlerfall
Die Bezugsposition wird durch einen der folgenden 3 Werte fuer "origin" gekennzeichnet :
0 (SEEK_SET) Dateianfang
1 (SEEK_CUR) augenblickliche Bearbeitungsposition
2 (SEEK_END) Dateiende
Achtung : Bei Textdateien muß offset == 0 oder ein von der Funktion "ftell()" zurück gegebener Wert sein, wobei dann origin == SEEK_SET (Dateianfang !) sein muß.

frewind() Zurücksetzen auf Dateianfang `void rewind(FILE *fp);`
setzt die durch "fp" referierte Datei auf den Anfang zurueck.
Übergabeparameter: fp Filepointer
Funktionswert: keiner es gilt : rewind(fp) == (void)fseek(fp, 0L, SEEK_SET)

ftell() Ermitteln der aktuellen Bearbeitungsposition `long ftell(FILE *fp);`
ermittelt die aktuelle Bearbeitungsposition der durch "fp" referierten Datei.
Übergabeparameter: fp Filepointer
Funktionswert: aktuelle Bearbeitungsposition
bei Binärdateien : Anzahl Bytes relativ zum Dateianfang
bei Textdateien : für "fseek()" verwertbare Information (s. oben)
bzw. "-1L" im Fehlerfall

feof() Überprüfung auf Dateiende `int feof(FILE *fp);`
überprüft, ob für die durch "fp" referierte Datei das Dateiende-Flag gesetzt ist.
Übergabeparameter: fp Filepointer
Funktionswert: ein Wert !=0, wenn das Dateiende-Flag gesetzt ist. "0", wenn das Dateiende-Flag nicht gesetzt ist.

fflush() Herausschreiben eines Dateipuffers `int fflush(FILE *fp);`
Wenn "fp" eine Datei referiert, die zum Schreiben oder Update - bei zuletzt durchgeführter Schreiboperation - geöffnet wurde, veranlasst "fflush()", dass der zugehörige Dateipuffer geleert, d.h. an das Betriebssystem zum Herausschreiben in die Datei übergeben wird; die Datei bleibt geöffnet.
Übergabeparameter: fp Filepointer
Funktionswert : "0", wenn der Puffer erfolgreich geleert werden konnte, EOF, wenn beim Schreiben ein Fehler aufgetreten ist.

setbuf() Bereitstellen eines benutzerdefinierten Dateipuffers
`void setbuf(FILE *fp, char *buffer);`
stellt für die durch "fp" referierte Datei einen durch "buffer" referierten benutzerdefinierten Dateipuffer zur Verfügung. "buffer" muss auf den Anfang eines char-Arrays der Laenge BUFSIZ (definiert in <stdio.h>) zeigen.
Ist "buffer" ==NULL, so erfolgt der Zugriff zur Datei ungepuffert.
Übergabeparameter: fp Filepointer buffer Pointer auf benutzerdefinierten Dateipuffer

6.3 Funktionen zum Datentransfer

Headerdatei stdio.h

fgetc() Zeichenweises Lesen `fgetc(FILE *fp);`
getc() Zeichenweises Lesen `getc(FILE *fp);`
liest das nächste Zeichen aus der durch "fp" referierten Datei.
Übergabeparameter: fp Filepointer
Funktionswert : das gelesene Zeichen (als int-Wert !) bzw. EOF bei Dateiende/ Fehlerfall
es gilt : getc(stdin) == getchar() !

ungetc() Rückgabe eines Zeichens in den Eingabepuffer `int ungetc(int c, FILE *fp);`
gibt das Zeichen "c" (umgewandelt in unsigned char) in den Eingabepuffer der durch "fp" referierten Datei zurück.
Übergabeparameter: c zurueckzugebendes Zeichen, fp Filepointer
Funktionswert : das zurückgegebene Zeichen bzw EOF im Fehlerfall

fputc() Zeichenweises Schreiben `int fputc(int c, FILE *fp);`
putc() Zeichenweises Schreiben `int putc(int c, FILE *fp);`
schreibt das Zeichen "c" (umgewandelt in unsigned char) in die durch "fp" referierte Datei.
Übergabeparameter: c zu schreibendes Zeichen, fp Filepointer
Funktionswert : das geschriebene Zeichen (als int-Wert !) bzw. EOF im Fehlerfall
es gilt: putc(c,stdout) == putchar(c) !

