

Einführung in die Programmiersprache C

Inhaltsverzeichnis

1. Einführung	1 - 1
1. 1. Historischer Überblick	1 - 1
1. 2. Einführendes Beispiel	1 - 3
1. 3. Zeichenvorrat	1 - 5
1. 4. Wörter	1 - 6
1. 5. Kommentare	1 - 7
1. 6. C-Präprozessor	1 - 7
1. 7. main-Funktion	1 - 8
1. 8. Datenausgabe	1 - 9
1. 9. Dateneingabe	1 - 10
Übungen	1 - 13
2. Elementare Ausdrücke	2 - 1
2. 1. Konstanten	2 - 1
2. 1. 1. Integerkonstanten	2 - 1
2. 1. 2. Gleitkommakonstanten	2 - 3
2. 1. 3. Aufzählungskonstanten	2 - 4
2. 1. 4. Zeichenkonstanten	2 - 4
2. 1. 4. Zeichenketten	2 - 5
2. 2. Variablen	2 - 7
2. 3. Funktionsaufrufe	2 - 8
2. 4. Speicherklassen	2 - 9
Übungen	2 - 13
3. Ausdrücke und Operatoren	3 - 1
3. 1. Ausdrücke	3 - 1
3. 2. Operatoren	3 - 2
3. 2. 1. Arithmetische Operatoren	3 - 2
3. 2. 2. Vergleichsoperatoren	3 - 4
3. 2. 3. Logische Operatoren	3 - 4
3. 2. 4. Bitmanipulation	3 - 6
3. 2. 5. Bedingte Bewertung	3 - 8
3. 2. 6. sizeof-Operator	3 - 8
3. 2. 7. Zuweisungsoperatoren	3 - 9
3. 2. 8. Inkrement- und Dekrementoperatoren	3 - 10
3. 2. 9. Kommaoperator	3 - 10
3. 2. 10. Typkonvertierung	3 - 11
3. 2. 11. Prioritätstabelle	3 - 11
Übungen	3 - 12
4. Anweisungen	4 - 1
4. 1. Einfache Anweisungen	4 - 1
4. 1. 1. Ausdrucksanweisung	4 - 1
4. 1. 2. Anweisungsblock	4 - 2
4. 1. 3. Leere Anweisung	4 - 2
4. 2. Wiederholungsanweisungen	4 - 3
4. 2. 1. while-Schleife	4 - 3
4. 2. 2. for-Schleife	4 - 4
4. 2. 3. do-while-Schleife	4 - 5
4. 3. break-Anweisung	4 - 6
4. 4. continue-Anweisung	4 - 6

4. 5. Entscheidungsanweisungen	4 - 6
4. 5. 1. if-Anweisung	4 - 6
4. 5. 2. switch-Anweisung	4 - 8
4. 6. Sprunganweisung	4 - 9
4. 7. return-Anweisung	4 - 10
Übungen	4 - 11
 5. Zeiger und Felder	 5 - 1
5. 1. Zeiger	5 - 1
5. 1. 1. Adressen	5 - 1
5. 1. 1. Adressoperator und Dereferenzierung	5 - 2
5. 1. 3. Deklaration von Zeigern	5 - 3
5. 2. Felder	5 - 6
5. 2. 1. Einfach indizierte Felder	5 - 6
5. 2. 2. Initialisierung	5 - 7
5. 2. 3. Mehrfach indizierte Felder	5 - 8
5. 2. 4. Feldnamen als Zeiger	5 - 9
Übungen	5 - 12
 6. Funktionen	 6 - 1
6. 1. Definition einer Funktion	6 - 1
6. 2. Parameter	6 - 2
6. 2. 1. call-by-value	6 - 2
6. 2. 2. call-by-reference	6 - 2
6. 2. 3. Felder als Parameter	6 - 3
6. 2. 4. Parameter in main ()	6 - 4
6. 3. Rekursive Funktionen	6 - 5
6. 4. Zeiger auf Funktionen	6 - 6
6. 5. Bibliotheksfunktionen	6 - 7
6. 5. 1. Ein-/Ausgabefunktionen	6 - 7
6. 5. 2. Weitere Bibliotheksfunktionen	6 - 9
6. 6. Makrofunktionen	6 - 11
Übungen	6 - 12
 7. Elementare Datenstrukturen	 7 - 1
7. 1. Allgemeiner Strukturtyp	7 - 1
7. 2. Bitfelder	7 - 3
7. 3. Varianten	7 - 4
7. 4. Initialisierung	7 - 4
7. 5. Eigene Typbezeichner	7 - 5
7. 6. Zeiger auf Strukturen	7 - 6
7. 6. 1. Lineare Listen	7 - 6
7. 6. 2. Binäre Bäume	7 - 12
Übungen	7 - 15

Anhang

ASCII-Tabelle

1. Einführung

1. 1. Historischer Überblick

Die Programmiersprache C ist eine moderne und sehr leistungsfähige Programmiersprache, die sich für technisch-wissenschaftliche Anwendungen, den kommerziell administrativen Einsatz und für die Systemprogrammierung sehr sehr gut eignet. Maschinennahe Programme, die früher nur mit Assembler zu bewältigen waren, können nun häufig mit Hilfe von C leichter und zeitsparender gelöst werden.

Brian Kerningham und *Dennis Ritchie* entwickelten die Programmiersprache C Anfang der 70er Jahre nach Einführung des von *Ken Thompson* und *Brian Kerningham* entworfenen Betriebssystems *UNIX*, das größtenteils in C geschrieben ist. Die erste Beschreibung erfolgte

1978 in dem Werk von Kerningham und Ritchie: *The C-Programming-Language*

1983 erfolgte die Standardisierung im American National Standards Institute (ANSI)

ANSI-C enthält

- Funktionsprototypen
- Schlüsselwörter
- void *-Zeiger
- eine Funktionenbibliothek
- Grenzwerte arithmetischer Datentypen

1983 entwickelte Bjarne Stroustrup die objektorientierte Erweiterung C++

1985 wurde der erste kommerzielle C-Compiler vorgestellt

seit 1989 liefert die Firma Borland Pascal- und C-Werkzeuge unter Windows mit integrierter Entwicklungsumgebung, Mausunterstützung, Compiler, Linker, Debugger und Assembler.

In einer Ankündigung, die die Computer-Industrie verblüffte, haben Thompson, Ritchie und Kerningham zugegeben, daß das von Ihnen geschaffene Betriebssystem Unix und die Programmiersprache C ein raffinierter Aprilscherz sind, der sich über 20 Jahre am Leben erhalten hat. Bei einem Vortrag vor dem letzten UnixWorld-Software-Entwicklungsforum enthüllte Thompson:

„1969 hatte AT&T gerade die Arbeit am GE/Honeywell/AT&T-Multics-Projekt beendet. Brian und ich experimentierten zu dem Zeitpunkt mit einer früheren Pascal-Version von Professor Niklas Wirth vom ETH-Laboratorium in der Schweiz und waren beeindruckt von seiner eleganten Einfachheit und Mächtigkeit. Dennis hatte gerade „Der Herr der Klinge gelesen“, eine spöttische Parodie auf Tolkiens große Trilogie „Der Herr der Ringe“. Im Übermut beschlossen wir, Parodien zur Multics-Umgebung und zu Pascal zu verfassen. Dennis und ich waren für die Betriebssystemumgebung verantwortlich. Wir sahen uns Multics an und entwarfen ein neues System, das so komplex und kryptisch wie möglich sein sollte, um die Frustration der gelegentlichen Nutzer zu maximieren. Wir nannten es Unix in Anspielung auf Multics und fanden es auch nicht gewagter als andere Verballhornungen. Danach entwickelten Dennis und Brian eine wirklich perverse Pascal-Version namens „A“. Als wir bemerkten, daß einige Leute tatsächlich versuchten, in A zu programmieren, fügten wir schnell einige zusätzliche Fallstricke hinzu, nannten es B, - unter Übernahme vieler Konzepte der von Martin Richards am MIT (Massachusetts Institute of Technology) 1967 entwickelten Sprache BCPL (Basic Combined Programming Language) - und schließlich C. Wir hörten damit auf, als wir eine saubere Übersetzung der folgenden Konstruktion erhielten:

```
for (;P("\n"),R--;P("|"))
for (e=C;e--;P(" _"+(*u++/8)%2))
P(" _"+(*u/4)%2);
```

Der Gedanke, daß moderne Programmierer eine Sprache benutzen würden, die solch eine Anweisung zuließ, lag jenseits unseres Vorstellungsvermögens. Wir dachten allerdings daran, alles den Sowjets zu verkaufen, um ihren Computerfortschritt 20 Jahre und mehr zu verhindern.

Unsere Überraschung war groß, als dann AT&T und andere US-Unternehmen tatsächlich begannen, Unix und C zu verwenden! Sie haben 20 weitere Jahre gebraucht, genügend Erfahrung zu sammeln, um einige bedeutungslose Programme in C zu entwickeln, und das mit einer Parodie auf die 60er Jahre! Dennoch sind wir beeindruckt von der Hartnäckigkeit (falls nicht doch Gemeinsinn) des gewöhnlichen Unix- bzw. C-Anwenders. Jedenfalls haben Brian, Dennis und ich in den letzten Jahren nur in Pascal auf einem Apple Macintosh programmiert, und wir fühlen uns echt schuldig an dem Chaos, der Verwirrung und dem wirklich schlechten Programmierstil, der von unserem verrückten Einfall vor so langer Zeit ausging.“



Ken Thompson



Brian Kerningham



Dennis Ritchie

Namhafte Unix- und C-Anbieter und Benutzer, einschließlich AT&T, Microsoft, Hewlett-Packard, GTE, NCR und DEC haben vorläufig jede Stellungnahme abgelehnt. Borland international, ein führender Anbieter von Pascal- und C-Werkzeugen, einschließlich der populären Turbo Pascal, Turbo C und Turbo C++, meinte, sie hätten diesen Verdacht schon seit Jahren gehegt und würden nun dazu übergehen, ihre Pascal-Produkte zu verbessern und weitere Bemühungen um die C-Entwicklung zu stoppen. Ein IBM-Sprecher brach in unkontrolliertes Gelächter aus und verschob eine hastig einberufene Sitzung über das Schicksal der RS-6000 und konstatierte lediglich: "VM wird nun wirklich bald verfügbar sein." In einer kryptischen Stellungnahme bemerkte Professor Wirth von der ETH und Vater der strukturierten Sprachen Pascal, Modula-2 und Oberon, daß P. T. Barnum recht gehabt habe.

In ähnlicher Weise haben gewöhnlich gut unterrichtete Kreise durchblicken lassen, daß eine vergleichbare Enthüllung von William Gates bezüglich MS-DOS und der Windows-Umgebung zu erwarten ist. Und: IBM-Sprecher dementieren nun schon, daß das Virtual Machine Produkt (VM) ein interner Rohrkrepiere sein könnte.

Dieser Beitrag stammt aus einer der zahlreichen NetNewsGruppen von Bernard L. Hayes und wurde vom Englischen ins Deutsche übersetzt unter dem Titel:

Erfinder geben zu: alles Quatsch.

1. 2. Einführendes Beispiel

Als Einführung wollen wir uns ein C-Programm ansehen, das folgende Aufgabe löst:

Wir möchten wissen, wieviel Pfund und wieviel Rubel man am 3. September 2002 für 1€, 2€, ... 10€ bekam. Die Werte sollen in Form einer Währungstabelle auf dem Bildschirm ausgegeben werden.

Der Programmtext wurde nach der Standardvorlage eines Editors farbig geschrieben. Kommentare sind grün, Schlüsselwörter blau, sonstiger Text schwarz dargestellt.

```
// Mein erstes C-Programm rechnet Euro in Pfund und Rubel um
// Stand: 3. 9. 2002

//Praeprozessoranweisungen

//Headerdatei

#include <stdio.h>                // Bibliothek für Ein- und Ausgabefunktionen

//Konstantendefinitionen

#define Kurs1 0.6                // Umrechnungsfaktor 1 Euro = 0,6 Pfund
#define Kurs2 30                // Umrechnungsfaktor 1 Euro = 30 Rubel

//Hauptprogramm

void main() {

    // Deklarationsteil
    int Euro_Anfang, Euro, Euro_Ende;
    float Pfund, Rubel;

    // Anweisungsteil
    Euro_Anfang = 1;              // Wertzuweisungen
    Euro = Euro_Anfang;
    Euro_Ende = 10;

    printf("Waehrungstabelle\n\n\n"); // Textausgaben
    printf("Euro Pfund Rubel\n\n");

    while (Euro <= Euro_Ende) {    // while-Schleife
        Pfund = Euro*Kurs1;        // Zuweisungen von Ausdruecken
        Rubel = Euro*Kurs2;

        printf("%3d %5.2f %6.2f\n", Euro++, Pfund, Rubel);
        // formatierte Ausgabe der Ergebnisse
    }

    getchar( );                  // Programmstop, bis eine Tastatureingabe erfolgt
}
```

Zum Ablauf des Programmes ist am Computer

1. die integrierte Entwicklungsumgebung aufzurufen,
2. der Programmtext zu editieren,
3. der Text unter einem Dateinamen z. B. **Uebung.c** abzuspeichern*,
4. der Compiler aufzurufen.

Der Compiler (Übersetzer) ist ein Programm, das den Text in eine Folge maschinenverständlicher Befehle umcodiert, sofern keine syntaktischen Fehler vorliegen. In diesem Fall entsteht ein sogenanntes Objektprogramm unter dem Namen **Uebung.obj**. Andernfalls gibt der Rechner Fehlermeldungen aus, und der Programmtext muß korrigiert, erneut abgespeichert und übersetzt werden.

Ist das Programm syntaktisch korrekt, so kann

5. das Binden (Linken) erfolgen.

Dabei verbindet der Binder (Linker) die Objektdateien, sofern mehrere vorliegen, und interne Programmodule zu einem ausführbaren Programm mit dem Namen **Uebung.exe**.

6. Die Ausführung (Execution)

erfolgt nach dem Binden automatisch und kann durch den Aufruf **Uebung** wiederholt werden. Das Programm läuft dann sequentiell ab, sofern keine Laufzeitfehler z. B. Division durch 0, Speicherbereichsüberschreitung usw. einen vorzeitigen Abbruch mit Fehlermeldung hervorrufen. In einer

6. Testphase

ist fernerhin anhand der Ergebnisse zu kontrollieren, ob das Programm logisch korrekt arbeitet, auch bei anderen Eingabedaten zu sinnvollen Ergebnissen führt und vor allem keine Endlosschleifen enthält.

Beispiel:

Währungstabelle

	Euro	Pfund	Rubel
1	0.60		30.00
2	1.20		60.00
3	1.80		90.00
4	2.40		120.00
5	3.00		150.00
6	3.60		180.00
7	4.20		210.00
8	4.80		240.00
9	5.40		270.00
10	6.00		300.00

1. 3. Zeichenvorrat

* Statt der Endung ".c" wird in C++ die Endung ".cpp" verwendet. Falls der C-Compiler die Dateierweiterung ".c" ohne Probleme akzeptiert, sollte man sie an Stelle von ".cpp" verwenden, da viele C-Compiler, besonders UNIX-Werkzeuge recht empfindlich auf Dateierweiterungen reagieren.

Ein Programm ist im Grunde eine Folge von Zeichen aus einem Zeichenvorrat, die nach bestimmten grammatikalischen Regeln, der sogenannten Syntax, zu bilden ist.

Der Zeichenvorrat der Programmiersprache C umfaßt:

1. die Buchstaben <letters> von a bis z, A bis Z und den Unterstrich _ <underscore>
2. die Ziffern <digits> von 0 bis 9
3. Sonderzeichen <special characters>
4. Trennzeichen <whitespace characters>

Folgende Tabelle enthält die, für spezielle Zwecke reservierten Sonderzeichen:

Zeichen	Name	Zeichen	Name
!	Ausrufzeichen <exclamation point>	;	Semikolon <semicolon>
"	Anführungszeichen <quotation mark>	#	Nummerzeichen <number sign>
<	kleiner als <less than>	>	größer als <greater than>
:	Doppelpunkt <colon>	=	Gleichheitszeichen <equals>
%	Prozentzeichen <percent sign>	?	Fragezeichen <question mark>
&	Kommerzielles Und <ampersand>		Vertikalstrich <vertical line>
'	Hochkomma <apostrophe>	\	Fluchtsymbol <backslash>
(	linke, runde Klammer <left parenthesis>	)	rechte, runde Klammer <right parenthesis>
[	linke, eckige Klammer <left bracket>	]	rechte, eckige Klammer <right bracket>
{	linke, geschweifte Klammer <left brace>	}	rechte, geschweifte Klammer <right brace>
*	Stern <asterisk>	^	Zirkumflex <caret>
+	Pluszeichen <plus>	-	Unterstrich <underscore>
-	Bindestrich/Minuszeichen <hyphen/minus>	~	Tilde <tilde>
.	Punkt <point>	,	Komma <comma>
/	Schrägstrich <slash>		

Weitere Symbole z. B. Umlaute ä, ö, ü, @ <commercial at> dürfen nur innerhalb von Kommentaren, Zeichenketten <strings> oder als Zeichenkonstante verwendet werden.

Trennzeichen werden in Zeichenketten, als Zeichenkonstante, hauptsächlich jedoch dazu verwendet, um ein Programm gut lesbar zu machen und übersichtlich zu gestalten. Sie werden in diesem Fall vom Compiler ignoriert, so als wären sie gar nicht vorhanden. Man unterscheidet folgende Trennzeichen

- das Leerzeichen <blank>
- die horizontale Tabulation <horizontal tabulation>
- die vertikale Tabulation <vertical tabulation>
- den Zeilenvorsprung <line feed>
- den Seitenvorschub <form feed>
- den Wagenrücklauf <carriage return>

1. 4. Wörter

Eine Zusammenfassung von mehreren Zeichen zu einer logischen Einheit wird als Wort bezeichnet. C kennt fünf Klassen von Wörtern

1. Bezeichner <identifiers>
2. Konstanten <constants>
3. Operatoren <operators>
4. sonstige Trenner <separators>
5. reservierte Wörter/Schlüsselwörter <keywords>

Bezeichner sind Namen beliebiger Länge für Funktionen, Variablen usw. aus Buchstaben, Underscorezeichen und Ziffern. Es gibt drei Einschränkungen

1. das erste Zeichen muß ein Buchstabe oder Underscore-Zeichen sein,
2. nur die ersten n Zeichen sind signifikant, wobei $n \geq 8$ (compilerabhängig),
3. es wird zwischen Groß- und Kleinbuchstaben unterschieden (in C++ einstellbar)

Bei den reservierten Wörtern handelt es sich um spezielle Namen, die bereits eine vordefinierte Bedeutung haben und deshalb nicht für andere Zwecke benutzt werden dürfen.

Tabelle der Schlüsselwörter in ANSI-C:

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

Je nach Compilertyp kommen noch weitere Schlüsselwörter hinzu z. B. in Borland C++ asm, far, near usw.

Der Editor von Borland-C++ läßt sich so einstellen, daß Schlüsselwörter durch Fettschrift hervorgehoben werden.

1. 5. Kommentare

Um ein Programm verständlich, übersichtlich und besser lesbar zu machen, sollte es mit einem erklärenden Text (Kommentar <comment> versehen werden. Für Kommentare gilt:

1. Jeder Kommentar beginnt mit dem Symbol `/*` , kann sich über mehrere Zeilen erstrecken und endet mit `*/`. In C++ kann ein Kommentar auch mit zwei Schrägstrichen `//` angelegt werden. Er kann an einer beliebigen Stelle beginnen und erstreckt sich bis zur nächsten neuen Zeile.
2. Der Kommentar wird vom Compiler wie ein Trennzeichen <white space> behandelt, also beim Übersetzen völlig ignoriert.
3. Kommentare dürfen nicht ineinander verschachtelt werden.
`/* Kommentar in einem /* Kommentar */ ist nicht erlaubt */`

Der Borland-C++-Editor kann Kommentare automatisch kursiv schreiben.

1. 6. C-Präprozessor

Vor dem Ablauf eines Programmes werden von dem sogenannten Präprozessor Direktiven ausgeführt. Das sind Anweisungen zum Einfügen anderer Quelldateien, für Makrosubstitutionen und zur bedingten Compilierung. Sie werden gewöhnlich am Anfang des Programmes geschrieben.

1. Das Einfügen anderer Quelldateien wird durch die `#include`-Direktive veranlaßt.

Beispiele:

`#include <stdio.h>` stellt dem Compiler Standardfunktionen für die Ein- und Ausgabe von Daten zur Verfügung, z. B.

`printf ( ... );` zur Ausgabe von Daten auf dem Bildschirm
`scanf ( ... );` zur Eingabe von Daten von der Tastatur

`#include <conio.h>` liefert Funktionen für die Bildschirmsteuerung z. B.

`clrscr ( );` zum Löschen des Bildschirminhaltes

2. Mit der Direktive `#define NAME` ersatz wird jedes Vorkommen des Textes NAME durch den Text Ersatz ersetzt. Der Text NAME wird üblicherweise mit Großbuchstaben geschrieben und heißt Makroname, der Text ersatz heißt Makrosubstitution. Mit `#undef NAME` kann die vorher definierte Substitution wieder aufgehoben werden.

Beispiel:

`#define PI 3.1415926535897932384626`

3. Mit Hilfe der Direktiven `if`, `endif`, `else`, `ifdef` und `ifndef` lassen sich C-Programme rechner- und betriebssystemunabhängig schreiben. Man kann damit für jedes System individuellen Programmcode schreiben, aber nur den jeweils passenden übersetzen lassen.

```

void main ( ) {

    #if BS == DOS
        ...      // wird übersetzt, wenn vorher #define BS DOS stand
    #else if BS == UNIX
        ...      // wird übersetzt, wenn vorher #define BS UNIX stand
    #else
        ...      // wird übersetzt, wenn keins von beiden stand
    #endif
        ...      // wird in jedem Fall übersetzt
}

```

Statt **#if** können auch die Direktiven **#ifdef** NAME oder **#ifndef** NAME benutzt werden, die testen, ob vorher **#define** NAME ... angegeben wurde.

1. 7. main-Funktion

1. Ein C-Programm besteht aus einer oder mehreren (meist kleinen) Funktionen. Eine Funktion besteht aus einem Kopf, gefolgt von einem Block.
2. Der Kopf ist ein typisierter Funktionsname mit einem runden Klammernpaar, das eine Parameterliste enthalten kann. Der Typ entspricht dem Datentyp des Funktionswertes z. B. **int** bei ganzzahligem Wert. Eine Funktion kann auch 0-wertig sein; dann wird ihr der Typ **void** zugeordnet.
3. Ein Block ist eine von geschweiften Klammern eingerahmte Folge von Typvereinbarungen und Anweisungen, die beim Programmlauf sequentiell ausgeführt werden.
4. Jedes Programm enthält eine Funktion mit dem Namen **main** vom Typ **int** oder **void**. Beim Programmstart werden zuerst die Funktion **main** aufgerufen und deren Anweisungen nacheinander ausgeführt. Von dort aus kann der Programmlauf von einer beliebigen Stelle zu einer anderen Funktion verzweigen, die ihrerseits wieder eine weitere Funktion aufruft. Nach Durchlaufen einer Funktion wird der Programmlauf mit der dem Aufruf folgenden Anweisung fortgesetzt. Folglich endet der Programmlauf mit Ausführung der letzten Anweisung vor der schließenden, geschweiften Klammer der Funktion **main**.

Beispiel: Wir betrachten den Ablauf des folgenden Programmes:

```

#include <stdio.h>
#include <conio.h>

void Fkt1 ( ) {                                //5, 12
    printf("Wir sind in Fkt1!\n");              //6, 13
}

void Fkt2 ( ) {                                //9
    printf("Wir sind in Fkt2!\n");              //10
    Fkt1( );                                    //11
    printf("Wir sind wieder in Fkt2!\n");       //14
}

```

```

void main ( ) {                                     //1
    clrscr ( );                                       //2
    printf("Es geht los!\n");                         //3
    Fkt1( );                                          //4
    printf("Wir sind wieder im Hauptprogramm!\n");    //7
    Fkt2( );                                          //8
    printf("Das war \'s!");                          //15
    getchar( );                                       //16
}

```

Der Programmlauf beginnt mit der Ausführung des Hauptprogrammes main (1). Nach Löschen des Bildschirms (2) wird der Text "Es geht los!" ausgegeben, und es erfolgt ein Zeilenvorschub (3).

Der Funktionsaufruf Fkt1() bewirkt einen Programmsprung (4) zur Funktion Fkt1() (5), die den Text "Wir sind in Fkt1!" mit Zeilenvorschub liefert (6). Nach Beendigung von Fkt1 erfolgt ein Rücksprung in das aufrufende Hauptprogramm zu der dem Aufruf folgenden Anweisung (7). Nach deren Ausführung, der Ausgabe "Wir sind wieder im Hauptprogramm!" mit Zeilenvorschub wird die Funktion Fkt2 aufgerufen (8).

Von dort geht es zur Funktion Fkt2() (9), die zuerst den Text "Wir sind in Fkt2!" mit einem Zeilenvorschub liefert (10). Danach wird von Fkt2 aus die Funktion Fkt1() erneut aufgerufen (11) und ausgeführt (12), (13). Nach Beendigung von Fkt1 erfolgt ein Rücksprung zur aufrufenden Funktion Fkt2 (14) und nach Ausgabe des Textes "Wir sind wieder in Fkt2!" mit Zeilenvorschub weiter zurück nach main (15).

Nach Ausführung der beiden letzten Anweisungen von main(), der Ausgabe "Das war 's!" (15) und der Stoppfunktion getchar() (16) endet main und damit das gesamte Programm.

1. 8. Datenausgabe

Ein Programm ist nur dann sinnvoll, wenn es die verarbeiteten Daten auch dem Anwender auf einem externen Gerät z. B. Bildschirm, Drucker, Speicher usw. ausgibt.

Die Funktion, die eine formatierte Ausgabe erzeugt, lautet

```
printf ( <format>, <arg1>, <arg2>, ... ).
```

Format bestimmt die Art und Weise, in der die Funktion printf ihre Argumente arg1, arg2, ... konvertiert, formatiert und anschließend ausgibt. Bei <format> handelt es sich um eine in doppelten Anführungsstrichen eingeschlossenen Zeichenfolge, die zum einen gewöhnliche Zeichen, zum anderen Formatschlüssel enthält.

Alle gewöhnlichen Zeichen werden ohne weitere Umwandlung an die Standardausgabe weitergeleitet und dort ausgegeben.

Ein Formatschlüssel beginnt immer mit einem Prozentzeichen und endet mit einem Buchstaben, der die Art der Umwandlung angibt. Dabei gehört der erste Formatschlüssel zum ersten Argument <arg1>, der zweite Formatschlüssel zum zweiten Argument <arg2> usw.

Erlaubt sind folgende Buchstaben:

d, u, o, x, X, c, s, f, e, E, g, G und das Sonderzeichen %.

Bedeutung der Schlüsselzeichen:

d	dezimale Zeichendarstellung
u	vorzeichenlose, dezimale Zeichendarstellung
o	vorzeichenlose, oktale Zeichendarstellung
x, X	vorzeichenlose, hexadezimale Zeichendarstellung
c	Einzelzeichen
s	Zeichenkette
f	Festpunktzahl in der Form -ddd.dddddd
e, E	Gleitpunktzahl in der Form -d.ddddddE±dd
g, G	kürzere der Formen von f oder e
%	das Zeichen % wird ausgegeben

Zwischen % und dem Formatbuchstaben sind noch folgende Angaben zur Formatierung nacheinander möglich:

-	die Ausgabe erfolgt linksbündig
<natürliche Zahl>	minimale Feldbreite
. <nat. Zahl>	Anzahl der Zeichen einer Zeichenkette oder Nachkommastellen einer Zahl
h	Argumenttyp short
l	Argumenttyp long

Bemerkungen und Beispiel:

- Bei Verwendung von x, e, g werden Klein-, bei Verwendung von X, E, G Großbuchstaben ausgegeben.
- Die Benutzung von printf erfordert das Präprozessorkommando: `#include <stdio.h>` das den Zugriff auf die Standardeingabe- und Standardausgabefunktionen ermöglicht.
- Zu jedem Argument muß ein Formatschlüssel erscheinen. Fehlen Formatschlüssel, so erfolgt eine Fehlermeldung, gibt es mehr Formatschlüssel als Argumente, so werden die überzähligen Formatschlüssel ignoriert.

Beispiel:

```
printf (" i = %10d und j = %5X, x = %5.2f und c = %8.2e\n", i, j, x, 300000);
```

bedeutet folgende Ausgabe:

1. Textausgabe: i =
2. Ausgabe der Variablen i mit dem Schlüssel %10d,
3. Textausgabe: und j =
4. Ausgabe der Variablen j mit dem Schlüssel %5X,
5. Textausgabe: x =
6. Ausgabe der Variablen x mit dem Schlüssel %5.2f,
7. Textausgabe: und c =
8. Ausgabe der Konstanten 300000 mit dem Schlüssel %8.2e,
9. Zeilenvorschub.

Mit den Argumenten werden der Funktion `printf` Binärworte übergeben, die je nach dem Schlüssel als Dualzahl mit oder ohne Vorzeichenbit, als ASCII-Code eines Zeichens, einer Zeichenfolge oder als normalisierte Gleitpunktform einer Dualzahl interpretiert werden. `%10d` zu `i` bedeutet, daß das Bitmuster von `i` als Dualzahl mit Vorzeichenbit interpretiert und auf 10 Schreibstellen rechtsbündig geschrieben wird.

`%-5X` zu `j` bedeutet, daß der Hexadezimalwert von `j` auf 5 Schreibstellen linksbündig ausgegeben wird.

`%10.2e` zu `x` bedeutet, daß das Binärwort von `x` als normalisierte, duale Gleitpunktzahl interpretiert und mit zwei Nachkommastellen ausgegeben wird. Dementsprechend führt der Schlüssel `%6.2e` für 300000 zur Ausgabe `3.00e+05`.

Weitere Ausgabefunktionen:

Zur Ausgabe eines einzelnen Zeichens `c` oder einer Zeichenfolge `str` über die Standardausgabe gibt es die Funktionen

`putchar ( c )` und `puts ( str )`

1. 9. Dateneingabe

Will man das letzte Programm statt mit dem Wert 'A' mit einem anderen Wert z. B. 'B' arbeiten lassen, so ist nach den bisherigen Kenntnissen die Zuweisung `char c = 'A'`; durch die Zuweisung `char c = 'B'`; zu ersetzen. Das bedeutet, daß das Programm auch neu übersetzt und erneut zum Ablauf gebracht werden muß. Allerdings gibt es in jeder Programmiersprache Lesefunktionen, die bewirken, daß der Programmlauf gestoppt wird, damit dann eine Dateneingabe von der Tastatur erfolgen kann. Nach der Eingabe läuft das Programm weiter und arbeitet mit den eingegebenen Daten. Soll der Programmlauf mit anderen Daten wiederholt werden, so braucht er nur neu gestartet zu werden.

Die Funktion zur formatierten Dateneingabe über die Standardeingabe (Tastatur) lautet:

`scanf ( <format>, <arg1>, <arg2>, ... ).`

Beispiel:

```
int i;
scanf ("%d", &i);
```

Beachte:

Vor jeden Variablennamen ist der Adreßoperator `&` zu setzen!

Für die Formatzeichenfolge gilt:

- Leer- und Tabulatorzeichen werden ignoriert
- Textzeichen außer `%` verlangen ein entsprechendes Zeichen bei der Eingabe

- Eine Formatangabe beginnt mit einem %, und endet mit einem Formatbuchstaben (s. Tab. bei printf)
- Zwischen % und Formatbuchstaben kann eine Zahl zur Vorgabe einer maximalen Feldweite, ein Buchstabe h, l oder L zur Angabe eines short bzw. long-Typs oder ein * stehen; * bedeutet, daß ein Eingabewert zwar eingelesen jedoch nicht zugewiesen wird.

Beispiel:

```
// In diesem Programm wird eine ganze Zahl x eingelesen und ihr Oktal-, Dezimal- und
// Hexadezimalwert wieder ausgegeben
```

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

```
void main( )    {
    int x;
    clrscr ( );
    printf ( " Bitte eingeben: x=<ganze Zahl>\n" );
    scanf ( "x=%d", &x );
    getchar ( );
    printf ( "Oktalwert: %o  Dezimalwert: %d  Hexadezimalwert: %X", x, x, x);
    getchar ( );
}
```

Zur Eingabe eines einzelnen Zeichens c über die Standardeingabe gibt es die Funktion

getchar ()

Damit läßt sich ein einzelnes Zeichen von der Tastatur einlesen, einer Variablen vom Typ char oder int zuweisen und im Programm weiter verwerten.

In folgendem Beispiel wird die getchar-Funktion auf die letzte Übung zur Eingabe eines Wertes für c angewandt:

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

```
void main( )    {
    char c;
    clrscr ( );
    printf(" Bitte ein Zeichen eingeben:");
    c = getchar ( );    // das Programm wartet auf Eingabe eines Zeichen
    getchar ( );        // Dummy zum Überlesen des LF-Steuerzeichens
    clrscr ( );         // löscht den Bildschirm wieder
    ...                // hier kann c verwendet werden
    ...
}
```

Übung 1_1: Schreiben Sie das Programm derart um, daß es die Euro- und Rubelwerte für 1 Pfund, 2 Pfund, ... 10 Pfund berechnet und in Tabellenform ausgibt:

Währungstabelle		
Pfund	Euro	Rubel
1	1.67	50.00
...	...	...

Übung 1_2: Ersetzen Sie die Sternchen *** in den folgenden print-Befehlen durch Format-schlüssel, so daß das Programm

- die Werte der Variablen c, -c, i, -i und x mit den Schlüsseln d, u, o, X und c,
- den Wert der Variablen x mit den Schlüsseln f, E und G in Form einer Tabelle mit begleitendem Text ausgibt.

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

```
void main ( ) {
    char c = 'A';    // Initialisierung der Variablen c, i und x
    int i = 12345;
    float x = 12345;
    clrscr( );
    printf("          Beispiele formatierter Ausgaben: \n\n");
    printf("          dezimal vorzeichenlos oktal hexadezimal ASCII \n\n");
    printf("char positiv:   ***      ***      ***      ***      *** \n", c, c, c, c, c);
    printf("char negativ:  ***      ***      ***      ***      *** \n", -c, -c, -c, -c, -c);
    printf("int positiv:    ***      ***      ***      ***      *** \n", i, i, i, i, i);
    printf("int negativ:   ***      ***      ***      ***      *** \n", -i, -i, -i, -i, -i);
    printf("float:         ***      ***      ***      ***      *** \n\n", x, x, x, x, x);
    printf("          f-Format          e-Format          g-Format \n\n");
    printf("float:         ***          ***          *** \n\n", x, x, x);
    getchar( );
}
```

In diesem Beispiel werden versuchsweise auch solche Schlüssel verwendet, die nicht direkt zum Datentyp der Variablen passen. In diesem Fall ergibt sich evtl. kein sinnvolles Ergebnis.

Bei Ausgabe eines Zeichens vom Typ char mit einem int-Schlüssel wird das Zeichen durch seinem ASCII-Wert ersetzt z. B. hat 'A' die Nr. 65.

In Borland-C++ ist es möglich, ein Zeichen mit einem Minuszeichen zu versehen, weil dort Zeichen in Worten von 2 Bytes gespeichert werden; das 1. Byte enthält dann das Vorzeichen in Form von Nullen (+) oder Einsen (-).

Bei Ausgabe einer Zahl vom Typ int mit dem Schlüssel c wird das ASCII-Zeichen des niederwertigen Bytes ausgegeben.

Die vorzeichenlose Darstellung mit den Schlüsseln u, o, x oder X von negativen Zahlen erfolgt mit mindestens 10 Dezimal-, 11 Oktal- und 8 Hexadezimalstellen.

Beachte: Die Ausgabe eines float-Typs mit einem typfremden Schlüssel ergibt evtl. kein sinnvolles Ergebnis.

Mit dem Compiler Visual C++ 6.0 erhält man folgendes Ergebnis:

Beispiele formatierter Ausgaben:

	dezimal	vorzeichenlos	oktal	hexadezimal	ASCII
char positiv:	65	65	101	41	A
char negativ:	-65	4294967231	37777777677	FFFFFFBF	]
int positiv:	12345	12345	30071	3039	9
int negativ:	-12345	4294954951	37777747707	FFFCFC7	A
float:	0	1086856320	0	40C81C80	

	f-Format	e-Format	g-Format	
float:	12345	1.2345E+04	12345	0

Beachte: Offensichtlich ergeben die Schlüssel d, o und c für den Typ float kein sinnvolles Ergebnis.

Das hexadezimale Ergebnis 40C81C80 für x kann als hexadezimale Darstellung des internen Bitmusters interpretiert werden.

Dies berechnet man folgendermaßen:

Zuerst wird der Hexadezimalwert von 12345 bestimmt. Er lautet 3039. Damit bildet man den Dualwert 11000000111001.

Die normalisierte Gleitpunktdarstellung des Dualwertes von 12345 lautet dann

$$1.1000000111001 * 10^{1101}.$$

Zur Speicherung der Gleitpunktdarstellung in einem Wort von 4 Bytes wird zuerst die Konstante 1023 zum Exponenten addiert. Die Summe heißt Charakteristik C und hat in diesem Beispiel den Wert 10000001100.

Man erhält das interne Bitmuster durch Hintereinanderfügen des Vorzeichenbits der Mantisse, der Charakteristik in den folgenden 11 Bits und des Mantissentails, der hinter dem Dezimalpunkt steht, in den restlichen 20 Bits, also

$$0100\ 0000\ 1100\ 1000\ 0001\ 1100\ 1000\ 0000 = 40C81C80 = 12345$$

Berechnet man den Dezimalwert dieser Dualzahl, so erhält man genau das Ergebnis des u-Schlüssels 1086856320.

Übung 1_3: Wiederholen Sie Übung 1_2 für beliebige Variablen durch Verwendung der Eingabefunktionen getchar für c und scanf für i und x.

2. Elementare Ausdrücke

2. 1. Konstanten

Unter einer Konstanten versteht man ein Wort, das eine unveränderliche Größe z. B. eine bestimmte Zahl, ein bestimmtes Zeichen oder einen Text bezeichnet.

Jede Konstante besitzt einen Datentyp, der festlegt, wieviel Bytes zur Speicherung des Wertes zur Verfügung gestellt werden, in denen die Konstante binär codiert abgelegt wird.

C unterscheidet fünf Arten von Konstanten

- Integerkonstanten <integer constants>
- Gleitkommakonstanten <floating-point constants>
- Aufzählungskonstanten <enumerable constants>
- Zeichenkonstanten <character constants>
- Zeichenketten <string constants>

2. 1. 1. Integerkonstanten

Alle Zahlen ohne Dezimalpunkt sind Integerkonstanten (ganzzahlige Konstanten). Sie können vorzeichenbehaftete Dezimalzahlen mit Ziffern von 0 bis 9, Oktalzahlen mit Ziffern von 0 bis 7 oder Hexadezimalzahlen mit Ziffern von 0 bis 9 und A bis F (oder a bis f) sein.

Zur Unterscheidung von einer Dezimalzahl enthält jede Oktalzahl eine führende Null und jede Hexadezimalzahl zusätzlich ein zweites Zeichen x oder X. Folglich darf eine Dezimalzahl außer der Null selber nie mit einer Null beginnen.

Beispiele:

1328 -7 512 -15764

010 (entspricht dem dezimalen Wert 8)

076663 (" 32179)

0xD, 0XD, 0xd oder 0Xd (entsprechen dem dezimalen Wert 13)

0x4cf1 oder 0x4CF1 usw. (" 19697)

Intern wird der Dualwert der Integerkonstanten je nach Größe in 2 oder 4 Bytes gespeichert. Die Art der Speicherung definiert einen Datentyp, der von ihrem Wert, ihrer Basis und dem verwendeten Rechnersystem abhängt.

Borland-C unterscheidet für Integerkonstanten die Datentypen

char (synonym: signed char) für Zeichen (optativ mit Vorzeichen),

unsigned char für Zeichen ohne Vorzeichen,

int (synonym: short, short int) für max. 2 Bytes lange, ganze Zahlen,*

unsigned int für derartige, nicht negative Zahlen,

long (synonym: long int) für mehr als 2, max. 4 Bytes lange, ganze, Zahlen,

unsigned long (synonym: unsigned long int) für derartige, nicht negative Zahlen,

* andere Systeme verwenden 2 Bytes bei short und 4 Bytes bei int (UNIX)

Der Datentyp einer Integerkonstanten hängt zwar von dem verwendeten System ab, stimmt jedoch in den meisten Punkten mit den Angaben in folgender Tabelle überein. Gehört eine Konstante zu mehreren Wertebereichen, so ist ihr Datentyp der mit der geringeren Bytezahl.