fscanf() Formatiertes Lesen `int fscanf(FILE *fp, const char *ctrl , ...);`
liest die nächsten Zeichen aus der durch "fp" referierten (Text-)Datei und weist die demgemäß konvertierten Werte den durch ihre Pointer (weitere Parameter !) referierten Variablen zu.
Übergabeparametert: fp Filepointer, ctrl String zur Steuerung des Eingabeformats
... weitere Parameter : Pointer auf Variable, die die Eingabewerte aufnehmen sollen
Aufbau und Bedeutung von "ctrl" exakt wie bei der Funktion scanf().
Funktionswert : die Anzahl der erfolgreich zugewiesenen Werte bzw EOF wenn – beim Lesen des ersten Werts – versucht wurde über das Dateiende hinaus zu lesen
es gilt : fscanf(stdin,ctrl, ...) == scanf(ctrl, ...)

fprintf() Formatiertes Schreiben `int fprintf(FILE *fp, const char *ctrl, ...);`
gibt die als weitere Parameter übergebenen Werte entsprechend den Formatangaben in "ctrl" als Zeichenfolgen in die durch "fp" referierte (Text-) Datei aus.
Übergabeparametetr: fp Filepointer, ctrl String zur Festlegung des Ausgabeformats
... weitere Parameter: Ausgabewerte, Anzahl und Typ entsprechend den Konvertierungsanweisungen in "ctrl".
Aufbau und Bedeutung von "ctrl" exakt wie bei der Funktion printf().
Funktionswert : Anzahl der ausgegebenen Zeichen
es gilt : fprintf(stdout,ctrl, ...) == printf(ctrl, ...)

fgets() Zeilenweises Lesen `char *fgets(char *s, int n, FILE *fp);`

liest die nächste Zeile (einschliesslich '\n') aus der durch "fp" referierten Datei, max (n-1) Zeichen. Die gelesenen Zeichen werden in String abgelegt (**einschliesslich '\n'** sofern es gelesen wurde), wobei '\0' angefügt wird.

Übergabeparameter: s String zum Ablegen der gelesenen Zeile (genauer: Zeiger auf String), n max Anz der zu lesenden Zeichen + 1,fp Filepointer

Funktionswert : String "s" (genauer: Pointer "s" auf String) bzw NULL-Pointer bei Erreichen des Dateiendes ohne vorheriges Lesen von Zeichen oder im Fehlerfall

fputs() Stringweises Schreiben `int fputs(const char *s, FILE *fp);`

schreibt den durch "s" referierten String (ohne abschließenden '\0'-Character) in die durch "fp" referierte Datei.

Übergabeparameter: s auszugebender String, fp Filepointer

Funktionswert : ein nicht-negativer Wert bei Fehlerfreiheit bzw EOF (== -1) im Fehlerfall

Hinweis : Anders als bei puts() wird an den ausgegebenen String kein '\n' angefügt.

fread() Datenobjektorientiertes Lesen

```
size_t fread(void *bptr, size_t size, size_t count, FILE *fp);
```

liest maximal "count" Datenobjekte der Länge "size" Bytes aus der durch "fp" referierten Datei und legt sie in dem durch "bptr" bezeichneten Buffer ab.

Übergabeparameter: bptr Pointer auf Buffer zur Ablage der gelesenen Datenobjekte, size Länge eines Datenobjekts in Bytes, count Anzahl der zu lesenden Datenobjekte, fp Filepointer

Funktionswert: Anzahl der tatsächlich in voller Länge gelesenen Datenobjekte (diese kann im Fehlerfall oder bei Erreichen des Dateiendes weniger als "count" sein).

fwrite() Datenobjektorientiertes Schreiben

```
size_t fwrite(const void *bptr, size_t size, size_t count, FILE *fp);
```

schreibt "count" Datenobjekte der Länge "size" Bytes in die durch "fp" referierte Datei. Die auszugebenden Datenobjekte werden dem Buffer entnommen, auf den "bptr" zeigt.