Dezimalkonstante	Datentyp	Bytes
-128 bis 127	char	1
128 bis 255	unsigned char	1
-32768 bis 32767	int	2
32768 bis 65535	unsigned int	2
-2.147.483.648 bis 2.147.483.647	long	4
2.147.483.648 bis 4.294.967.295	unsigned long	4
>4.294.967.295 oder <-2.147.483.648	wird zu unsigned long abgeschnitten	4

Oktalkonstante	Datentyp	Bytes
-0100000 bis 077777	int	2
0100000 bis 0177777	unsigned int	2
-020000000000 bis 01777777777	long	4
040000000000 bis 03777777777	unsigned long	4
<-020000000000 oder >03777777777	wird zu unsigned long abgeschnitten	4

Hexadezimalkonstante	Datentyp	Bytes
-0x8000 bis 0x7FFF	int	2
0x8000 bis 0xFFFF	unsigned int	2
-0x80000000 bis 0x7FFFFFFF	long	4
0x80000000 bis 0xFFFFFFFF	unsigned long	4
<-0x80000000 oder >0xFFFFFFFF	wird zu unsigned long abgeschnitten	4

Beispiel:

12345 ist vom Datentyp int und wird in 2 Bytes = 16 Bits gespeichert.
 = 0x3039 = 0011 0000 0011 1001

-123.456.789 ist vom Datentyp long und wird in 4 Bytes = 32 Bits gespeichert.
 = -0xCD5595 = - 0000 0000 1100 1101 0101 0101 1001 0101 =
 1111 1111 0011 0010 1010 1010 0110 1011 (Zweierkomplement)

Bei Verwendung von Konstanten als Funktionsparameter kann es vorkommen, daß die Syntaxregeln einen zum Konstantenwert z. B. 123 unpassenden Datentyp z. B. einen 4-Byte-Typ verlangen. Durch Anhängen eines Suffixes u oder U an eine Konstante vom Typ int bzw. long wird dieser der Datentyp unsigned int bzw. unsigned long zugeordnet, durch Anhängen eines Suffix l oder L der Datentyp long int bzw. unsigned long gebildet, beide Suffixe führen zum Typ unsigned long.

Beispiele:

12345L (aus int wird long) 12345U (aus int wird unsigned int)
 12345UL (aus int wird unsigned long)

2. 1. 2. Gleitkommakonstanten

Eine Gleitkommakonstante besteht aus fünf Bestandteilen:

- Vorkommastellen
- Dezimalpunkt
- Nachkommastellen
- e oder E und ein Integerexponent mit oder ohne Vorzeichen (optional)
- Suffix: f, F oder l, L (optional)

Die Vor- und Nachkommastellen dürfen weggelassen werden (aber nicht beide), ferner dürfen der Dezimalpunkt oder e bzw. E mit dem Exponenten weggelassen werden (aber nicht beide).

Beispiele:

-1.602e-19 bedeutet $-1,602 \times 10^{-19}$
 9.11E-31 " $9,11 \times 10^{-31}$
 3E5 " $3 \times 10^5 = 300000$
 1.0 kurz 1. " 1

Intern werden der Exponent und die Mantisse der nach einem mittlerweile einheitlichen Standard normalisierten Gleitkommazahl je nach Größe in einem Wort (4 Bytes), Doppelwort (8 Bytes) oder in 10 Bytes gespeichert.

Dies gilt auch dann, wenn der Wert ganzzahlig ist.

Borland C++ unterscheidet für Gleitkommakonstanten die Datentypen

float für 4-Byte-,

double für 8-Byte-,

long double für 10-Byte-Darstellung,

Der Datentyp einer Gleitkommakonstanten ist normalerweise **double**, bei Beträgen ab ca. 10^{38} **long double**, ab ca. 10^{4932} nicht mehr darstellbar (Fehlermeldung!).

Durch Anhängen eines Suffixes f oder F kann eine Umwandlung in den Datentyp **float**, durch Anhängen von l oder L in den Datentyp **long double** durchgeführt werden.

Suffix	Wertebereich	Datentyp	Bytes
f oder F	$3,4 \times 10^{-38}$ bis $3,4 \times 10^{38}$	float	4
	$1,7 \times 10^{-308}$ bis $1,7 \times 10^{308}$	double	8
l oder L	$3,4 \times 10^{-4932}$ bis $1,1 \times 10^{4932}$	long double	10

Beispiele:

3.14 ist vom Typ double, wird also in 8 Bytes gespeichert,

3.14f " float " 4 "

3.14L " long double " 10 "

2. 1. 3. Aufzählungskonstanten

Aufzählungskonstanten sind unterschiedliche Bezeichner die in einer Liste folgendermaßen definiert werden:

```
enum <Listenname>    { <Name0>, <Name1>, ... };
```

Der Datentyp einer Aufzählungskonstanten heißt Aufzählungstyp, der Wert der Konstanten ist die mit Null beginnende Position in der Liste, d. h. der Wert von <Name0> ist 0, von <Name1> ist 1 usw.

Aufzählungskonstanten können wie Konstante vom Typ `int` verwendet werden.

Beispiel:

```
enum Wochentag {Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag};
```

2. 1. 4. Zeichenkonstanten

Eine Zeichenkonstante ist eine in Apostrophen eingeschlossenes Symbol zur Darstellung von Zeichen aus dem ASCII-Zeichensatz. Ist das Zeichen darstellbar, also kein Steuerzeichen, so wird das Zeichen selbst als Symbol benutzt.

Beispiele: `'A'` `'b'` `'1'` `'('` `' '`

In jedem Falle kann eine Escape-Sequenz zur Repräsentation eines Zeichens benutzt werden. Diese beginnt mit einem Backslash (`\`), gefolgt von einem Buchstaben, einer Oktal- oder Hexadezimalzahl mit dem Präfix `x` oder `X`.

Beispiele: `'\n'` `'\t'` `'\53'` `'\x2B'`

Bei 256 ASCII-Zeichen betragen die Maximalwerte `\377` und `\xFF`.

Folgende Tabelle enthält die Borland C++ Escape-Sequenzen.

Sequenz	Bezeichnung	Wert dezimal	Wert oktal	Wert hexa	Bedeutung
<code>\a</code>	BEL	7	07	0x7	Alarmton <alert>
<code>\b</code>	BS	8	08	0x8	Rückschritt <backspace>
<code>\t</code>	HT	9	011	0x9	Horizontaltabulator <horizontal tab>
<code>\n</code>	LF	10	012	0xA	Zeilenvorschub <line feed>
<code>\v</code>	VT	11	013	0xB	Vertikaltabulator <vertical tab>
<code>\f</code>	FF	12	014	0xC	Seitenvorschub <form feed>
<code>\r</code>	CR	13	015	0xD	Wagenrücklauf <carriage return>
<code>\"</code>	"	34	042	0x22	Anführungszeichen <double quote>
<code>\'</code>	'	39	047	0x27	Apostroph <single quote>
<code>\?</code>	?	63	077	0x3F	Fragezeichen <question mark>
<code>\\</code>	\	92	0134	0x5C	Fluchtzeichen <backslash>
<code>\O</code>					O = Oktalwert von 0 bis 377
<code>\xH</code> oder <code>\XH</code>					H = Hexawert von 0 bis FF

Zur Speicherung einer Zeichenkonstanten genügt ein Byte, um den Dualwert ihrer ASCII-Nummer aufzunehmen. Der Datentyp ist **char**.

In Borland C++ wird der Dualwert einer Zeichenkonstante wie eine Integerkonstante vom Typ `int` gespeichert. Handelt es sich um ein Zeichen aus dem erweiterten Teil d. h. ist der Wert größer als 127, so wird das höherwertige Byte auf 0xFF gesetzt.

2. 1. 5. Zeichenketten

Unter einer Zeichenkette (String, Literal) versteht man eine Folge von Zeichenkonstanten, die durch Anführungszeichen " begrenzt werden.

Beispiele:

" C ist eine tolle Sprache!"

" // Besteht nur aus Leerzeichen

" !↔ \" aD@:\n5- " // Chaotische Zeichenkette

Intern werden die Zeichen einer Zeichenkette nacheinander gespeichert und als Abschluß das Nullzeichen \0 angehängt. Man kann auch eine leere Kette "" formulieren; diese enthält nur das Nullzeichen.

"Guten Morgen"

G	u	t	e	n		M	o	r	g	e	n	\0
---	---	---	---	---	--	---	---	---	---	---	---	----

im Dezimalcode

71	117	116	101	110	32	77	111	114	103	101	110	0
----	-----	-----	-----	-----	----	----	-----	-----	-----	-----	-----	---

im Hexacode

47	75	74	65	6E	20	4D	6F	72	67	65	6E	0
----	----	----	----	----	----	----	----	----	----	----	----	---

Beachte:

- Um ein Sonderzeichen mit der Tastatur zu schreiben, benutzt man die ASCII-Zeichensatztafel (siehe Anhang). Soll z. B. das Linienzeichen == ausgegeben werden, so gibt man dessen Nr. 205 unter gleichzeitigem Drücken der Alt-Taste in der Form 0205 auf dem rechten Ziffernblock der Tastatur ein. Beachten Sie, daß auf dem Bildschirm nicht das ASCII-Zeichen ==, sondern das Windowszeichen der Nr. 205 erscheint!
- Wird eine lange Zeichenkette mehrzeilig eingegeben, so gelangen auch die bei jeder Zeilenumschaltung durch die Eingabetaste erzeugten Steuerzeichen in die Zeichenkette. Zur Unterbindung dieser Steuerzeicheneingabe ist vor die Zeilenumschaltung ein Backslash zu setzen.

Beispiel:

"Dieser String wird so abgespeichert \
als stände er in einer Zeile."

2. 2. Variablen

Unter einer Variablen versteht man eine durch einen Bezeichner eindeutig identifizierte Größe, deren Wert im Gegensatz zu einer Konstanten an beliebiger Stelle im Programm geändert werden kann.

Jeder Variable ist in einer Variablendeklaration ein Datentyp zuzuordnen. Intern wird ihr dort vom Laufzeitsystem des Rechners ein adressierter, zum Typ passender Speicherbereich zugeordnet, in den ein mit dem Variablentyp verträglicher Wert z. B. eine Zahl, ein Zeichen oder eine Zeichenkette abgelegt werden kann.

Bevor eine Variable in einem C-Programm benutzt wird, muß sie deklariert (vereinbart) werden. Sie erhält dann ihren

1. Namen,
2. Datentyp,
3. wird einer Speicherklasse zugewiesen und
4. wird evtl. mit einem Anfangswert initialisiert.

Die Deklaration erfolgt durch Angabe des Datentyps, einer durch Kommata getrennten Namensliste und einem abschließenden Semicolon am Anfang des Programmes oder Blockes, in dem die Variablen gebraucht werden, oder für jede Variable einzeln in der Parameterliste einer Funktion.

Beim Aufruf einer Funktion werden die sogenannten **formalen** Parameter mit **aktuellen** Werten initialisiert; soll eine anderweitig deklarierte Variable mit einem Wert initialisiert werden, so folgen dem Namen noch ein Gleichheitszeichen und der Anfangswert.

Beachte: Zwei verschiedene Variable dürfen nur dann den gleichen Namen erhalten, wenn sie nicht im gleichen Block auf gleicher Ebene deklariert werden. Sie verhalten sich hinsichtlich Gültigkeitsbereich und Lebensdauer so, als besäßen sie unterschiedliche Namen.

Beispiele:

```
int i, j, k;
float x, y, z;
long double pi = 3.1415926535897932384626;
char Symbol = 'A'; // Deklarationen mit Initialisierung
int Stunde, Minute, Sekunde;

void Fkt ( int a, int b, int c) {      // formale Parameter a, b, c

    int v1, v2, v3;      // lokale Größen
    ...
}

void main () {
    ...
    Fkt (1, 2, 3); // Funktionsaufruf mit Initialisierung der Parameter a = 1, b = 2, c = 3
    ...
}
```

2. 3. Funktionsaufrufe

Vor der Programmierung einer umfangreichen Aufgabe ist in der Regel eine Problemanalyse und Aufteilung der Aufgabe in kleinere Teilaufgaben erforderlich. Die Teilaufgaben können von verschiedenen Programmierern in Form von Funktionen unabhängig voneinander programmiert und zu einem Gesamtprogramm zusammengefügt werden.

Zur Ausführung einer Funktion benötigt der Programmierer Kenntnis über

1. den Namen der Funktion,
2. ihren Datentyp,
3. Anzahl und Datentyp der Funktionsparameter.

Dazu genügt die Kenntnis des Funktionskopfes, während die Art des Rumpfes nicht bekannt zu sein braucht und als Privatsache des Programmierers angesehen werden kann, der diese Funktion entwickelt hat. Enthält z. B. ein Programm eine Funktion zur Berechnung der Länge eines 3-dim. Vektors, so genügt es, den Funktionskopf zu kennen z. B.

double VektorLaenge (**double** x, **double** y, **double** z)

Die Anwendung erfolgt durch einen Funktionsaufruf, indem der Funktionsname mit aktuellen, zum Typ der formalen Parameter passenden aktuellen Parametern angegeben wird. Der Funktionsaufruf

VektorLaenge (1.5, 2.5, -3.1);

ist ein einfacher Ausdruck mit dem Wert 4.255584566.

Beachte: In C sind auch **rekursive** Funktionsaufrufe erlaubt!

Für die gebräuchlichsten Funktionen stehen in C Bibliotheksfunktionen bereit, die mittels der **#include**-Direktive in jedes Programm aufgenommen werden können. Dazu dienen die sogenannten Header-Dateien, in denen die Funktionen thematisch angeordnet sind.

Zum Beispiel stammen `getchar` und `printf` aus der Headerdatei `stdio.h`, `clrscr` aus `conio.h`.

Die Direktive `#include <math.h>` erlaubt die Verwendung folgender Funktionen vom Funktionstyp `double`. Die mit `x`, `y` bezeichneten Argumente sind Ausdrücke vom Typ `double`.

<code>acos(x)</code>	Arcuscosinus	<code>fmod(x,y)</code>	Rest einer Division x/y
<code>asin(x)</code>	Arcussinus	<code>log(x)</code>	natürlicher Logarithmus
<code>atan(x)</code>	Arcustangens	<code>log10(x)</code>	dekadischer Logarithmus
<code>atan2(x,y)</code>	Arcustangens x/y	<code>modf(x,y)</code>	ganzzahliger und gebrochener Teil
<code>ceil(x)</code>	Aufrunden	<code>pow(x,y)</code>	potenzieren x^y
<code>cos(x)</code>	Cosinus	<code>sin(x)</code>	Sinus
<code>cosh(x)</code>	Cosinus hyperbolicus	<code>sinh(x)</code>	Sinus hyperbolicus
<code>exp(x)</code>	Exponentialfunktion	<code>sqrt(x)</code>	Quadratwurzel
<code>fabs(x)</code>	Absolutwert	<code>tan(x)</code>	Tangens
<code>floor(x)</code>	Abrunden	<code>tanh(x)</code>	Tangens hyperbolicus

2. 4. Speicherklassen

Die Speicherklasse eines Objektes (z. B. Konstante, Variable, Funktion) bestimmt, von welchen Programmteilen zu welcher Zeit auf das Objekt zugegriffen werden kann. Jedes Objekt besitzt eine Attribute für folgende vier Eigenschaften:

1. Gültigkeit
2. Sichtbarkeit
3. Bindung
4. Lebensdauer

1. Der Gültigkeitsbereich eines Objektes ist der Programmteil, in dem darauf zugegriffen werden kann. Wurde das Objekt innerhalb eines Blockes deklariert, so erstreckt sich ihr Gültigkeitsbereich von der Deklarationsstelle bis zur schließenden, geschweiften Klammer am Ende dieses Blockes. Sie gilt dann als **lokale** Variable dieses Blockes. Eine in einem Block deklarierte Variable ist auch in all dessen Unterblöcken definiert. Für solche Blöcke gilt sie als **globale** Variable.

Variablen können auch außerhalb aller Blöcke vor, zwischen und hinter den Funktionsdeklariert werden. Dann erstreckt sich ihr Gültigkeitsbereich von der Deklarationsstelle bis zum Dateiende. Sie gelten dann in allen Blöcken zwischen Deklarationsstelle und Dateiende als **global**.

Beispiel:

```
void Fkt1 ( float a, float b, float c ) { // Parameter gelten als lokale Größen
```

```
    float x, y;
    // in diesem Block sind a, b, c und x, y lokale Größen
    ...
}
```

```
int i, j, k; // i, j, k sind von hier bis zum Programmende gültig
```

```
void Fkt2 ( int a, int b ) {
    float y, z;
    // in diesem Block sind a, b, y, z lokale, i, j, k globale Größen
    // a, y aus Fkt2 haben nichts mit a, y aus Fkt1 zu tun!
}
```

```
void main ( ) {
    float e, pi;
    // in diesem Block sind e, pi lokal, i, j, k global
    ...
    {
        char ch; // in diesem Block sind ch lokal, e, pi, i, j, k global
        ...
    }
}
```

2. Der Sichtbarkeitsbereich eines Objektes stimmt normalerweise mit seinem Gültigkeitsbereich überein. Die Ausnahme gilt dann, wenn eine Namensgleichheit zwischen einer globalen und einer lokalen Variablen besteht. In diesem Fall wird die globale Variable durch die lokale in deren Block „verdeckt“. Wird z. B. in o. a. Beispiel in Fkt eine weitere Variable `i` deklariert, so repräsentiert diese einen eigenen Speicherplatz und verhält sich so, als besäße sie einen anderen Namen.

Beispiel:

```

int i, j, k;

void Fkt ( float a, float b, float c ) {

    float i, x, y, z; // das globale i ist in Fkt unbekannt
    ...
}

void main ( ) {

    float eps, pi;
    // ab hier gilt eps
    ...

    {
        // hier beginnt ein neuer Block
        char eps, ch; // in diesem Block gilt ein eigenes, lokales eps
        ...
    }
    // ab hier gilt wieder das globale eps
    ...
}

```

Beispiel:

```

#include <stdio.h>
#include <conio.h>

void main ( ) {

    int a = 10;           // globales a
    clrscr ( );
    printf ("%d", a++);  // Ausgabe: 10, dann wird a = 11
    {
        int a = 20;      // lokales a
        printf ("%d", a++); // Ausgabe: 20, dann wird a = 21
    }
    printf ("%d", a );    // Ausgabe: 11
    getch ( );
}

```

3. Der Vorgang der Bindung gewinnt dann Bedeutung, wenn ein Programm aus verschiedenen Dateien besteht, die unabhängig voneinander übersetzt und mit vorhandenen Bibliotheksdateien zu einem ausführbaren Programm verbunden (gelinkt) werden.

Besteht ein Programm aus mehr als einer Datei, so besitzt eine Größe

- externe Bindung, wenn sie in allen Programmdateien das gleiche Objekt bezeichnet,
- interne Bindung, wenn sie nur in einer Datei das gleiche Objekt bezeichnet,
- sonst keine Bindung.

3. 1. Eine in einer Datei nicht innerhalb einer Funktion definierte Variable oder Funktion besitzt externe Bindung. Das bedeutet, daß sie nicht nur in der gleichen Datei, sondern auch in jeder anderen eingebundenen Datei unter dem gleichen Bezeichner benutzt werden kann, wenn sie dort als Prototyp deklariert wird.

Zur Deklaration eines Prototyps gibt man den mit dem Vorsatz extern versehenen Funktionskopf an, schreibt in die Parameterliste aber lediglich die Datentypen der formalen Parameter.

Beispiel:

Eine Datei importiert aus einer zweiten die Variable pi und die Funktion Fkt1.

1. Datei

extern double pi;

extern double Fkt1 (**float**);

extern double Fkt2 (**float**, **int**);

// die Zahl pi und die Funktionen

// Fkt1 und Fkt2 besitzen externe Bindung

void main () {

...
}

2. Datei

double pi = 3.1415926535897932384626;

double Fkt1 (**float** z) {

...

int Fkt2 (**float** x, **int** i) {

...
}

Beachte:

Variablen mit externer Bindung dürfen in ihrer Exportdatei mit einer Konstanten initialisiert werden.

3. 2. Eine mit dem Vorwort **static** definierte globale (lokale) Variable oder Funktion besitzt interne Bindung. Sie ist nur innerhalb ihrer Datei (ihres Blockes) zugänglich und steht deshalb nicht in Konflikt zu einem in einer anderen Datei auftretenden gleichen Namen.

(Kerningham & Ritchie bezeichnen globale static-Variable als externe und lokale static-Variable als interne static-Variable)

3. 3. Funktionsparameter und lokale Größen besitzen das Attribut: keine Bindung.

Beachte:

Externe und interne Größen belegen einen während des gesamten Programmlaufes permanenten Speicherbereich.

4. Unter der Lebensdauer eines Objektes versteht man den Zeitraum während des Programmablaufes, in dem dieses einen Speicherplatz besitzt. Man unterscheidet hinsichtlich der Lebensdauer

- statische,
- automatische und
- dynamische Objekte.

4. 1. Statische Objekte behalten während des gesamten Programmlaufes ihren Speicherplatz. Dazu gehören alle Funktionen und Variablen mit externer oder interner Bindung.

Sie werden entweder explizit mit einer Konstanten oder automatisch mit Null initialisiert.

Eine innerhalb eines Funktionsblockes definierte, lokale **static**-Variable behält ihren Wert auch, wenn der Programmlauf die Funktion verläßt, so daß sie beim Wiederaufruf der Funktion weiter verwendet werden kann.

Beispiel:

Eine Funktion mit dem Kopf **void** Summe () soll bei jedem Aufruf einen Zähler um 1 erhöhen und ausgeben.

```

void Summe ( ) {

    static Zaehler;

    Zaehler++;
    printf ("%d \n", Zaehler);
}

void main ( ) {

    Summe ();
    Summe ();
    Summe ();
    Summe ();
    ...
}

```

4. 2. Die bisher innerhalb eines Blockes ohne den Zusatz extern oder static deklarierten Variablen gehören zu den automatischen Variablen.

Die Werte automatischer Variablen werden nicht in einen permanenten Speicherbereich sondern nur temporär zu der Zeit gespeichert, während dessen der Block, in dem sie deklariert wurden, vom Programm abgearbeitet wird.

Bei erneutem Eintritt des Programmlaufes in den Block wird zwar der Speicherplatz erneut eingerichtet, jedoch sind die alten Variablenwerte gelöscht.

Beachte:

Im Gegensatz zu **static**-Variablen, die nur mit einer Konstanten initialisiert werden dürfen, können automatische Variablen mit konstanten Ausdrücken definierter Größen initialisiert werden.

Während nicht explizit initialisierte **static**-Variablen automatisch mit 0 initialisiert werden, sind nicht explizit initialisierte, automatische Variablen **undefiniert**!

Man unterscheidet bei automatische Variablen noch die Speicherklassen **automatic** und **register**. Zu den ersten gehören die mit dem optativen Zusatz **auto** deklarierten, zu den anderen die mit dem Zusatz **register** deklarierten Variablen. Letztere haben die Eigenschaft, daß bei häufigem Zugriff dem Compiler empfohlen wird, schnelle Register als Speicher zu benutzen.

Beispiel:

```
void Fkt ( int a ) {
    int b = a + 1;
    ...
}

void main ( ) {
    int i = 10;
    int j = i + 1;
    Fkt ( j );
    ...
}
```

4. 3. Objekten mit dynamischer Lebensdauer wird während des Programmlaufes durch besondere Funktionsaufrufe wie **malloc** (in C++ auch **new**) temporärer Speicherplatz zugewiesen. Mit **free** (in C++ auch **delete**) kann der Speicher wieder freigegeben werden.

Beispiel:

Mit `int *p;`
`double *q;`
 werden Zeigervariablen p und q deklariert; sie repräsentieren (noch undefinierte) Speicheradressen einer int-Variablen *p und einer double-Variablen *q.

Durch die Zuweisungen

```
p = ( int * ) malloc ( sizeof ( int ) );
q = ( double * ) malloc ( sizeof ( double ) );
```

werden die Adressen definiert und den Zeigervariablen zugeordnet.

Mit `free ( p );`
`free ( q );`

wird der Speicher wieder freigegeben, so daß die Adressen anderen Objekte zur Verfügung stehen.

Übung 2_1: Formulieren Sie in eine Funktion **void** Rahmen (), die einen programmbegleitenden Kopftext nach folgendem Muster eingerahmt ausgibt.

Benutzen Sie eine Folge von printf-Befehlen, so daß nach einem Signalton folgende Zeilen, denen ein Tabulatorsprung vorangehen soll, ausgegeben werden:

<Aufgabenstellung>
<Vorname> <Nachname>
<Übungsgruppe>
<Datum der letzten Änderung>

Speichern Sie das Programm unter dem Namen Rahmen.cpp auf Ihre Diskette und ergänzen Sie die folgenden Übungsaufgaben um diese Funktion mit passenden Textstellen.

Rufen Sie vor der ersten Ausgabeanweisung in main die Funktion Rahmen auf.

Übung 2_2: Ergänzen Sie Übung 2_1 um #define- und printf-Befehle, so daß sie nach Ausgabe des o. a. Kopfes eine Tabelle physikalischer Konstanten in folgender Form ausgibt:

Physikalische Konstanten in SI-Einheiten		
Vakuumlichtgeschwindigkeit	c	$2,99792458 \cdot 10^8$
Gravitationskonstante	G	$6,67259 \cdot 10^{-11}$
Elementarladung	e	$1,60217733 \cdot 10^{-19}$
Plancksche Konstante	h	$6,620755 \cdot 10^{-34}$
Avogadro-Konstante	NA	$6,0221367 \cdot 10^{23}$
Elektronenmasse	me	$9,1093897 \cdot 10^{-31}$
Atomare Masseneinheit	u	$1,6605402 \cdot 10^{-27}$
Protonenmasse	mp	$1,6726231 \cdot 10^{-27}$
Neutronenmasse	mn	$1,6749286 \cdot 10^{-27}$
Magn. Moment des Elektrons	μe	$9,2847701 \cdot 10^{-24}$
Magn. Moment des Protons	μp	$1,41060761 \cdot 10^{-26}$
Quanten-Hallwiderstand	RH	$2,58128056 \cdot 10^4$
Molare Gaskonstante	R	8,314510
Magnetische Feldkonstante	μ0	$1,25663706 \cdot 10^{-6}$

3. Ausdrücke und Operatoren

3. 1. Ausdrücke

Ein Ausdruck ist eine mathematische Formel, in der Operanden <operands> durch Operatoren <operators> verknüpft werden, so daß aus den Werten der Operanden ein neuer, einzelner typisierter Wert erzeugt wird. Die Operanden sind selbst Ausdrücke, deren Werte sich bei der Auswertung des Gesamtausdrucks ändern können.

Zu den elementaren Ausdrücken gehören Konstante und Namen von Variablen und Funktionen.

Beispiel: $(5 + x / 3 + \sin(x)) / i++$

Hier werden die elementaren Ausdrücke: Konstanten 5, 3, Variablen x, i, Funktion sinus durch die arithmetischen Operatoren +, -, /, ++ zu neuen Ausdrücken und diese weiter bis zum Gesamtausdruck verknüpft. Gleichzeitig ändert sich der Wert der Variablen i.

Ein Ausdruck kann

- auf der rechten Seite einer Gleichung stehen z. B. $y = (5 + x / 3 + \sin(x)) / i++$, um seinen Wert einer Variablen zuzuweisen,
- als aktueller Parameter in einem Funktionsaufruf auftreten
z. B. $\cos((5 + x / 3 + \sin(x)) / i++)$
 $\text{printf}("%8.5f \backslash n", (5 + x / 3 + \sin(x)) / i++)$

Die Zuweisungsoperation <Variable> = <Ausdruck>; ist ebenfalls ein Ausdruck mit dem Wert der rechten Seite.

Für die eindeutige Bewertung eines Ausdrucks ist die Reihenfolge, in der die Teilausdrücke durch die Operatoren zu neuen Ausdrücken verknüpft werden, von entscheidender Bedeutung.

Diese kann durch Verwendung runder Klammernpaare gesteuert werden, andererseits besitzen die Operatoren unterschiedliche Bindungsstärken und Auswertungsrichtungen, durch die eine bestimmte Reihenfolge der Auswertung automatisch festgelegt wird.

Beispiele:

$(a * b) + c$ ist dasselbe wie $a * b + c$, weil *Punkt- vor Strichrechnung* gilt,

$a + b + c$ ist dasselbe wie $(a + b) + c$, weil hier die Auswertung von links nach rechts erfolgt.

Beachte: Ist der Ausdruck $a + b + c$ vom Gleitkommatyp, so muß aufgrund des numerischen Fehlers nicht $(a + b) + c = a + (b + c)$ sein.

Mehrdeutigkeiten:

In dem Ausdruck $j = i + i++$; ist nicht klar, welchen Wert j im Falle $i = 1$ annimmt.

j kann 2, 3 oder 4 sein, je nachdem ob die ++-Operation vor oder nach der Addition, vor oder nach der Wertzuweisung an j ausgeführt wird.

Testen Sie diesen Ausdruck, außerdem $j = i++ + i$; mit Hilfe eines kleinen Programmes.

3. 2. Operatoren

Die Operatoren der Programmiersprache C lassen sich in folgende Gruppen einteilen:

- Arithmetische Operatoren <arithmetics operators>
- Vergleichsoperatoren <relational operators>
- Logische Operatoren <logical operators>
- Operatoren zur Bitmanipulation <bitwise and shift operators>
- sizeof-Operator <sizeof operator>
- Zuweisungsoperatoren <assignment operators>
- Inkrement- und Dekrementoperatoren <unary increment and decrement operators>
- Kommaoperator <sequential evaluation operator>
- cast-Operator <cast operator>
- Auswahl- und Indexoperator
- Verweis- und Adreßoperator

Ein Operator mit nur einem Operanden heißt **unitär**, z. B. das Minuszeichen -<operand>, mit zwei Operanden **binär**, z. B. das Pluszeichen <operand1> + <operand2>, mit drei Operanden **ternär**, z. B. die bedingte Bewertung <operand1> ? <operand2> : <operand3>.

3. 2. 1. Arithmetische Operatoren

C kennt folgende arithmetischen Operatoren:

Operator	Bedeutung	Operandentyp
-	Vorzeichen für negative Zahlen	Integer- oder Gleitkommatyp
+	Addition	"
-	Subtraktion	"
*	Multiplikation	"
/	Division	"
%	Divisionsrest	Integertyp

Bemerkungen und Beispiele:

- Das Minuszeichen darf nie direkt einem Operator folgen. Ggf. muß es geklammert werden. Z. B. $2 + (-3)$ und nicht $2 + -3$.
- Ein unitärer Plusoperator + ist normalerweise in C unbekannt und auch unnötig.
- Der Multiplikationsoperator * darf nicht, wie in der Mathematik üblich, weggelassen werden. Z. B. ist $(5 + 3) 4$ falsch.
- Das Minuszeichen ergibt für einen unsigned-Operanden wieder einen unsigned-Wert, indem in diesem Falle das Einerkomplement gebildet wird. Das bedeutet bei 2 Bytes (=16 Bits) die Addition von 65535.

Beispiele:

beispiel	-beispiel
27329	38206 (= 65535 - 27329)
0	65535 (= 65535 - 0)
1	65534 (= 65535 - 1)
65535	0 (= 65535 - 65535)

- Bereichsüberschreitungen führen bei Integertypen zu falschen Ergebnissen (modulo-Rechnung) und bei Gleitkommatypen zum Programmabbruch mit Fehlermeldung (arithmetic overflow).
- Sind bei der Division mit / beide Operanden vom Ganzzahltyp, so wird auch das Ergebnis zu einem Ganzzahltyp abgerundet. Ist ein Operand negativ, so wird rechnerabhängig auf- oder abgerundet.

Beispiele:

21 / 5 ergibt 4
 17 / 9 " 1
 12 / 2 " 6
 5 / 7 " 0
 -16 / 5 " -3 oder -4

- Der Operator % darf nur auf Operanden eines Integertyps angewandt werden. Er ergibt den bei der Division entstehenden ganzzahligen Rest.

Beispiele:

8 % 2 ergibt 0
 15 % 6 " 3

- Bei der Auswertung eines Ausdrucks werden vorrangig

Klammerausdrücke ausgewertet (höchste Prioritätsstufe),
 dann Minuszeichen gebildet,
 dann multiplikative Operationen *, /, %,
 dann additive Operationen +, - ausgeführt. (Regel: *Punkt- vor Strichrechnung*)

Bei Operatoren gleicher Priorität gilt bis auf das Minuszeichen die Reihenfolge von links nach rechts.

Beispiele:

3 * 2 % 4 $\Leftrightarrow$ 6 % 4 $\Leftrightarrow$ 2
 (5 + 3)*4 $\Leftrightarrow$ 8 * 4 $\Leftrightarrow$ 32
 (3 + 2.5) * (6 - 2) - 2 $\Leftrightarrow$ 5.5 * 4 - 2 $\Leftrightarrow$ 22 - 2 $\Leftrightarrow$ 20
 6.0 / 1.5 - 1.1 * (14 - 12) $\Leftrightarrow$ 6.0 / 1.5 - 1.1 * 2 $\Leftrightarrow$ 4.0 - 2.2 $\Leftrightarrow$ 1.8
 3.2 + 8.7 - 2.1 * 7.25 $\Leftrightarrow$ 3.2 + 8.7 - 15.225 $\Leftrightarrow$ 11.9 - 15.225 $\Leftrightarrow$ 3.325

3. 2. 2. Vergleichsoperatoren

C stellt folgende binären Vergleichsoperatoren zur Verfügung:

Operator	Bedeutung	Operandentyp
<	kleiner als	Integer- oder Gleitkommatyp
<=	kleiner als oder gleich	"
>	größer als	"
>=	größer als oder gleich	"
==	gleich	"
!=	ungleich	"

Bemerkungen und Beispiele:

- Die Operanden sind Ausdrücke, die miteinander verglichen werden. Ist die Aussage WAHR, dann lautet das Ergebnis 1, ist sie FALSCH, so wird es 0.

Beispiele:

3.15 <= 9.07	ergibt	1
-2.97 != 2.97	"	1
0.5 == 1.2	"	0
12 < 9	"	0
a <= h	hängt vom Wert der Variablen a und h ab	
i = 10	hängt vom Wert der Variablen i ab	

- Es gibt unter den Vergleichsoperatoren zwei Prioritätsgruppen. Die Gruppe <, <=, >, >= hat eine geringere Bindungsstärke als die arithmetischen Operatoren und eine höhere als die andere Gruppe ==, !=.
- Innerhalb einer Gruppe gilt auch hier die Auswertungsreihenfolge von links nach rechts.

Beispiel:

$5 + 3 >= 5 - 3 == 5 * 3 < 5 / 3 \Leftrightarrow 8 >= 2 == 6 < 1 \Leftrightarrow 1 == 0 \Leftrightarrow 0$ ist also FALSCH

3. 2. 3. Logische Operatoren

Logische Operatoren verknüpfen den Wahrheitswert der Operanden zu einem neuen Wahrheitswert. Während andere Programmiersprachen für Boolesche Größen einen eigenen Datentyp z. B. Boolean mit den Werten TRUE und FALSE kennen, benutzt C einen Integertyp. Ein Operand gilt genau dann als FALSCH, wenn er den Wert 0 hat. Ist er ungleich 0, so wird er als WAHR angesehen. In C gibt es die elementaren booleschen Verknüpfungen:

Operator	Bedeutung	Operandentyp
!	Negation	Integertyp
&&	logische UND-Verknüpfung	"
	logische ODER-Verknüpfung	"

mit folgenden Werten:

Op1	Op2	!Op1	Op1 && Op2	Op1 Op2
0	0	1	0	0
0	≠0	1	0	1
≠0	0	0	0	1
≠0	≠0	0	1	1

Bemerkungen:

- Der Negationsoperator hat die gleiche Priorität wie das Minuszeichen.
- Das logische UND && hat geringere Priorität als ein Vergleichsoperator und höhere als das logische ODER ||.

Beispiele:

1.) Die vier logische Variablen: Ziffer, Kleinbuchstabe, Grossbuchstabe und Sonderzeichen sind genau dann 1, wenn die Variable Zeichen vom Typ char einen dem Namen entsprechenden Wert hat.

```
Ziffer = Zeichen >= '0' && Zeichen <= '9';
Kleinbuchstabe = Zeichen >= 'a' && Zeichen <= 'z';
Grossbuchstabe = Zeichen >= 'A' && Zeichen <= 'Z';
Sonderzeichen = !( Ziffer || Kleinbuchstabe || Grossbuchstabe );
```

2.) Folgende logische Variablen testen, ob ein mit den int-Variablen Tag, Monat, Jahr vorgegebener Datumswert ab dem Jahre 1 zum Julianischen- oder Gregorianischen Kalender gehört. Ferner wird die Gültigkeit der Zahlenangaben festgestellt.

```
Julian = Jahr > 0 && Jahr < 1582;
Julian = Julian || (Jahr == 1582 && ( Monat < 10 || Monat == 10 && Tag < 5 ));
SchaltjahrJ = !( Jahr % 4 );
```

```
Gregor = Jahr > 1582;
Gregor = Gregor || (Jahr == 1582 && ( Monat > 10 || Monat == 10 && Tag > 14 ));
SchaltjahrG = SchaltjahrJ && Jahr % 100 || !( Jahr % 400 );
```

```
Schaltjahr = Julian && SchaltjahrJ || Gregor && SchaltjahrG;
```

```
Monatkorrekt = Monat > 0 && Monat < 13;
```

```
Tagkorrekt = ( Monat == 2 ) && ( Tag < 29 + Schaltjahr ) ||
              ( Monat != 2 ) && ( Tag < 31 + ( Monat % 2 != Monat / 8 ));
```

```
Tagkorrekt = Tagkorrekt && Tag > 0;
```

3. 2. 4. Bitmanipulation

Die Operatoren zur Bitmanipulation gehören zu den Sprachelementen, mit deren Hilfe man maschinennah programmieren kann, so daß sich C besonders gut für die Systemprogrammierung eignet.

Operator	Bedeutung	Operandentyp
~	Bitkomplement	Integertyp
&	UND-Verknüpfung von Bitfolgen	"
	ODER-Verknüpfung von Bitfolgen	"
^	Exklusiv-ODER-Verknüpfung von Bitfolgen	"
<<	Linksverschiebung von Bitfolgen	"
>>	Rechtsverschiebung von Bitfolgen	"

Bemerkungen und Beispiele:

- Der unitäre Operator ~ komplementiert alle Bits des Operanden.
- Es gilt $\text{muster} + \sim\text{muster} = -1$, also $\sim\text{muster} = 1 - \text{muster}$.

Beispiel:

Die Variable namens muster sei vom Datentyp int und belege 2 Bytes (= 16 Bits) mit dem Wert $0246 = 166 = 0xA6$. Der Rechner legt diese Zahl in seinem Speicher folgendermaßen ab:

muster:

0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Bildet man das Komplement, so entsteht:

~muster:

1	1	1	1	1	1	1	1	1	0	1	0	1	1	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

mit dem Zahlenwert $0177531 = -167 = 0xFF59$

- Die drei binären Operatoren &, |, ^ verknüpfen jedes einzelne Bit des ersten Operanden mit dem an gleicher Stelle stehenden Bit des zweiten Operanden. Das Resultatbit nimmt dann die gleiche Stelle im Ergebnis ein.

Beispiele:

Im folgenden wird der 1. Operand 01736 mit dem 2. Operanden muster elementar verknüpft:

01736:

0	0	0	0	0	0	0	1	1	1	1	0	1	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

 &
 muster:

0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

 =

Ergebnis:

0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

 01736:

0	0	0	0	0	0	1	1	1	1	0	1	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

 ||
 muster:

0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

 =
 Ergebnis:

0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

01736:

0	0	0	0	0	0	1	1	1	1	0	1	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

 ^
 muster:

0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

 =
 Ergebnis:

0	0	0	0	0	0	1	1	0	1	1	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- Der binäre Operator >> (<<) verschiebt den Inhalt eines Speicherbereiches, also einer Bitfolge, um eine gewisse Anzahl von Bitpositionen nach rechts (links). Der erste Operand repräsentiert das zu verschiebende Bitmuster, der zweite Operand gibt die Anzahl der Bitpositionen an, um die verschoben werden soll.

Bei einer Linksverschiebung werden von rechts immer Nullen nachgezogen. Die Bits, die auf der linken Seite aus dem Speicherbereich hinausgeschoben werden, gehen dabei verloren. Dies entspricht einer Multiplikation mit 2^n modulo 32768.

Wird ein Bitmuster, das als unsigned interpretiert wird, nach rechts verschoben, werden von links immer Nullen nachgezogen. Sonst werden rechnerabhängig Nullen (logisches Schieben) oder das Vorzeichenbit (arithmetisches Schieben) nachgezogen. Der letzte Fall entspricht einer Ganzzahldivision durch 2^n .