Übergabeparameter: bptr, size , count, fp

Funktionswert: Anzahl der geschriebenen Datenobjekte (kann im Fehlerfall weniger als "count" sein).

6.3 Funktionen zur Dateiverwaltung

Headerdatei stdio.h

remove() Entfernen einer Datei aus dem Dateisystem `int remove(const char *path);`

entfernt die durch "path" bezeichnete Datei aus dem Dateisystem. Falls diese Datei zum Aufruf von "remove()" geöffnet ist, ist Verhalten der Funktion implementierungsabhängig.

Übergabeparameter: path Pfadname der zu entfernenden Datei

Funktionswert: "0", falls erfolgreich bzw. ein Wert !="0" im Fehlerfall

rename() Umbenennen einer Datei im Dateisystem

```
int rename(const char *old, const char *new);
```

benennt die durch "old" bezeichnete Datei um in "new" → "old" ist danach ungültig.

Falls "new" existiert, ist das Verhalten implementierungsabhängig.

Übergabeparameter: old alter Pfadname der Datei new neuer Pfadname der Datei

Funktionswert: "0", falls erfolgreich bzw. ein Wert !="0" im Fehlerfall, der alte Pfadname bleibt gültig

Kapitel 7: Ergänzungen zu Datenstrukturen in C

7.1 Dynamische Speicherallokation

- dynamisch allokierte - Objekte können nicht wie statische Objekte definiert werden. Sie werden auch nicht über einen Namen, sondern nur über ihre Adresse angesprochen
- Die dynamische Speicherbereitstellung erfolgt mittels spezieller Speicherallokationsfunktionen, die in der Standardbibliothek (<stdlib.h>) enthalten sind :

  malloc() Allokation eines Speicherblocks bestimmter Größe.
  calloc() Allokation eines Speicherblocks für eine bestimmte Anzahl von Objekten einer bestimmten Größe, Initialisierung aller Bytes des Speicherblocks mit 0.
  realloc() Veränderung der Größe eines allokierten Speicherblocks

  Diese Funktionen liefern die Anfangsadresse des allokierten Blocks als void-Pointer (void *).
  →Es ist kein Type-Cast bei der Zuweisung an Pointer-Variable erforderlich.
- Der für die dynamische Speicherverwaltung zur Verfügung stehende Speicherbereich wird als *Heap* bezeichnet.
- Die Lebensdauer dynamisch allokierten Speichers ist nicht an die Ausführungszeit eines Blocks gebunden. Nicht mehr benötigter dynamisch allokierter Speicher ist explizit freizugeben. Bibliotheksfunktion hierfür : free() - Gibt dynamisch allokierten Speicher frei.

Headerdatei stdlib.h

---

malloc() Allokation eines Speicherblocks `void *malloc(size_t size);`

allokiert einen Speicherblock von wenigstens der Länge "size" Bytes
Übergabeparameter : size Anzahl der Bytes des zu allokierenden Blockes
Funktionswert : Pointer auf den allokierten Speicherblock;
NULL-Pointer, falls der Speicherplatz nicht ausreicht.

calloc() Allokation und Initialisierung eines Speicherblocks

```
void *calloc(size_t nobj, size_t size);
```

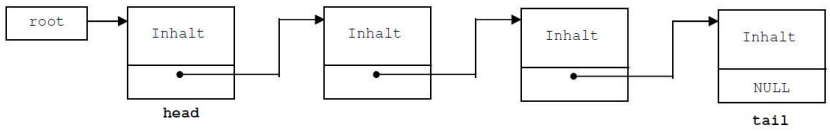
allokiert einen Speicherblock von wenigstens der Länge "nobj" * "size" Bytes und initialisiert alle Bytes mit 0.
Übergabeparameter: nobj Anzahl der Objekte, für die Speicherplatz allokiert werden soll;
size Länge eines Objektes in Bytes.
Funktionswert: wie malloc

free() Freigabe von Speicherplatz `void free(void *ptr);`

gibt den - zuvor allokierten - Speicherblock, auf den "ptr" zeigt, wieder frei.
Übergabeparameter: ptr Zeiger auf allokierten Speicherblock
Funktionswert : keiner

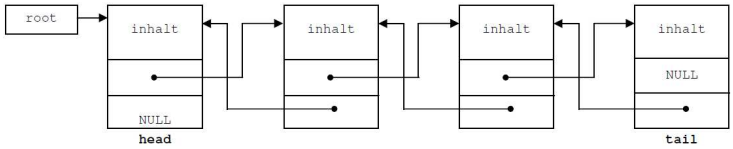
7.2 Dynamische Datenstrukturen

- Dynamische Datenstrukturen ändern ihre Struktur und den von ihnen belegten Speicherplatz während der Programmausführung. Sie sind aus einzelnen Elementen (Knoten) aufgebaut, zwischen denen üblicherweise eine bestimmte ("Nachbarschafts"-) Beziehung besteht.
- Die Beziehung der einzelnen Knoten muss untereinander über Zeiger, die jeweils auf den Speicherort des "Nachbarn" zeigen, hergestellt werden.
  → Verkettete Datenstrukturen
- Die wichtigsten dieser Datenstrukturen sind:
  Lineare Listen (einfach, doppelt, Spezialformen, Queue, Stack)
  Bäume (Binärbäume, Vielweg-Bäume)
  Allgemeine Graphen
- In C lassen sich die einzelnen Elemente (Knoten) durch "structures" darstellen. Zur Realisierung verketteter Datenstrukturen müssen diese Strukturen Zeiger auf Strukturen ihres eigenen Typs enthalten. Derartige Strukturen nennt man rekursiv.



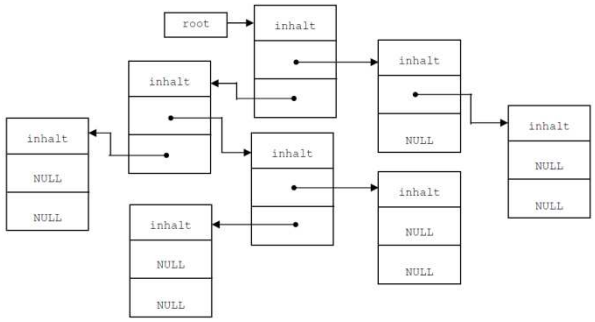
- einfach verkettete Liste

```
struct listelement
{ int inhalt;
struct listelement *next;
};
```



- doppelt verkettete Liste

```
struct listelement
{ int inhalt;
struct listelement *next;
struct listelement *back;
};
```



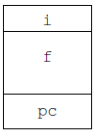
- Binärer Baum

```
struct baumelement
{ int inhalt;
struct baumelement *rechts;
struct baumelement *links;
};
```

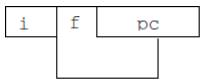
7.3 Unions (Variante Strukturen)

- Unions sind Strukturen, deren Komponenten alle an der gleichen Anfangsadresse im Arbeitsspeicher beginnen.
  → liegen "übereinander" im gleichen Speicherbereich, der am Unionanfang beginnt

```
struct
{ int i;
float f;
char *pc;
      } svar;
```




```
union
{ int i;
float f;
char *pc;
      } uvar;
```



- Der von der Union belegte Speicherplatz richtet sich nach der längsten Komponente.
- Die Vereinbarung von Unions (Typdefinition und Variablenvereinbarung) erfolgt analog zur Vereinbarung von Strukturen.
- Initialisierung von Unions ist nur für die 1. Komponente mit konstantem Ausdruck zulässig.
- Eine Union-Variable kann als eine Variable aufgefasst werden, die Werte unterschiedlichen Typs annehmen kann.