Beispiele:

muster:

0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

 << 4

0	0	0	0	1	0	1	0	0	1	1	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

muster = 0246, muster << 4 = 05140 = 2656 = 0xA60

muster:

0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

 >> 4

0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

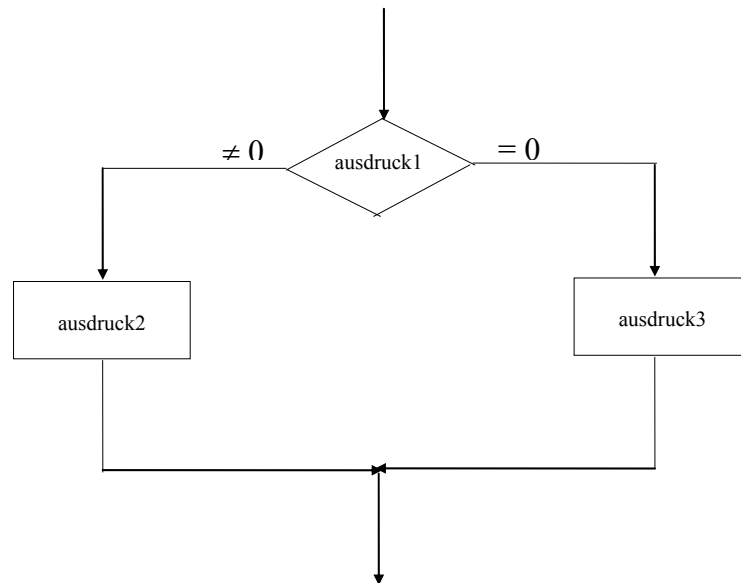
muster = 0246, muster >> 4 = 012 = 10 = 0xA

3. 2. 5. Bedingte Bewertung

Ein Ausdruck mit der Syntax `<ausdruck1> ? <ausdruck2> : <ausdruck3>` heißt bedingte Bewertung. Zunächst wird der Integerausdruck `<ausdruck1>` ausgewertet. Ist er WAHR, also $\neq 0$, so wird anschließend `<ausdruck2>`, andernfalls `<ausdruck3>` ausgewertet. Das jeweilige Ergebnis ist der Wert der bedingten Bewertung.

Der ternäre Operator besitzt die geringste Bindungsstärke, die Auswertung erfolgt von rechts nach links.

Im Syntaxdiagramm läßt sich der Sachverhalt folgendermaßen darstellen:



Beispiele:

`Betrag = ( Zahl > 0 ) ? Zahl : -Zahl;`

`Maximum = ( Zahl1 > Zahl2 ) ? Zahl1 : Zahl2;`

`Klausur = ( Note >= 4 ) ? "Bestanden" : "Durchgefallen";`

3. 2. 6. sizeof-Operator

Der unitäre `sizeof`-Operator ist eine Funktion eines Objektes, welche die Anzahl der von dem Objekt belegten Bytes liefert. Objekt kann eine Variable, ein Vektor, elementarer Datentyp oder Strukturtyp außer `void` sein. Er hat die gleiche Priorität wie `-` und `!`.

Beispiele:

`bytes_von_x = sizeof ( x );`

`int_laenge = sizeof ( int );`

3. 2. 7. Zuweisungsoperatoren

Jede Variable repräsentiert einen typabhängigen Speicherbereich, der bei der Deklaration entweder einen undefinierten Zufallswert erhält oder automatisch mit Null initialisiert wird. Mit Hilfe einer Zuweisung, z. B. der einfachen Zuweisung

$$\langle \text{variable} \rangle = \langle \text{ausdruck} \rangle$$

wird der Wert des Ausdrucks auf der rechten Seite in den Speicherbereich der Variablen geschrieben. Ausdrücke, die auf der linken Seite einer Gleichungsform stehen dürfen, z. B. Variablen, heißen L-Werte $\langle \text{lvalue} \rangle$, die nur auf der rechten Seite stehen dürfen, z. B. $a + b$, R-Werte.

In C bildet eine Zuweisung wieder einen Ausdruck mit dem zugewiesenen Wert, so dass auch Mehrfachwertzuweisungen möglich sind:

$$\langle \text{variable1} \rangle = \langle \text{variable2} \rangle = \langle \text{ausdruck} \rangle;$$

der Wert des Ausdrucks zwei Variablen zugewiesen.

Beispiele:

```
int Note, Punkte;
char Buchstabe, Trennzeichen;
double x, y;
...
Note = 5 - Punkte*0.04;
Buchstabe = 'C';
Trennzeichen = '\n';
x = 2.5E1;
y = 3.1 * x * y - 0.9 * (11 + x) + x / 20;
```

In vielen Fällen taucht der L-Wert auch in der rechten Seite der Zuweisung auf z. B. $i = i + 1$; wenn der Wert einer Variablen geändert wird. Für derartige Zuweisungen stellt C eine Anzahl von Zuweisungsoperatoren zur Verfügung, die eine abkürzende Schreibweise ermöglichen. Statt

$$\langle \text{variable} \rangle = \langle \text{variable} \rangle \text{ op } \langle \text{ausdruck} \rangle;$$

wobei op für einen der Operatoren: +, -, *, /, %, <<, >>, &, |, ^ steht, kann man schreiben:

$$\langle \text{variable} \rangle \text{ op } = \langle \text{ausdruck} \rangle;$$

Beispiele:

Betrag += 25;	ist äquivalent	Betrag = Betrag + 25;
Betrag -= 3;	"	Betrag = Betrag - 3;
Betrag *= i - 11;	"	Betrag = Betrag * (i - 11); <u>nicht</u> Betrag * i - 11;
Betrag %= i;	"	Betrag = Betrag % i;
Muster >>= 4;	"	Muster = Muster >> 4;

Die Bindungsstärke der Zuweisungsoperatoren ist kleiner als die der bedingten Bewertung, die Auswertung erfolgt von rechts nach links.

2. 8. Inkrement- und Dekrementoperatoren

Inkrementieren (Dekrementieren) bedeutet, einen Wert um 1 zu erhöhen (erniedrigen). C stellt zur Inkrementierung eines L-Wertes den unitären Operator ++, zur Dekrementierung - - zur Verfügung. Er steht unmittelbar vor (Präfixnotation) oder hinter (Postfixnotation) dem L-Wert.

Bei Präfixnotation wird die Variable zuerst in- bzw. dekrementiert und dann der Ausdruck, in dem sie auftritt, ausgewertet, bei Postfixnotation wird sie erst nach Auswertung des gesamten rechts stehenden Ausdrucks in- bzw. dekrementiert.

Beispiele:

```
Zahl = 10;
Resultat = ++Zahl; // Zahl = Zahl + 1;
                // Resultat = Zahl;
                // dann gilt Zahl = Resultat = 11
```

```
Zahl = 10;
Resultat = Zahl++; // Resultat = Zahl;
                // Zahl = Zahl + 1;
                // dann gilt Zahl = 11, Resultat = 10
```

```
a = 2;
b = 5;
```

Postfixnotation:
Resultat = a * b--;

```
a = 2
b = 4
Resultat = 10
```

Danach gilt

Präfixnotation:
Resultat = a * --b;

```
a = 2
b = 4
Resultat = 8
```

3. 2. 9. Kommaoperator

Zwei durch ein Komma getrennte Ausdrücke bilden syntaktisch einen neuen Ausdruck.

<ausdruck1> , <ausdruck2>;

Zuerst wird der Ausdruck vor, dann der nach dem Komma ausgewertet. Letzterer liefert den Wert des neuen Ausdrucks.

Beispiel:

```
k = i = 1, i = 2;
```

- Der Kommaoperator besitzt die geringste Bindungsstärke, die Auswertung erfolgt von links nach rechts.

3. 2. 10 Typkonvertierung

Enthält ein Ausdruck Komponenten mit unterschiedlichen Datentypen, so werden die Werte aller beteiligten Operanden automatisch, also implizit, in einen einheitlichen Datentyp umgewandelt.

Haben die bei einer arithmetischen Operation mit binärem Operator beteiligten Operanden beide unterschiedliche Datentypen der gleichen Gruppe (Ganzzahl- oder Gleitkommatyp), dann wird der Typ, der den geringeren Speicherplatz beansprucht in den anderen konvertiert.

1. **char** und **short** werden nach **int**, **float** nach **double** umgewandelt.
2. Ist ein Typ **long**, so wird auch der andere in **long**, ist ein Typ **unsigned**, so wird auch der andere in **unsigned** umgewandelt.

Im Falle einer Zuweisung erhält das Ergebnis den Datentyp des L-Wertes, so daß ein Informationsverlust auftreten kann, falls dieser Typ weniger Bits belegt.

In C kann eine Typumwandlung auch durch den Programmierer, also explizit, mit Hilfe des sogenannten **cast**-Operators erfolgen. Soll ein Ausdruck <ausdruck> in einen Datentyp <typ> umgewandelt werden, so schreibt man: (<typ>) <ausdruck>

Beispiele:

```
float x;
double y;
int Wert;
char i;
y = sin ((double)x); // sinus erwartet den Typ double
i = (char)Wert;      // hier werden führende Bits gestrichen
```

3. 2. 11. Prioritätstabelle

Symbole	Assoziation
() [] .	von links nach rechts
! ~ - ++ -- & * (typ) sizeof	von rechts nach links
* / %	von links nach rechts
+ -	"
>> <<	"
> >= < <=	"
!= ==	"
&	"
^	"
	"
&&	"
	"
? :	von rechts nach links
= += -= *= /= %= <<= >>= &= = ^=	"
,	links nach rechts

Übung 3_1: Korrigieren Sie das folgende Programm und lassen Sie es ablaufen.

```

#include <stdio.h>
#include <conio.h>

void main ( )    {

    int  a = 20;
        b = 7;
        c = 2;
    float f = 20.0;

    printf ("%d %d %d %d %d %d \n", a/b, a%b, b/a, b%a, f/7.0, 7.0/20);
    printf ("%d %d %d %d %d \n", ++c + c, c++, ++c, ++c + c++, c);
    getchar ( );
}

```

Übung 3_2: Streichen Sie aus folgenden Anweisungen diejenigen, die nicht erlaubt sind?

```

y = x + y - 2;
y - 2 = x + 3;
++y = x++;
y = x - 3 * 2;
y = (x - 3) * 2;
y = (x - 3)++;
y = x++ % y;
y = ++x % y;
y = ++x++;
y = 10, y%=2, y+=5, y-=3, y*=3, y/=6, y+=1 ;
y = (z = 7, ++z, z = 72 / z, ++z);

```

Welche Werte haben die int-Variablen x, y und z nach Ausführung einer gültigen Anweisung, wenn sie anfangs die Werte x = 10, y = 5 und z = 0 hatten?

Geben Sie nach jeder Anweisung die Werte von x und y aus!

Was passiert, wenn Sie die letzte Zuweisung durch

`printf ("%d %d %d %d", z = 7, ++z, z = 72 / z, ++z );` ersetzen?

Übung 3_3: Berechnen sie mit Bleistift und Papier folgende Ausdrücke. Kontrollieren Sie anschließend Ihre Ergebnisse mit Hilfe eines Programms.

14 && 7	14 7	!14	!7
14 & 7	14 7	14 ^ 7	~14
~(14 & 7)	~14 & ~7	~14 ~7	~~7
13 << 4	13 >> 2	(13 >> 4) << 4	(13 << 4) >> 4

Übung 3_4: Ergänzen Sie Übung 2_2 um Ausdrücke, mit denen weitere physikalische Konstanten berechnet werden, und zwar folgende:

Elektrische Feldkonstante	$\epsilon_0 = (\mu_0 * c^2)^{-1}$
Faradaykonstante	$F = N_A * e$
Feinstrukturkonstante	$\alpha = e^2 (2 * \epsilon_0 * h * c)^{-1}$
Rydberg-Konstante	$R_y = (2h)^{-1} m_e * c * \alpha^2$
Elektronenradius	$r_e = h * \alpha (2\pi * m_e * c)^{-1}$
Comptonwellenlänge	$\lambda_c = h (m_e * c)^{-1}$
Bohrscher Radius	$r_0 = r_e * \alpha^{-2}$
Bohrsches Magneton	$\mu_B = e * h (4\pi * m_e)^{-1}$
Kern-Magneton	$\mu_N = e * h (4\pi * m_p)^{-1}$
Boltzmann-Konstante	$k = R * N_A^{-1}$
Stefan-Boltzmann-Konstante	$\sigma = 2\pi^5 * k^4 (15 * h^3 * c^2)^{-1}$

Übung 3_5: Schreiben Sie ein Programm, das drei int-Werte Tag, Monat, Jahr einliest und ausgibt, ob die Werte zum Julianischen- oder Gregorianischen Kalender gehören oder ungültig sind.

Deklariieren Sie dazu die logischen Variablen Julian, SchaltjahrJ, Gregor, SchaltjahrG, Monatkorrekt und Tagkorrekt.

- Julian soll genau dann WAHR sein, wenn das Datum zwischen dem 1. 1. 1. und dem 4. 10. 1582 liegt, SchaltjahrJ genau dann, wenn das Jahr durch 4 teilbar ist.
- Ab dem 15. 10. 1582 soll Gregor WAHR sein, SchaltjahrG genau dann, wenn das Jahr durch 4 und nicht durch 100 teilbar ist. Ist es durch 400 teilbar soll SchaltjahrG dagegen ebenfalls WAHR sein
- Schaltjahr ist genau dann WAHR, wenn Julian und JulianJ oder Gregor und GregorG WAHR sind.
- Monatkorrekt ist genau dann WAHR, wenn der Wert zwischen 1 und 12 liegt, Tagkorrekt, wenn er zwischen 1 und der Tagzahl des eingegebenen Monats liegt. Unkorrekt sind auch die Tage vom 5. 10. 1582 bis 14. 10. 1582, weil diese aus dem Kalender zur Korrektur des Julianischen Kalenders derzeit gestrichen wurden.

Geben Sie die Werte der logischen Größen mit einem Begleittext aus z. B.

Julianischer Kalender?	0	Gregorianischer Kalender?	1
Monatkorrekt?	1	Tagkorrekt?	1

4. Anweisungen

Unter einer Anweisung <statement> versteht man eine mit einem Semicolon abgeschlossene Zeichenfolge, die eine Aktion des Rechners auslöst. Das kann ein Schreib-, Lesevorgang, eine Rechenoperation oder ein Befehl für ein externes Gerät sein.

Die Anweisungen eines Programms stehen gewöhnlich in einem Anweisungsteil hinter dem Deklarationsteil eines Blockes und werden normalerweise nacheinander in der geschriebenen Reihenfolge ausgeführt.

Ausnahmen von der sequentiellen Ausführung machen die Wiederholungsanweisungen, bei denen eine oder mehrere Anweisung je nach dem Wert einer Bedingung wiederholt werden oder die Sprunganweisungen, die den Programmablauf mit einer anderen als der als nächsten geschriebene Anweisung fortsetzen.

Im einzelnen kennt C folgende Anweisungsarten:

- Einfache Anweisungen <simple statements>
 - Ausdrucksanweisung <expression statement>
 - Anweisungsblock <compound statement>
 - Leere Anweisung <null statement>
- Schleifenanweisungen <iteration statements>
 - while-Schleife <while statement>
 - for-Schleife <for statement>
 - do-while-Schleife <do-while-statement>
- break-Anweisung <break statement>
- continue-Anweisung <continue statement>
- Entscheidungsanweisungen <selection statements>
 - if-Anweisung <if statement>
 - switch-Anweisung <switch statement>
- Sprunganweisung <goto statement>
- return-Anweisung <return statement>

4. 1. Einfache Anweisungen

Einfache Anweisungen sind im Gegensatz zu Schleifen, bedingten Anweisungen und Sprunganweisungen solche, die streng sequentiell abgearbeitet werden: Ausdrucksanweisung, Anweisungsblock und die leere Anweisung.

4. 1. 1. Ausdrucksanweisung

Wird ein Ausdruck mit einem Semicolon abgeschlossen, so entsteht daraus eine Anweisung.

Beispiele:

```
Betrag = 500;      // Wertzuweisung

printf ( );        // Funktionsaufruf

1;                 // sinnlose, jedoch syntaktisch korrekte Ausdrucksanweisung
```

4. 1. 2. Anweisungsblock

Unter einem Anweisungsblock versteht man eine in geschweifte Klammern eingerahmte Folge von Anweisungen. Er wird syntaktisch als eine einzige Anweisung angesehen und deshalb auch als zusammengesetzte Anweisung oder Verbundanweisung bezeichnet.

Jeder Anweisungsblock kann lokale Deklarationen enthalten. Zum Beispiel ist jeder Funktionsrumpf <function body> ein derartiger Anweisungsblock.

Bemerkungen und Beispiele:

- Man bildet dann einen Block, wenn die Syntax nur eine einzige Anweisung erfordert, jedoch an dieser Stelle mehrere Anweisungen ausgeführt werden sollen.
- Wenn mehrere Anweisungen eine logische Einheit bilden, so empfiehlt der gute Programmierstil, diese zu einem Block zusammenzufassen und um eine Tabulatorstelle einzurücken..
- Die schließende, geschweifte Klammer beinhaltet auch das Abschlußsemicolon. Deshalb sollte unmittelbar nach } kein Semicolon folgen. Im Falle eines Funktionsblockes muß } das letzte Zeichen sein.

```
void main ( )      {
    int i = 1; j = 1;

        {
            int i = 2;    // dies i gilt nur im Unterblock
            i++;
            j++;          // j stammt aus dem übergeordneten Block
        }              // hier sollte kein Semicolon stehen

        printf (" %d %d \n", i, j); // hier muß ein Semicolon stehen
    }                  // hier darf kein Semicolon stehen
```

4. 1. 3. Leere Anweisung

Eine leere Anweisung besteht nur aus dem Semicolon. Z. B. ist

```
;  
;  
;  
;
```

eine Folge von drei Leeraanweisungen.

Bemerkungen:

- Man bildet dann eine Leeraanweisung, wenn die Syntax eine Anweisung erfordert, jedoch an dieser Stelle nichts ausgeführt werden soll.
- Ferner wird man Leeraanweisungen verwenden, um Zeilen zu markieren, in denen im Laufe der Programmentwicklung noch Eintragungen erfolgen sollen.

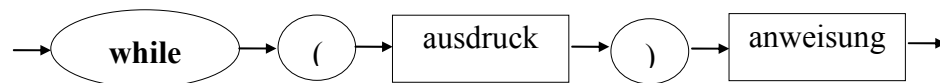
4. 2. Schleifenanweisungen

Mit Hilfe der Schleifenanweisungen (Iterationen) können Programmteile wiederholt abgearbeitet werden. C stellt drei verschiedene Schleifenanweisungen zur Verfügung:

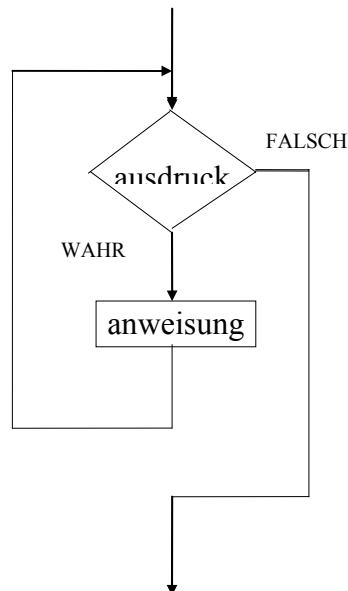
1. while-Schleife
2. for-Schleife
3. do-while-Schleife

4. 2. 1. while-Schleife

Syntax:



Syntaxdiagramm:



Semantik:

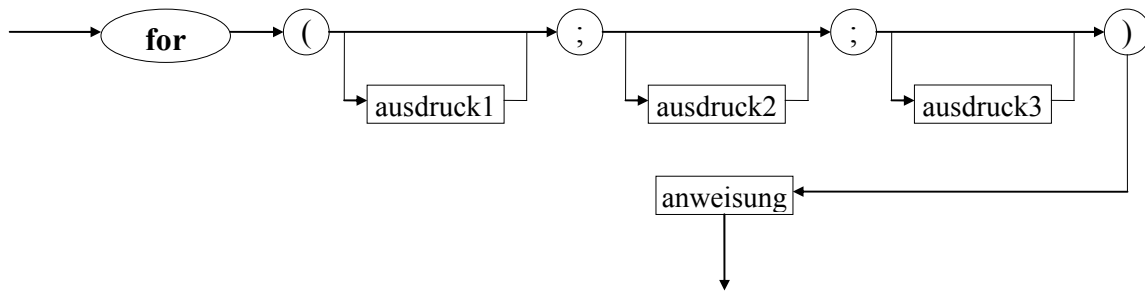
1. Der Ausdruck wird ausgewertet.
2. Ist er WAHR ($\neq 0$), so wird die Anweisung ausgeführt, andernfalls endet die while-Anweisung.
3. Der Vorgang wiederholt sich bei 1.