7.4 Vorwärtsdeklaration von Struktur-Typen

- In ANSI-C ist es möglich, einen struct- oder union-Typ vorwärts zu deklarieren. Diese Typ-Vorwärtsdeklaration besteht lediglich aus dem Wortsymbol struct bzw. union und dem Struktur- bzw. Union-Name. Die Auflistung der Komponenten fehlt.

```
struct st2
{ int ba;
int lu;
};

int main(void)
{ struct st2;              Typ-Vorwaertsdeklaration, setzt globale
                           Vereinbarung von struct st2 ausser Kraft
struct st1
{ struct st2* sp2;         Pointer auf lokal definierte sruct st2
int a;
};
struct st2                 lokale Definition von struct s2 */
{ struct st1 sp1;
int b;
};

struct st1 mys1 = { NULL, 7};
struct st2 mys2 = { mys1, 5 };
mys1.sp2 = &mys2;
printf("%d\n", (mys1.sp2)->b);   ohne Typ-Vorwaertsdeklaration falsch
return 0;
}
```

- Anwendung : Die Typ-Vorwärtsdeklaration ermöglicht die Definition von structure- bzw. union-Typen, die gegenseitig aufeinander Bezug nehmen wobei wenigstens einer der Typnamen bereits für einen anderen structure- bzw. union-Typ in einem übergeordneten Block vereinbart ist.

Kapitel 8: Die C-Satandardbibliothek

8.1 Die AINSI/ISO-C-Standardbibliothek

Makro ≠ Funktion	Makros bei kurzen Funktionen sinnvoll (Springt nicht zur Funktion sondern ersetzt Funktion)

Standard-Header-Dateien von ANSI/ISO-C

<stdio.h>	Definition von Konstanten und Typen, Deklaration von Funktionen zur Ein- und Ausgabe (I/O-Funktionen)
<stdlib.h>	Definition von Konstanten und Typen, sowie Deklaration diverser Utility-Funktionen (z.B. String-Konvertierungsfunktionen, Speicherverwaltungsfunktionen, Environment-Funktionen, Such- und Sortier-Funktionen, Integer-Arithmetik-Funktionen, Multibyte-Zeichen-Funktionen)
<string.h>	Deklaration von Funktionen zur Stringbearbeitung und Zeichenarraybearbeitung

Headerdatei stdio.h

```
int fclose (FILE *fp); Schließen einer Datei
int fflush (FILE *fp); Herausschreiben eines Dateipuffers
FILE *fopen (char *path, char *mode); Öffnen einer Datei
int remove (char *path); Löschen einer Datei
int rename (char *oldpath, char *newpath); Umbenennen einer Datei
int feof (FILE *fp); Überprüfung des EOF-Flags

int getchar (void);zeichenweise Eingabe von stdin
char *gets (char *s); zeilenweise Eingabe von stdin
int printf (char *ctrl, ...);formatierte Ausgabe nach stdout
int putchar (int c); zeichenweise Ausgabe nach stdout
int puts (char *s); zeilenweise Ausgabe nach stdout
int scanf (char *ctrl, ...); formatierte Eingabe von stdin

int fgetc (FILE *fp); zeichenweise Eingabe
char *fgets (char *s, int n, FILE *fp); zeilenweise Eingabe
int fprintf (FILE *fp,char *ctrl, ...);
formatierte Ausgabe
int fputc (int c, FILE *fp); zeichenweise Ausgabe
int fputs (char *s, FILE *fp); stringweise Ausgabe

int fscanf (FILE *fp, char *ctrl, ...); formatierte Eingabe
size_t fwrite (void *ptr, size_t size,size_t count, FILE *fp);
datenobjektorientierte Ausgabe
int getc (FILE *fp); zeichenweise Eingabe
int putc (int c, FILE *fp); zeichenweise Ausgabe
```