In folgendem Beispiel werden mit der Funktion `getchar` solange Zeichen eingelesen, bis eine Zeilenumschaltung erfolgt. Nach jeder Eingabe erfolgt eine Ausgabe des zu einem Kleinbuchstaben gehörigen Großbuchstabens.

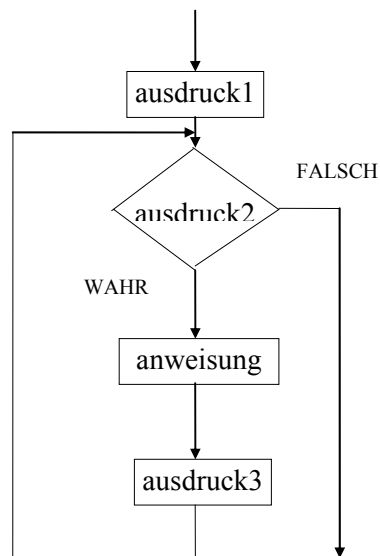
```
while ( (c = getchar ( )) != '\n')
    printf ("%c", c - 32);
```

4. 2. 2. for-Schleife

Syntax:



Syntaxdiagramm:



Semantik:

Bei der for-Schleife handelt es sich lediglich um eine erweiterte while-Schleife der Form:

```
<ausdruck1>;
while (<ausdruck2>) {
    <anweisung>;
    <ausdruck3>;
}
```

Zuerst wird <ausdruck1>, als nächstes die Schleifendurchlaufbedingung <ausdruck2> ausgewertet. Fehlt diese oder ist sie WAHR, so wird zunächst die Anweisung und anschließend, falls vorhanden, <ausdruck3> bewertet. Andernfalls endet die for-Schleife.

Das folgende Beispiel benutzt eine for-Schleife zur Ausgabe einer Funktionstabelle:

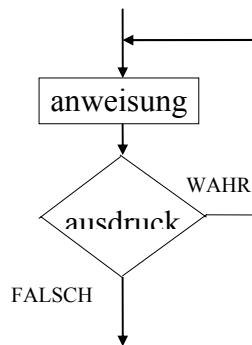
```
for ( i = 1; i < 90; i++)    {
    rad = i *pi/180;
    printf(" %3d %15.10lf\n",i,sin((double)rad));
}
```

4. 2. 3. do-while-Schleife

Syntax:



Syntaxdiagramm:



Semantik:

1. Die Anweisung wird ausgeführt.
2. Der Ausdruck wird ausgewertet.
2. Ist er WAHR ($\neq 0$), so wird die Anweisung wieder ausgeführt, andernfalls endet die do-while-Anweisung.
3. Der Vorgang wiederholt sich bei 2.

Bemerkung und Beispiel:

- Im Gegensatz zur while-Schleife wird bei der do-while-Schleife das Schleifenabbruchkriterium erst am Ende der Schleife geprüft. Das bedeutet, daß die Schleifenanweisung mindestens einmal ausgeführt wird.

```

do  {
    c = getchar ( );           // getchar liest wiederholt ein Zeichen für c ein,
    das
    ...                       // hier weiter verarbeitet wird,
}
while (c < 'A' || c > 'Z'); // solange das Zeichen kein Großbuchstabe ist
  
```

In C dürfen Schleifen ineinander verschachtelt werden. Dabei muß die innerer Schleife vollkommen in der äußeren eingebettet sein.

4. 3. break-Anweisung

Mit Hilfe der Anweisung `break;` kann eine Schleifenanweisung an jeder Stelle unverzüglich verlassen werden. Damit wird die Schleife, deren Anweisungsteil eine `break`-Anweisung enthält, vorzeitig beendet.

4. 4. continue-Anweisung

Die Anweisung `continue;` bricht, im Gegensatz zur `break`-Anweisung, nur den aktuellen Durchlauf einer Schleife ab. Nach `continue;` erfolgt ein Programmsprung zur Schleifenbedingung.

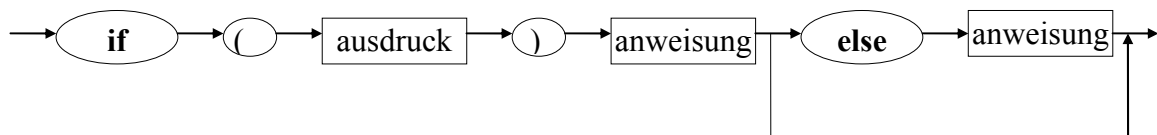
4. 5. Entscheidungsanweisungen

Mit Hilfe einer Entscheidungsanweisung (Selektion) kann, in Abhängigkeit vom Resultat eines Ausdrucks, ein bestimmter Programmteil entweder abgearbeitet oder übersprungen werden. C stellt zwei verschiedene Entscheidungsanweisungen zur Verfügung:

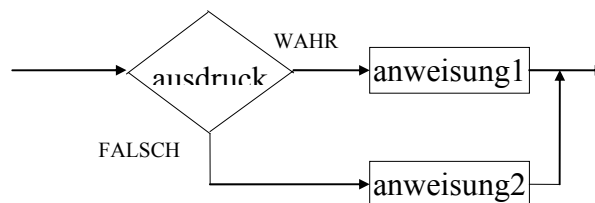
1. if-Anweisung
2. switch-Anweisung

4. 5. 1. if-Anweisung

Syntax:



Syntaxdiagramm:



Semantik:

Zunächst wird der Ausdruck, der dem **if** folgt ausgewertet.

Ist er WAHR, so wird die folgende <anweisung1>, andernfalls die dem **else** folgende <anweisung2> abgearbeitet.

Soll in dem Fall, daß der Ausdruck den Wert FALSCH hat, keine andere Anweisung ausgeführt werden, so kann man den gesamten **else**-Zweig weglassen.

Bemerkungen und Beispiele:

```
// Betrag einer Zahl
// zweiseitige Form
```

```
if (x < 0)
    y = -x;
else
    y = x;
```

oder

```
// Betrag einer Zahl
// einseitige Form
```

```
y = x;
if (x < 0)
    y = -x;
```

- <anweisung1> und <anweisung2> können auch einen Anweisungsblock repräsentieren

```
// Kehrwert
if (x != 0)
{
    y = 1 / x;
    printf ("Kehrwert von %5.2f is %5.2f\n", y);
}
else
    printf ("Es gibt keinen Kehrwert von 0\n");
```

- <anweisung1> und <anweisung2> dürfen wieder if-Anweisungen sein, so daß der Programmablauf in Abhängigkeit von mehreren Alternativen gesteuert werden kann.

```
// Vorzeichen einer Zahl
if (x > 0)
    y = 1;
else
    if (x == 0)
        y = 0;
    else
        y = -1;
```

- Wenn zwei if-Anweisungen verschachtelt werden, wobei nur ein else-Teil auftritt, dann tritt eine Mehrdeutigkeit auf, die dadurch behoben wird, daß die erste if-Anweisung als einseitige und die zweite als zweiseitige Form zu deuten ist.

```
// falsche Betragsberechnung, weil das else nicht zum ersten if gehört
if (x >= 0)
    if (x == 0)
        y = 0;
    else
        y = -x;           // else gehört zum if der 2. Zeile
```

Im Falle $x < 0$, hätte die if-Anweisung keine Wirkung

4. 5. 2. switch-Anweisung

Eine switch-Anweisung sieht folgendermaßen aus:

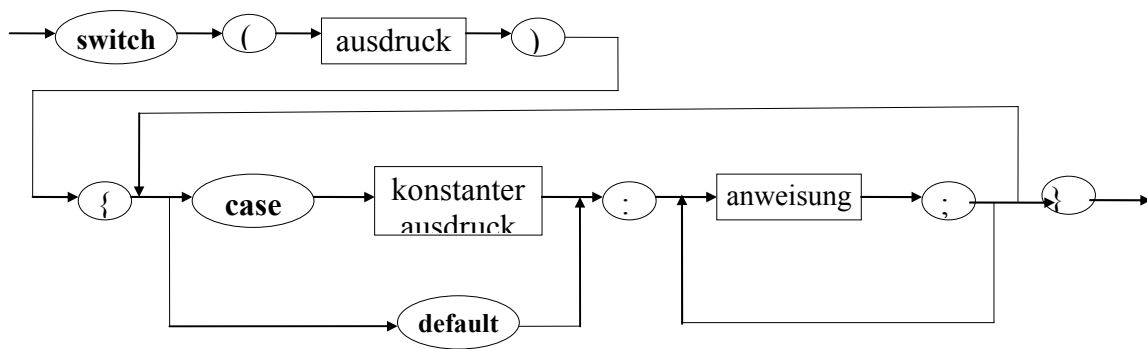
```

switch (ausdruck) {

    case <konstante1>: <anweisung1>;
    case <konstante2>: <anweisung2>;
    case <konstante3>: <anweisung3>;
    ...
    default:           <anweisungN>;
}

```

Syntax:



Semantik:

Zunächst wird der Ausdruck ausgewertet. Er muß einen ganzzahligen Wert liefern.

Das Ergebnis wird anschließend mit den Konstanten <konstante1>, <konstante2>, usw. verglichen, bis es entweder mit einer Konstanten übereinstimmt oder keine weiteren Konstanten mehr vorhanden sind.

Wurde eine Konstante gefunden, die mit dem Wert des Ausdrucks übereinstimmt, dann werden alle Anweisungen, die zwischen dieser Konstanten und dem Ende der switch-Anweisung stehen, ausgeführt.

Wurde keine Konstante gefunden und existiert eine default-Marke, so werden alle Anweisungen zwischen der default-Marke und dem Ende der switch-Anweisung ausgeführt.

Wurde keine Konstante gefunden und existiert keine default-Marke, so wird keine Anweisung ausgeführt.

Bemerkungen und Beispiele:

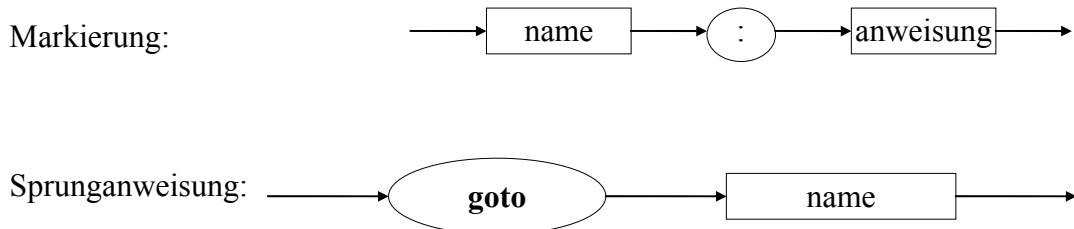
- Alle case-Konstanten, einschließlich default, dürfen nur einmal auftreten.
- Falls nicht alle Anweisungen bis zum Ende ausgeführt werden sollen, kann die switch-Anweisung mit break; verlassen werden.

Beispiel:

```
switch (wochentag) {
    case '0':  printf ("Sonntag\n");
               break;
    case '1':  printf ("Montag\n");
               break;
    case '2':  printf ("Dienstag\n");
               break;
    case '3':  printf ("Mittwoch\n");
               break;
    case '4':  printf ("Donnerstag\n");
               break;
    case '5':  printf ("Freitag\n");
               break;
    case '6':  printf ("Samstag\n");
               break;
    default:   printf ("wochentag nicht zwischen 0 und 6\n");
}
```

4. 6. Sprunganweisung

Syntax:



Semantik:

Mit Hilfe der Anweisung **goto** <Marke>; kann die Fortsetzung des Programmlaufes innerhalb einer Funktion zu einer beliebigen Anweisung gelenkt werden. Das Sprungziel <Marke> ist ein innerhalb einer Funktion lokaler, eindeutiger Name, der gefolgt von einem Doppelpunkt vor die anzuspringende Anweisung zu setzen ist.

Bemerkungen:

- Jeder Anweisung darf eine Marke <label> mit einem Doppelpunkt vorangestellt werden, so daß sie einem unbedingten Sprung als Ziel dienen kann.
- Die Sprunganweisung und die Marke müssen sich innerhalb des gleichen Funktionsblocks befinden.

- Der Name muß eindeutig sein und kann nur gemeinsam mit der goto-Anweisung verwendet werden. Innerhalb eines Funktionsblocks dürfen also unterschiedliche Positionen nicht mit der gleichen Marke gekennzeichnet werden.

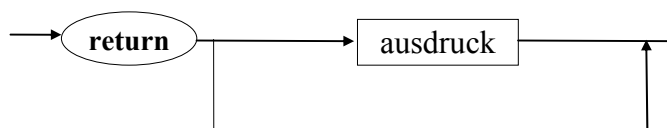
Beispiel:

```
void main ( )
{
    ...
    if (status != 0) goto Fehler;
    ...
    FEHLER: printf ("Fehler\n");
    ...
}
```

- Die goto-Anweisung kann immer und sollte, abgesehen für Testzwecke und zum Sprung aus tief verschachtelten Schleifen, vermieden werden, weil sie zu unübersichtlichen, unstrukturierten und kaum verifizierbaren Programmen führt!

4. 7. return-Anweisung

Syntax:



Semantik:

Nach Ausführung der return-Anweisung wird die Funktion, in der sich diese Anweisung befindet, vorzeitig beendet und in die aufrufende Funktion zurückgesprungen.

Folgt dem **return** kein Ausdruck, so wird diese mit der dem Programmaufruf folgenden Anweisung fortgesetzt. In diesem Fall liegt normalerweise eine 0-wertige Funktion vom Typ **void** vor.

Andernfalls erhält die Funktion den Wert des Ausdrucks und überträgt ihn in die aufrufende Funktion. Der Datentyp der Funktion muß in diesem Fall mit dem des Ausdrucks verträglich sein.

Beispiel:

```
double Quadrat ( double x ) {
    return x * x ;
}
```

Übung 4_1: Schreiben Sie ein Programm, das mit Hilfe einer **while**-Schleife und der Zeicheneingabefunktion `getchar ( )` eine Zeichenfolge einliest, welche die Anzahl der eingelesenen Zeichen zählt und das Ergebnis ausgibt.

Benutzen Sie alternativ eine **for**- und eine **do-while**-Schleife.

Übung 4_2: Schreiben Sie folgende Funktionen mit den Köpfen

1. **unsigned** Deziwert (), welche eine Dualzahl zeichenweise einliest, mit Hilfe des Schemas von Horner den Dezimalwert berechnet und diesen der Funktion zuweist.
2. **void** Zahldarst (**unsigned** Zahl), die rekursiv die Dualstellen einer positiven Integerzahl bestimmt und mit einer führenden Null ausgibt.
3. **void main** (), welche zuerst die Funktion Deziwert und dann mit dem Parameter Deziwert die Funktion Zahldarst aufruft.

Übung 4_3: Bestimmen Sie möglichst genau die Nullstelle x_0 der Funktion

$$y = e^x + x - 2$$

Benutzen Sie das Verfahren der Intervallhalbierung mit dem Anfangsintervall $a = 0$ und $b = 1$.

Geben Sie eine Tabelle mit den Werten a , b , $y(a)$ und $y(b)$ aus!

Übung 4_4: Berechnen Sie mit Hilfe der Trapezregel die Fläche zwischen den Koordinatenachsen und dem Kurvenabschnitt $y = e^x + x - 2$ für $0 \leq x \leq x_0$.

Übung 4_5: Schreiben Sie ein Programm, das mit Hilfe einer **for**-Schleife eine Tabelle der trigonometrischen Funktionen von 0 bis 90 Grad liefert.
Die Formatschlüssel sollen so gewählt werden, daß die Tabelle folgende Form besitzt.

Grad	Sinus	Cosinus	Tangens	Cotangens
0	0.00000000	1.00000000	0.00000000	-
1	0.01745241	0.99984770	0.01745506	57.28996163
		...		
		...		
		...		
89	0.99984770	0.01745241	57.28996163	0.01745506
90	1.00000000	0.00000000	-	0.00000000

Übung 4_6: Formulieren Sie eine Funktion Tagzahl vom Typ long, in der ein Datum in Form einer Zeichenkette [t]t.[m]m.jjjj eingelesen, die Kette entschlüsselt und das Ergebnis int-Variablen Tag, Monat und Jahr zugewiesen wird.

Geben Sie an, wenn und warum die Eingabe unkorrekt ist. Weisen Sie dann der Funktion den Wert -1 zu.

Andernfalls soll als Funktionswert die Anzahl der Tage, die vom 1. 1. 1. bis zu dem eingegebenen Datum verflossen sind, bestimmt und im Hauptprogramm ausgegeben werden.

Stellen Sie sich vor, Sie wissen nicht, an welchem Wochentag Sie geboren sind?

Klären Sie diese Frage, indem Sie in der Funktion die berechnete Tagzahl durch 7 teilen und den Rest bilden.

Sie erhalten als Ergebnis eine Zahl zwischen 0 und 6. Ordnen Sie die Zahl dem Wochentag des eingegebenen Datums zu und geben Sie den Text "fällt auf einen Sonntag." bzw. "fällt auf einen Montag.", usw. aus.

Hinweis:

Der letzte Tag des Julianischen Kalenders war Donnerstag, der 4. Oktober 1582. Am nächsten Tag folgte unmittelbar Freitag, der 15. Oktober und damit der Gregorianische Kalender.

Übung 4_7: Schreiben Sie ein Programm zur Lösung einer quadratischen Gleichung

$$a x^2 + b x + c = 0$$

für beliebige, reelle Koeffizienten a, b, c.

Geben Sie zuerst folgenden Text aus: „Lösung der quadratischen Gleichung $ax^2+bx+c=0$ mit den Koeffizienten:“

a =

b =

c =

Während dieser Ausgabe ist nach a = ein Wert für a, nach b = ein Wert für b und nach c = ein Wert für c einzulesen.

Geben Sie im Falle a = b = 0 den Text "Identität", im Falle a = b = 0, c ≠ 0 den Text "Widerspruch" und im Falle a ≠ 0 den Text "Lineare Gleichung mit der Lösung x =" und die Lösung aus.

Bestimmen Sie im Normalfall a ≠ 0 die reellen Lösungen oder den Real- und Imaginärteil der konjugiert komplexen Lösungen und geben Sie diese in einer der folgenden Formen aus:

2 reelle Lösungen $x_1 = \dots$ $x_2 = \dots$ bzw.

1 Doppellösung $x =$ bzw.

2 konjugiert komplexe Lösungen $z = \dots \pm \dots j$

Ist der Realteil 0 oder der Imaginärteil 1, so sind diese nicht mit auszugeben.

5. Zeiger und Felder

5. 1. Zeiger

5. 1. 1. Adressen

Bekanntlich besteht der Arbeitsspeicher eines Rechners aus adressierten Speicherzellen, deren Größe vom Rechnertyp abhängig jeweils 1 Byte (Bytemaschinen) oder 1 Wort von 2 oder 4 Bytes (Wortmaschinen) beträgt.

Die Adresse ist eine Zahl, welche die laufende Nummer einer Speicherzelle im Arbeitsspeicher angibt, und für die gilt: $0 \leq \text{Adresse} < \text{Rechnerkapazität}$

Die Deklaration **int** i; bewirkt, daß zuerst eine Folge freier Speicherzellen im Arbeitsspeicher gesucht wird, die lang genug ist, um einen int-Wert aufzunehmen. Hat man es zum Beispiel mit einer 2-Byte-Wortmaschine zu tun, dann könnten dies die Speicherzellen mit den Byte-adressen 2234 und 2235 sein.

Der Variablenname i vom Typ int ist also nach der Deklaration charakterisiert durch einen Speicherplatz von 2 Bytes mit der Anfangsadresse 2233 mit einem noch undefinierten Inhalt, wenn er nicht automatisch mit 0 initialisiert wird..

	Adresse	Speicherinhalt
	2236	...
	2235	...
i →	2234	undefiniert
	2233	undefiniert
	2232	...

Erst durch eine Zuweisung z. B. $i = 4321$ wird bewirkt, dass der 16-stellige Dualwert von 4321 auf diesen beiden Speicherzellen abgelegt wird.

Mit $i = 4321 = 10E1_{16} = 0001\ 0000\ 1110\ 0001_2$ erhält man

	Adresse	Speicherinhalt	
	2236	...	
	2235	...	
i →	2234	11100001	4321 = 10E1
	2233	00010000	
	2232	...	

5. 1. 2. Adressoperator und Dereferenzierung

In C gibt es einen speziellen, unitären Operator `&`, nicht zu verwechseln mit dem UND-Operator, der die Adresse einer Variablen liefert. Er heißt Adreß- oder Referenzoperator.

Zum Beispiel liefern `&i` die Anfangsadresse 2233 und `sizeof ( i )` die Größe 2 des Speicherbereiches von `i`.

Man sagt: `&i` ist eine Referenz, ein Zeiger oder Pointer auf `i`.

Beachte:

- Der Operand des Adreßoperators muß ein L-Wert sein; dazu gehören zum Beispiel Variablen und Vektorkomponenten.
- Adreßwerte gehören zu den Ausdrücken und können erhöht oder erniedrigt werden, so daß der Programmierer wie bei Verwendung einer Assemblersprache direkten Zugang zum Arbeitsspeicher hat. Deshalb ist bei solchen Operationen äußerste Vorsicht geboten!
- `&i + 1` bewirkt eine Erhöhung des Adreßwertes um `sizeof ( i )`, nicht um 1!

Adresse	Speicherinhalt
<code>&i + 1</code>	...
<code>&i</code>	<code>i</code>
<code>&i - 1</code>	...
<code>&i - 2</code>	

Unter Dereferenzierung versteht man den Zugriff auf den Speicherinhalt einer durch einen Zeiger referenzierten Adresse.

Dazu verwendet man einen zum Adressoperator inversen, sogenannten Dereferenzoperator `*`. Er wird auf einen Zeiger angewandt und liefert den Inhalt der Speicherzelle, auf die der Zeiger verweist.

Aus `z = &i` folgt also `i = *z`. Die Zeigervariable `z` enthält eine Adresse und `i = *z` den unter dieser Adresse stehenden Inhalt. Folglich ist `*z` eine int-Variable, wie bisher gehabt.

Adresse	Speicherinhalt
<code>z + 1</code>	...
<code>z</code>	<code>*z</code>
<code>z - 1</code>	...
<code>z - 2</code>	

5. 1. 3. Deklaration von Zeigern

Mi Hilfe des Dereferenzoperators lassen sich Zeiger auf jedes beliebige Datenobjekt, außer Bitfeldern, deklarieren.

Zum Beispiel werden durch

```
int *z1, *z2, *z3;
```

drei Zeigervariablen z1, z2, z3 deklariert und damit drei Speicherplätzen zur Aufnahme dreier Adresswerte von int-Speicherplätzen eingerichtet.

Die Definition der Adresswerte erfolgt mit Hilfe der Funktion malloc aus der Headerdatei stdlib.h , auf die eine cast-Operation z. B. (**int ***) angewendet wird.

malloc erhält als Parameter die Speichergröße des verwendeten Typs. Dazu läßt sich die **sizeof**-Funktion verwenden. Z. B. erhält durch die Anweisung

```
z = ( int * ) malloc ( sizeof ( int ) );
```

z den Adresswert der int-Variablen *z.

Durch Wertzuweisung an *z wird der Speicherplatz von zi mit einem Zahlenwert belegt.

Adresse	Speicherinhalt
	...
z3	*z3
z2	*z2
z1	*z1

In folgender Darstellung wird der Name zi durch die Adresse &zi = 1001 repräsentiert, sein Speicherinhalt ist die Adresse zi = 2233. Unter der Adresse 2233 befindet sich der Wert *zi = 4321.

Adressen	Speicherinhalt	
	dual	
2236	...	
2235	...	
2234	11100001	= 4321
*zi ← 2233	00010000	← = 2233
2232	...	
1003	...	
1002	10111001	
zi ← 1001	00001000	

Die Adreßwerte lassen sich wie int-Werte mit printf am besten im X-Format ausgeben.

Beachte:

- Man soll nicht versuchen, Zeigern irgendwelche Zahlen als Adresswerte zuzuweisen, weil sonst die Gefahr besteht, den Systembereich zu manipulieren. Mittels der C-Funktion `malloc ( )` (in C++ auch der Operator `new`) kann Speicherbedarf je nach Bedarf angefordert werden.
- Wird ein Zeiger `z` nicht mehr benötigt, so lässt sich mit der Funktion `free ( z )` (in C++ auch der Operator `delete`) sein Speicher wieder freigeben, so dass dieser während des Programmlaufes anderweitig genutzt werden kann. Die Zeigervariable `z` bleibt dabei erhalten und kann erneut initialisiert werden.
- Einem Zeiger kann durch die Zuweisung der Konstanten **NULL** ein definierter Wert zugewiesen werden, der keine Speicheradresse und keinen Datentyp repräsentiert. Damit lassen sich Listen oder Baumstrukturen abschließen.
- **Ein häufig auftretender Laufzeitfehler ist der Versuch, einen nicht definierten oder NULL-Zeiger zu dereferenzieren, also auf `*z` zuzugreifen.**
- Durch Zuweisung eines Ausdrucks im Deklarationsteil eines Zeigers lässt sich der Zeiger (**nicht sein Dereferenzwert**) initialisieren.

Beispiele:

```
int *z1 = ( int * ) malloc ( sizeof ( int ) );
```

```
int *z2 = NULL;
```

Beispiel zur Veranschaulichung von Zeigeroperationen:

Es wurden mit der Deklaration `int *z1, *z2, *z3;` drei Zeiger deklariert, die jeweils auf einen Speicherbereich zeigen, der eine Zahl vom Typ `int` aufnehmen kann.

Die Namen `z1`, `z2`, `z3` repräsentierten in einem Beispielprogramm die Adressen:

<code>z1: 64FE00</code>		<code>1</code>
<code>z2: 64FDFC</code>	mit den Inhalten	<code>40ED90</code>
<code>z3: 64FDF8</code>		<code>408F64</code>

Die Definitionen

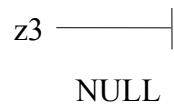
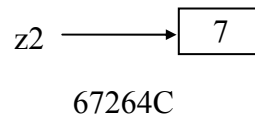
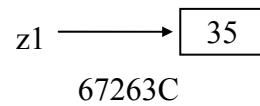
```
z1 = ( int * ) malloc ( sizeof ( int ) ); z2 = ( int * ) malloc ( sizeof ( int ) ); z3 = NULL;
```

ergaben die neuen Speicherinhalte `z1 = 67263C`, `z2 = 67264C` und `z3 = 0`. Unter den Adressen `67263C` und `67264C` standen noch undefinierte Werte.

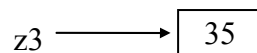
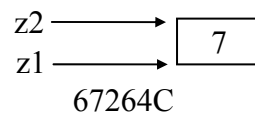
Beachte:

Auf den mit NULL zugewiesenen Wert kann nicht zugegriffen werden. Verwendung von *z3 würde zu einer Fehlermeldung führen!

Mit den Anweisungen *z1 = 35; und *z2 = 7 wurden Dereferenzwerte definiert. Man veranschaulicht die Zeigerdarstellung folgendermaßen:

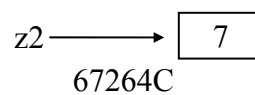
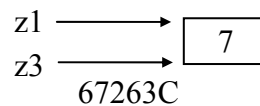


Die Anweisungen z3 = z1; und z1 = z2 bewirkten



67263C

Beachte den Unterschied: Die Anweisungen z3 = z1; und *z1 = *z2 führen zu:



5. 2. Felder

5. 2. 1. Einfach indizierte Felder

Einfach indizierte Größen gleichen Datentyps, wie z. B. Zahlen- und Zeichenfolgen lassen sich wie in der Mathematik auch in C mit einem einheitlichen Variablennamen kennzeichnen und indizieren. Der Index wird durch einen Ausdruck in eckigen Klammern an den Variablennamen angehängt und kann Werte zwischen 0 (einschließlich) und einer bei der Deklaration vorgegebenen Grenze (ausschließlich) annehmen.

Derartige Objekte heißen Felder <arrays> oder Vektoren. Sie werden wie einfache Variable mit einem anhängenden, eckig geklammerten, konstanten Ausdruck deklariert. Der Wert des Ausdrucks legt die Anzahl der damit definierten Feldelemente oder Vektorkomponenten fest.

Die Deklaration `int Vektor [ 5 ];` definiert einen 5-dimensionalen Vektor mit den Komponenten

Vektor [0], Vektor [1], Vektor [2], Vektor [3] und Vektor [4].

Diese können wie einfache Variable benutzt werden.

Setzt man

```
Vektor [ 0 ] = 29;
Vektor [ 1 ] = 31;
Vektor [ 2 ] = 39;
Vektor [ 3 ] = -31;
Vektor [ 4 ] = -19;
```

so werden diese Werte in unmittelbar aufeinanderfolgenden Speicherplätzen abgelegt.

Beachte: Der Name Vektor ist eine Konstante, also ein R-Wert, der auf den Feldanfang Vektor [0] zeigt. Folglich zeigt Vektor + i auf Vektor [i].

Adresse	Speicherinhalt
Vektor + 4	Vektor [4] = -19
Vektor + 3	Vektor [3] = -31
Vektor + 2	Vektor [2] = 39
Vektor + 1	Vektor [1] = 31
Vektor	Vektor [0] = 29

Beachte: Der Index darf nicht außerhalb seiner Grenzen liegen: Kleinster Indexwert: 0, größter Indexwert: Feldweite – 1. Darauf ist besonders zu achten, wenn der Index eine Rechenformel ist.

Für rechnerische Operationen eignen sich besonders gut for-Schleifen. Soll z. B. das Skalarprodukt zweier mit **float** `a [ N ]`, `b [ N ]`; deklarerter Vektoren berechnet werden, so genügt folgende Programmzeile:

```
for ( i = 0, Produkt = 0; i < N; Produkt += a [ i ] * b [ i++ ] );
```

5. 2. 2. Initialisierung

Ein Feld kann bei der Deklaration mit Zahlenwerten initialisiert werden. Dazu wird dem Feld eine Anfangswerteliste in geschweiften Klammern zugewiesen.

Beispiele und Regeln:

float Werte [3] = { 2.34, 5.67, -2.0 }; entspricht

Werte [0] = 2.34;
entspricht Werte [1] = 5.67;
Werte [2] = -2.0;

- Ist die Liste länger als die Dimension, so erfolgt eine Fehlermeldung, ist sie zu kürzer, so werden die letzten Elemente automatisch mit 0 initialisiert.

float Werte [3] = { 2.34, 5.67 }; entspricht

Werte [0] = 2.34;
entspricht Werte [1] = 5.67;
Werte [2] = 0;

- Die Dimensionsangabe in der eckigen Klammer kann weggelassen werden. Sie ergibt sich dann aus der Anzahl der Anfangswerte.

float Werte [] = { 2.34, 5.67, -2.0 }; entspricht **float** Werte [3] = { 2.34, 5.67, -2.0 };

float Werte [] = { 2.34, 5.67 }; entspricht **float** Werte [2] = { 2.34, 5.67 };

- Zeichenketten stellen eine Sonderform des Feldtyps char dar.

1. Die Anfangswerte dürfen zu einem String zusammen gefaßt werden:

2. Der Compiler fügt an die Zeichenkette automatisch das Abschlußzeichen \0 an.

char kette [] = " Hallo"; entspricht **char** kette [] = { 'H', 'a', 'l', 'l', 'o', '\0' };
und mit Dimensionsangabe:

char kette [6] = " Hallo"; bzw. **char** kette [6] = { 'H', 'a', 'l', 'l', 'o', '\0' };

Adresse	Speicherinhalt
Kette + 5	kette [5] = '\0'
kette + 4	kette [4] = 'o'
kette + 3	kette [3] = 'a'
kette + 2	kette [2] = 'a'
kette + 1	kette [1] = 'a'
kette	kette [0] = 'H'

5. 2. 3. Mehrfach indizierte Felder

Als Datentyp eines Feldes darf wieder ein Feldtyp verwendet werden. Dadurch lassen sich Datentypen zur Darstellung mehrfach indizierter Größen z. B. von Matrizen oder Tensoren bilden.

Zum Beispiel ergibt die Deklaration **int** Matrix [3] [2] ein Feld, das in folgender Reihenfolge im Speicher abgelegt wird:

```
Matrix [ 0 ] [ 0 ]
Matrix [ 0 ] [ 1 ]
Matrix [ 1 ] [ 0 ]
Matrix [ 1 ] [ 1 ]
Matrix [ 2 ] [ 0 ]
Matrix [ 2 ] [ 1 ]
```

Das Feld kann auf zwei Arten initialisiert werden. Entweder wird anschaulich für jede Zeile eine eigene Initialisierungsliste benutzt, die in geschweifte Klammern eingeschlossen ist, oder die Werte werden kurz in einer einzigen Liste den Komponenten der Reihe nach zugewiesen.

```
int Matrix [ 3 ] [ 2 ] = { { 4, 6 },
                           {-1, 2},
                           { 5, 9} };
```

oder

```
int Matrix [ 3 ] [ 2 ] = { 4, 6, -1, 2, 5, 9 };
```

Bei mehrfacher Indizierung darf die Dimensionsangabe in der Deklaration nur in der ersten Klammer weggelassen werden. Sie ergibt sich wie bei Vektoren aus der Anzahl der Anfangswerte, sofern eine vollständige, externe Initialisierung vorliegt.

```
int Matrix [ ] [ 2 ] = { { 4, 6 },
                           {-1, 2},
                           { 5, 9} };
```

oder

```
int Matrix [ ] [ 2 ] = { 4, 6, -1, 2, 5, 9 };
```

Für Rechnungen mit mehrfach indizierten Größen verwendet man am besten mehrfach geschachtelte for-Schleifen.

Beispiel: Ausgabe einer Matrix

```
for ( i = 0; i < 3; i++, printf ("\n"))
    for ( j = 0; j < 2; printf ("%d ", Matrix [ i ] [ j++ ] ));
```

5. 2. 4. Feldnamen als Zeiger

Die Zeigereigenschaft eines Feldnamens lässt sich dazu benutzen, statt indizierter Größen Zeiger und damit eine kürzere Schreibweise zu benutzen. Wurde z. B. deklariert:

float Temperatur [6];

so gilt die Identität Temperatur = &Temperatur [0],

d. h. die Konstante Temperatur zeigt auf den Anfang des Vektors.

Beispiel:

Adresse	Speicherinhalt
Temperatur + 3	Temperatur [3] = 1.0
Temperatur + 2	Temperatur [2] = 0.5
Temperatur + 1	Temperatur [1] = 3.3
Temperatur	Temperatur [0] = 5.5

Folglich liefert *Temperatur den gleichen Wert wie Temperatur [0], nämlich 5.5 und die beiden Ausdrücke &Temperatur [i] und Temperatur + i zeigen auf denselben Speicherplatz.

Offensichtlich besteht zwischen Feld und Zeiger ein enger Zusammenhang, der sich dadurch erklären läßt, daß der C-Compiler ein Feldelement in einen Zeiger umwandelt z. B. wird aus dem Feldelement Temperatur [i] der Zeigerausdruck *(Temperatur + i). Dementsprechend kann der Programmierer statt eines Feldes auch den äquivalenten Zeigerausdruck verwenden.

Beispiel: Initialisierung eines Feldes ohne Verwendung von Zeigern

```
for ( i = 0; i < 10; i++ )
    Temperatur [ i ] = 0;
```

bei Verwendung von Zeigern

```
p = Temperatur;
for ( i = 0; i < 10; i++, p++ )
    *p = 0;
```

Bei der Verwendung von Zeigern entfällt die zeitraubende Adreßberechnung, so daß das Programm wesentlich schneller wird.

Beachte: Eine als Zeiger deklarierte Variable darf sowohl inkrementiert als auch dekrementiert werden. Z. B. zeigt die Zeigervariable p nach der Anweisung p += i; bzw. p -= i; i Positionen hinter bzw. vor den Bereich, auf den p zeigt. Allerdings darf der Feldname Temperatur nicht verändert werden, weil er eine Zeigerkonstante ist. Folgende Ausdrücke sind verboten:

Temperatur++; Temperatur--; Temperatur = p;

Bekanntlich wird eine Zeichenkette in einem Vektor vom Typ **char** dargestellt. Dieser enthält die einzelnen Zeichen und ein abschließendes Nullzeichen '\0', die ab einer bestimmten Position in aufeinanderfolgenden Speicherplätzen abgelegt sind. Der interne Zugriff erfolgt auch hier über einen auf das erste Zeichen weisenden Zeiger.

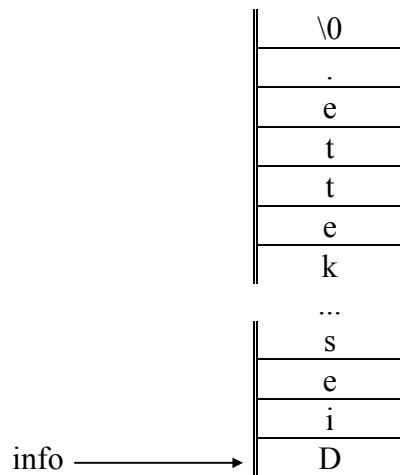
In C können Zeiger auf Zeichenketten auch explizit deklariert werden. Hierzu wird eine Variable als Zeiger auf char vereinbart z. B.

```
char *info;
```

Die Zuweisung

```
info = "Dies ist eine Zeichenkette."
```

weist dem Zeiger namens info die Anfangsadresse der Zeichenkette, also die des Zeichens 'D' zu. Folglich enthält der Ausdruck *info das Zeichen 'D', *info++ das folgende Zeichen 'i' usw.



In folgendem Beispielprogramm wird Zeiger info durch Initialisierung mit dem Text "Dies ist eine Zeichenkette." im Deklarationsteil definiert und ausgegeben. Bei der Ausgabe wird laufend der Zeiger info inkrementiert und sein Dereferenzwert ausgegeben, bis er das Abschlußzeichen '\0' erreicht hat. Damit wird die while-Bedingung (*info) FALSCH und die Ausgabe stoppt.

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

void main ( )    {
    char *info = "Dies ist eine Zeichenkette."; // Definition eines char-Zeigers
    clrscr ( );
    while ( *info )
        putchar ( *info++ ); // Ausgabefunktion für einzelne Zeichen
    getchar ( );
}
```

Im nächsten Beispiel wird statt einer Konstanten ein beliebiger Text mit Kleinbuchstaben von der Tastatur Zeichenkette eingegeben und wieder ausgegeben. In diesem Fall muß die Anfangsadresse definiert werden z. B. mit

```
info = ( char *) malloc ( sizeof ( char ));
```

Ein mit der Anfangsadresse `info` initialisierter Zeiger namens `Laufzeiger` wandert anschließend von Adresse zu Adresse, in deren Speicherstellen jeweils die mit `getchar( )` eingelesenen Zeichen geschrieben werden. Die letzte Speicherstelle erhält das Abschlußzeichen `'\0'`.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

void main ( )    {
    char *info, *Laufzeiger = info = ( char *) malloc ( sizeof ( char ));

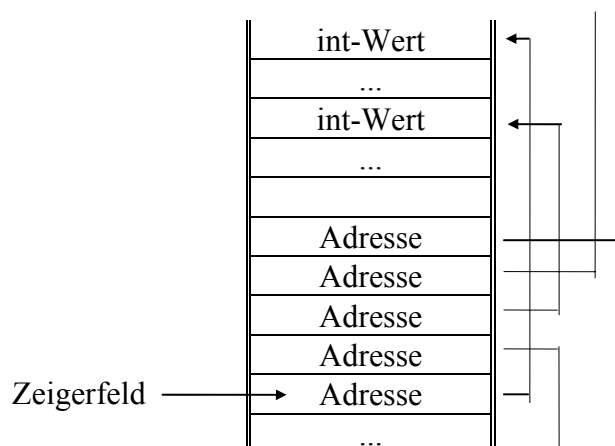
    clrscr ( );
    printf("Bitte eine Zeichenkette von Kleinbuchstaben eingeben: ");
    while ( (*Laufzeiger++ = getchar ( )) != '\n'); // Zeicheneingabe
        *--Laufzeiger = '\0';           // Abschlusszeichen

    printf("Die Zeichenfolge wird in Grossbuchstaben umgewandelt: ");
    while ( *info)
        putchar ( *info++ - 'a' + 'A' ); // Zeichenausgabe
}
```

Die Programmiersprache C bietet dem Programmierer die Möglichkeit, Vektoren mit Zeigern als Komponenten zu deklarieren und damit komplexe Datenstrukturen aufzubauen. Zum Beispiel ergibt die Deklaration

```
int *Zeigerfeld [ 5 ];
```

einen Vektor von 5 int-Zeigern. Der Zugriff erfolgt mit dem Indirektionsoperator * z. B. ist *Zeigerfeld [2] der int-Wert, auf den der Zeiger Zeigerfeld [2] verweist.