Headerdatei string.h

strcpy() Kopieren eines Strings `char *strcpy(char *str1, const char *str2);`
kopiert den String "str2" in den String "str1"
Übergabeparameter : str1, str2
Pointer auf den Zielstring, d.h. "str1" (== 1.Parameter)

strcat() Konkatenation zweier Strings `char *strcat(char *str1, const char *str2);`
hängt den String "str2" an den String "str1" (wird mit dem '\0' abgeschlossen) an.
Übergabeparameter : str1, str2
Funktionswert: Pointer auf den resultierenden String, d.h. "str1" (== 1.Parameter)

strcmp() Vergleich zweier Strings `int strcmp(const char *str1, const char *str2);`
vergleicht den String "str1" lexikographisch mit dem String "str2"
Übergabeparameter : str, str2
Funktionswert : <0 wenn "str1" kleiner "str2" ist
==0 wenn "str1" gleich "str2" ist
>0 wenn "str1" größer "str2" ist

strlen() Ermittlung der Länge eines Strings `size_t strlen(const char *str);`
ermittelt die Länge des Strings "str" in Anzahl Zeichen (ohne '\0')
Übergabeparameter : str String, dessen Länge ermittelt werden soll
Funktionswert : Länge des Strings "str" in Anzahl Zeichen

strchr () Durchsuchen eines Strings nach einem Zeichen
`char *strchr(const char *str, int c);`

durchsucht den String "str" nach dem ersten Auftreten des Zeichens "c".
Übergabeparameter : str zu durchsuchender String, C Suchzeichen
Funktionswert: Pointer auf die Stelle im String "str", an der das Zeichen "c" erstmals auftritt bzw NULL-Pointer, wenn das Zeichen nicht enhalten ist

strstr() Durchsuchen eines Strings nach einem anderen String

```
char *strstr(const char *str1, const char *str2);
```

durchsucht den String "str1" nach dem ersten Auftreten des Strings "str2".
Übergabeparameter : str1, str2
Funktionswert: Pointer auf die Stelle im "str1", an der der String "str2" beginnt bzw NULL-Pointer, wenn der String "str2" nicht enhalten ist

memcpy() Kopieren einer Zeichenfolge bestimmter Länge

```
void *memcpy(void *s1, const void *s2, size_t n);
```

kopiert "n" Zeichen aus dem '*aus*' s2 '*in*' s1
Übergabeparameter : s1 Pointer auf Ziel-Datenobjekt (Zielzeichenfolge)
s2 Pointer auf Quell-Datenobjekt, n Anzahl der zu kopierenden Zeichen
Funktionswert : Pointer auf den Ziel-Speicherbereich, d.h. "s1" (== 1.Parameter)
Anmerkungen: Quell- und Ziel-Speicherbereich dürfen sich nicht überlappen

memcmp() Vergleich zweier Zeichenfolgen bestimmter Länge

```
int memcmp(const void *s1, const void *s2, size_t n);
```

vergleicht die ersten "n" Zeichen '*aus*' s1 "n" '*aus*' "s2"
Für den Vergleich werden die Zeichen als unsigned char interpretiert.
Übergabeparameter : s1 Pointer auf …, s2 Pointer auf…, n Zeichen-Anzahl
Funktionswert :
<0 wenn "s1" < "s2"
==0 wenn die ersten "n" Zeichen in beiden Zeichnfolgen gleich sind
>0 wenn "s1" > "s2"

memchr() Suchen eines Zeichens in einer Zeichfolge bestimmter Länge

```
void *memchr(const void *s, int c, size_t n);
```

durchsucht die ersten "n" Zeichen des durch "s" referierten Datenobjekts nach dem ersten Auftreten des – in unsigned char umgewandelten – Zeichens "c".
Übergabeparameter : s Pointer auf das zu durchsuchende Datenobjekt, c Zeichen, nach dem gesucht wird, n Länge der zu durchsuchenden Zeichenfolge
Funktionswert : Pointer auf die Stelle, an der das Zeichen "c" erstmals auftritt bzw NULL-Pointer, wenn das Zeichen "c" nicht enhalten ist

memset() Füllen eines Speicherbereichs mit einem Zeichen

```
void *memset(void *s, int c, size_t n);
```

setzt die ersten "n" Zeichen ‚*von*' "s" auf den Wert Zeichens "c"(unsigned char)
Übergabeparameter : s Pointer auf den zu füllenden Speicherbereich, c Zeichen, mit dem derSpeicherbereich gefüllt wird, n Länge des zu füllenden Speicherbereichs (in Bytes)
Funktionswert : Pointer auf gefüllten Speicherbereich (== Parameter "s")