Hierbei werden die Adressen in aufeinanderfolgenden Speicherplätzen abgelegt, während ihre Werte auf beliebig verstreute Plätze im Speicher verweisen können.

Übung 5_1: Definieren Sie zwei Vektoren $a = \{ 4; -1; 5 \}$ und $b = \{ 6; 2; 9 \}$ und bestimmen Sie deren Skalarprodukt und Kreuzprodukt.

Übung 5_2: Schreiben Sie ein Programm, das eine beliebige $n \times n$ -Matrix einliest und deren Determinante bestimmt.
Transformieren Sie dazu die Matrix durch Zeilenoperationen in eine Diagonalmatrix um.

Beachten Sie die Regeln der Mathematik:

Ersetzt man eine Zeile durch ihre Summe mit einer anderen, so ändert sich die Determinante nicht.

Multipliziert man eine Zeile mit einem Faktor, so wird auch die Determinante mit diesem Faktor multipliziert.

Vertauschen zweier Zeilen ändert das Vorzeichen der Determinante.

Übung 5_3: Lesen Sie eine Zeichenkette ein und fügen Sie das Abschlußzeichen '\0' an.

a) Definieren Sie zwei Zeiger **char** *Anfangszeiger, *Endezeiger; die auf das erste bzw. letzte Element verweisen und geben Sie die Dereferenzwerte aus.

b) Definieren Sie zwei weitere Zeiger **char** *Minzeiger, *Maxzeiger;
Durchlaufen Sie die Zeichenkette und setzen Min- bzw. Maxzeiger auf die Speicherstelle mit dem der ASCII-Ordnung entsprechenden, kleinsten bzw. größten Wert.
Geben Sie die Dereferenzwerte aus.

Übung 5_4: Schreiben Sie ein Programm, das eine Zeichenkette einliest und die Zeichen der Größe nach geordnet mit dem kleinsten Wert beginnend wieder ausgibt.

Verwenden Sie das Bubblesort-Verfahren:

Seien $z_0, z_1, \dots, z_{n-1}$ die Zeichen einer Folge, so werden von $i = 1$ bis $i = n - 1$ die Zeichen z_i und z_{i-1} miteinander verglichen und genau dann miteinander vertauscht, wenn $z_i < z_{i-1}$ gilt. Dadurch gelangt das größte Zeichen an das Ende der Folge.

Das Verfahren wird von $i = 1$ bis $n - 2$ wiederholt, wobei das zweitgrößte Zeichen an die zweithöchste Stelle rückt. Entsprechende Wiederholung sortiert die Folge, wobei $(n - 1) \cdot (n - 2) / 2$ Vergleiche nötig sind.

Das Bubblesort-Verfahren ist einfach zu programmieren, jedoch mit einer Komplexität $O(n^2)$ ein zeitaufwendiges Verfahren.

6. Funktionen

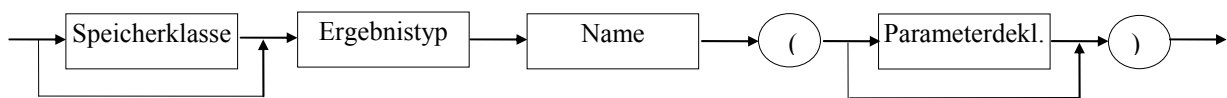
6. 1. Definition einer Funktion

Die Programmierung einer komplexen und komplizierten Aufgabe erfordert als Vorbereitung eine eingehende Analyse und Aufspaltung des Problems in möglichst mehrere Teilaufgaben, die jeweils ohne großen Aufwand als Funktionen formuliert werden können.

Die Definition einer Funktion umfaßt einen Funktionskopf und einen Funktionskörper. Der Funktionskopf enthält:

1. die Speicherklasse (optional),
2. den Ergebnistyp,
3. den Namen der Funktion,
4. die Parameterdeklaration (optional).

Das Syntaxdiagramm des Funktionskopfes sieht folgendermaßen aus:



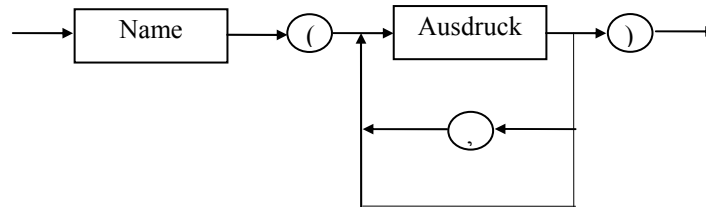
1. Als Speicherklasse können die Schlüsselworte **extern** und **static** verwendet werden, um die Sichtbarkeit der Funktion festzulegen. Fehlt die Angabe, so wird die Speicherklasse **extern** angenommen.
Ist die Speicherklasse **extern**, so kann die Funktion von jeder Compilereinheit aufgerufen werden, ist sie **static**, so kann die Funktion nur innerhalb der Datei, in der sie definiert wurde, aufgerufen werden.
2. Mit der Typangabe wird festgelegt, welchen Datentyp der Funktionswert hat. Strukturierte Typen und Funktionstypen sind als Ergebnistyp nicht zulässig, wohl aber Zeiger auf diese Objekte.
Funktionen, die keinen Wert zurückgeben sollen, erhalten die Typangabe **void**.
3. Jede Funktion muß einen eindeutigen Namen erhalten. Dieser stellt eine konstante Adresse dar, ist also ein Zeiger auf die Funktion.
4. Auf den Funktionsnamen folgt ein rundes Klammersymbol, das eine Parameterliste enthalten kann. Die Parameterliste besteht aus einer Folge von Deklarationen formaler Parameter, die bei jedem Funktionsaufruf mit aktuellen Werten initialisiert und im Funktionskörper als Eingabegrößen verarbeitet werden. Als Datentypen kommen auch Felder in Frage, weil Feldnamen als Zeiger angesehen werden können.

Beachte: Vor jedem Parameter muß eine Typangabe stehen, die Parameter mit ihren Typen werden durch Kommata getrennt. Als Speicherklassenangabe kann **auto** (überflüssig) oder **register** angegeben werden.

Der Funktionskörper ist ein Block mit einem Deklarationsteil lokaler Größen der Speicherklasse **auto** und einem Anweisungsteil. Dieser enthält eine **return**-Anweisung, sofern nicht der Ergebnistyp **void** vorliegt.

6. 2. Parameter

Der Aufruf einer Funktion bewirkt, daß zunächst die formalen Parameter, falls vorhanden, konkrete Werte erhalten. Anschließend wird die Ausführung des Programms mit den Anweisungen der Funktion fortgesetzt. Anzahl und Typ der aktuellen Parameter muß mit der Anzahl der formalen Parameter übereinstimmen. Ein Funktionsaufruf hat folgenden syntaktischen Aufbau:



Man unterscheidet zwei Arten der Parameterübergabe: call-by-value und call-by-reference

6. 2. 1. call-by-value

Diese Art des Aufrufes hat zur Folge, daß der Speicherplatz des formalen Parameters den Wert des aktuellen Parameters erhält. Jede Änderung des formalen Parameters in der Funktion hat keine Auswirkung auf den zugehörigen aktuellen Parameter. Das bedeutet, daß der Wert des aktuellen Parameters vor und nach dem Funktionsaufruf derselbe ist.

In diesem Fall werden Parameter ausschließlich dazu verwendet, Werte an die aufgerufene Funktion zu übergeben. Als solche können beliebige Ausdrücke verwendet werden.

6. 2. 2. call-by-reference

Diese Art des Aufrufs liegt dann vor, wenn es sich bei den Parametern um Zeiger handelt. In diesem Falle wird der Funktion nicht der Inhalt eines Speicherplatzes, sondern dessen Adresse übergeben. Das bedeutet, daß aktueller und zugehöriger formaler Parameter den gleichen Speicherplatz repräsentieren, und jede Änderung des formalen Parameters in der Funktion gleichzeitig den Wert des aktuellen Parameters ändert.

Man benutzt diese Form der Parameterübergabe, um zusätzlich zum Funktionswert weitere Werte von der Funktion in das aufrufende Programm zu übertragen z. B. `scanf ("%d",&x)`.

Beispiel: Vergleich zweier Funktionen, die den Inhalt zweier Variablen vertauschen sollen

```

void Austausch (float x, float y) {
// call by value
    float hilf = x;

    x = y;
    y = hilf;
}
  
```

```

void Austausch (float *x, float *y) {
// call by reference
    float *hilf = x;

    x = y;
    y = hilf;
}
  
```

Aufruf: Austausch (a, b);

Austausch (&a, &b);

Das linke Beispiel ist falsch, weil das Vertauschen von x und y keine Auswirkungen auf a und b hat. Das rechte Beispiel ist richtig, a und x, sowie b und y repräsentieren die gleichen Speicherplätze, so daß das Vertauschen von x und y den gleichen Vorgang bei a und b bewirkt.

6. 2. 3. Felder als Parameter

Die Liste der formalen Parameter kann auch Felddeklarationen enthalten. Dabei darf, wie bei Initialisierungen möglich, der erste Feldindex weggelassen werden. Als aktueller Parameter wird nur der Name eines typgleichen Feldes angegeben.

Die folgende Funktion berechnet die Länge eines Vektors.

```
double Betrag ( double v [ ] ) {      // Funktionskopf

    return ( sqrt ( v [ 0 ] * v [ 0 ] + v [ 1 ] * v [ 1 ] ) );

}

void main ( ) {                        // Hauptfunktion mit Ergebnisausgabe
    double Vektor [ ] = { 1, 2 };      // Testbeispiel

    printf ("Betrag = %lf", Betrag ( Vektor ));    // aktueller Parameter ( = Zeiger)
}
```

Der Übergabemechanismus bei Feldern ist derselbe wie bei Zeigern, also von der Art call-by-reference. Das bedeutet, daß eine Änderung von Feldelementen in der Funktion gleichzeitig die Feldelemente des Hauptprogramms ändert.

Das folgende analoge Beispiel verwendet diese Tatsache zum Transponieren einer Matrix:

```
void Transp ( double M [ ] [ 2 ] ) {      // Funktionskopf
    int i, j;
    double hilf;

    for ( i = 1; i < 2; i++ )
        for ( j = 0; j < i; j++ ) {
            hilf = M [ i ] [ j ];
            M [ i ] [ j ] = M [ j ] [ i ];
            M [ j ] [ i ] = hilf;
        }
}

void main ( ) {                          // Hauptfunktion mit Ergebnisausgabe
    int i, j;
    double Matrix [ ] [ 2 ] = { { 1, 2 }, { 3, 4 } };

    Transp ( Matrix );
    for ( i = 0; i < 2; i++ ) {
        for ( j = 0; j < 2; j++ )
            printf ("%lf ", Matrix [ i ] [ j ] );
        printf ("\n");
    }
}
```

6. 2. 4. Parameter in main ()

Die Hauptfunktion main läßt sich ebenfalls mit bestimmten, formalen Parametern ausrüsten. Als solche haben sich die Namen einer int-Zahl argc und eines char-Zeigerfeldes argv [] eingebürgert. Ferner kann der Funktion main mit return ein Funktionswert vom Typ int zugeordnet werden. Damit lautet der Funktionskopf

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

Die aktuellen Parameter sind Zeichenketten, die beim Funktionsaufruf in der DOS-Ebene angegeben werden. Lautet z. B. der Dateiname Uebung.cpp, so entsteht nach dem Übersetzen und Binden die ausführbare Datei Uebung.exe. Liegt diese z. B. in einem Verzeichnis C:\user, so bewirkt der DOS-Aufruf

```
C:\user\Uebung  Guten  Tag
```

daß dem Parameter argc die Anzahl dieser Zeichenketten, nämlich der Wert 3 zugeordnet wird. Ferner werden vier Zeiger auf die Zeichenketten definiert, nämlich

```
argv [0] auf "C:\user\Uebung"
argv [1] auf "Guten"
argv [2] auf "Tag"
argv [3] auf NULL
```

Die Parameterwerte lassen sich im Programm verarbeiten oder zur gezielten Steuerung des Programmablaufes verwenden. In der Praxis besteht eine Zeichenkette oft aus einer Ziffernfolge, um dem Hauptprogramm eine Zahl zu übergeben (z. B. telnet <Rechnername> <Portnummer>). In diesem Fall benutzt man zur Typumwandlung der Parameter die Funktionen atoi (ASCII to int) oder atof (ASCII to float) aus der Headerdatei <stdlib.h>.

Beispiel: Aufruf des Programms prog.exe mit den Parametern 4711 und 3.14159

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

int main (int argc, char *argv [ ] )    {
    int i;
    float x;

    for ( i = 0; i < argc; i++ )           // Kontrollausgabe
        printf (" argv[%d]  %s\n", i, argv [ i ] );    // der eingelesenen Parameter

    if ( argc != 3 )    {
        printf ("usage: <%s> <INTEGER> <FLOAT> \n", argv[0]);
        exit ( 1 );           // Falsche Anzahl von Argumenten
    }

    i = atoi ( argv[1] );      // atoi liefert den int-Wert  4711
    x = atof ( argv[ 2 ] );    // atof liefert den float-Wert  3.14159
    ...
}
```

6.3. Rekursive Funktionen

Unter einer rekursiven Funktion versteht man eine auf der Menge der natürlichen Zahlen $\{n\}$ und evtl. weitere Argumente x auf folgende Art definierte Funktion

$$\begin{aligned} f(0, x) &= g(x), \\ f(n, x) &= h(f(n-1), x) \\ \text{wobei } g \text{ und } h &\text{ vorgegeben sind} \end{aligned}$$

Das bedeutet, daß die Funktion von n eindeutig bestimmt ist, wenn sie als Funktion von $n-1$ bekannt ist. Wenn die Funktion für einen Anfangswert z. B. $n = 0$ definiert wird, ist sie nach dem Prinzip der vollständigen Induktion dann auch für alle $n > 0$ definiert.

Beispiele: Fakultät, Potenzfunktion und Fibonacci-Zahlen

```
0! = 1      n! = n*(n - 1)!
long fakultaet ( int n )  {
    return ( n ) ? n*fakultat ( n-1 ) : 1;
}
```

```
a0 = 1      an = a*an-1
long potenz ( int n )  {
    return ( n ) ? a*potenz ( n-1 ) : 1;
}
```

```
fibo(n) = n, falls 0 ≤ n < 2 , sonst fibo(n) = fibo(n-1)+fibo(n-2)

long fibo ( int n )  {
    return ( n < 2 ) ? n : fibo ( n-1 ) + fibo ( n-2 );
}
```

C ermöglicht die Verwendung rekursiver Funktionen durch Aufbau eines Stacks, dessen Elemente die Parameter, Funktionswerte und lokalen Variablen für jeden Aufruf speichern. Nach Erreichen der Funktion für $n = 0$, wird deren Wert gebildet und durch die pop-Funktion zur Funktion für $n = 1$ zurückgekehrt. Anschließend wird diese unter Verwendung des Funktionswertes von $n = 0$ berechnet und wieder mit pop zur Funktion für $n = 2$ zurückgekehrt. Der Vorgang wiederholt sich, bis $f(n)$ berechnet werden kann.

Zur Speicherersparnis müssen die Funktionen effektiv sein, d. h. mit möglichst wenig Rekursionsschritten auskommen.

Beispiel: O. a. Potenzfunktion besitzt die Rekursionstiefe n , effektiver ist die Funktion $a^n = (a^{n/2})^2$, deren Rekursionstiefe nur etwa $\lg n$ beträgt ($\lg = \text{Logarithmus dualis}$)

```
float Potenz ( float a, int n )  {
    float Q;

    if ( n ) {
        Q = Potenz ( a, n/2 );
        return ( n%2 ) ? a*Q*Q: Q*Q;
    } else return 1;
}
```

6. 4. Zeiger auf Funktionen

Zeiger auf Funktionen dienen dazu, Funktionen als Parameter anderen Funktionen zu übergeben. Als formaler Parameter wird ein Zeiger mit einer Parametertypenliste verwendet. Aktueller Parameter ist der Name einer vorher definierten Funktion.

In folgendem Beispiel wird eine Funktion dargestellt, die das numerische Integral über eine allgemeine Funktion $f(x)$ von a nach b mit Hilfe der Trapezregel für N Stützstellen berechnet.

Der Funktionskopf enthält neben den formalen Parametern für a , b und N den Funktionszeiger **double (*f)(double)** zur Übernahme einer einparametrischen Funktion f vom Parameter- und Rückgabetyt **double**.

Als Testbeispiele dienen zwei Funktionen **Quadrat** und **Kubus**.

```
// Numerische Integration
//
#include <stdio.h>
#include <conio.h>
#include <math.h>

double Quadrat(double x)  {
    return x*x;
}

double Kubus(double x)    {
    return x*x*x;
}

double Integral (double (*f)(double), double a, double b, long N)      {
    long i = 1;
    double dx = (b - a)/N, Summe = ( f(a) + f(b) )/2;

    for ( ; i < N; )
        Summe+=f(a + i++*dx);
    return Summe*dx;
}

void main()  {
    long N;

    printf("Durch wieviel Trapeze soll das Integral approximiert werden? ");
    scanf("%ld",&N);
    printf("Numerisches Integral fuer N = %ld:  %-20.17lf",N,Integral(Quadrat,0,1, N));
    printf("%-20.17lf\n",Integral(Kubus, 0, 1, N));
}
```

Ergebnisse für verschiedene N :

Numerisches Integral fuer $N = 10$: 0.335000000000000000 0,252500000000000000
 Numerisches Integral fuer $N = 1000$: 0.333333999999999995 0,250000002500000000
 Numerisches Integral fuer $N = 100000000$: 0.333333333333322251 0,24999999999995351
 (das letzte Beispiel benötigte ca. 20 sec)

6. 5. Bibliotheksfunktionen

Bibliotheksfunktionen sind kein Bestandteil der Sprache C, sondern beliebig ergänzbare Zusatzfunktionen, die sich in sogenannten Header-Dateien befinden. Diese können auch Deklarationen, symbolischen Konstanten, Makros und Typdefinitionen enthalten. Sie sind durch einen Namen mit dem Anhang .h gekennzeichnet. Der Aufruf einer Bibliotheksfunktion erfordert, daß zuvor die entsprechende Headerdatei dem Compiler bekannt gemacht wird. Das Einbinden der Datei kann an beliebiger Stelle des Programms mit Hilfe der Direktive `#include` erfolgen, wird jedoch üblicherweise am Programmanfang durchgeführt.

Für eine Headerdatei namens `header.h` kann die Direktive lauten:

```
#include <header.h>
```

Dann sucht der Compiler nach der angegebenen Datei zuerst in bestimmten, in der Entwicklungsumgebung spezifizierten Verzeichnissen.

```
#include "header.h"
```

Bei dieser Angabe sucht der Compiler zuerst im aktuellen Verzeichnis, dann in den in der Entwicklungsumgebung spezifizierten Verzeichnissen.

```
#include "C:\\progr\\header.h"
```

Wird ein Laufwerk und/oder Verzeichnis angegeben, so sucht er nur in dem angegebenen Pfad.

6. 5. 1. Ein-/Ausgabefunktionen

Um Daten in eine Datei zu schreiben oder Daten aus einer Datei zu lesen zu können, muß mit `#include <stdio.h>` die Headerdatei `stdio.h` (standard-input-output) eingebunden werden, damit der Compiler auf die Standardein- und Standardausgabefunktionen zugreifen kann.

1. Eine Datei, in die Daten geschrieben oder aus der Daten gelesen werden sollen, muß zuerst geöffnet werden. Angenommen, man will im Verzeichnis `C:\user` eine Datei mit dem Namen `Daten.dat` einrichten, die mit Zeichen beschrieben werden soll, so geschieht dies durch die Zuweisung

```
fp = fopen ("C:\\user\\Daten.dat", "w");
```

`fp` ist ein mit `FILE *fp;` deklarierter Name eines Zeigers auf die Datenstruktur `FILE` aus `stdio.h`, welche Dateiinformationen, wie Zugriffsart, Adresse eines Pufferbereichs usw. enthält. Der erste Parameter von `fopen` enthält die Dateiangabe mit Verzeichnis als String, der zweite `"r"`, `"w"` oder `"a"` (evtl. `"r+"`, `"w+"` oder `"a+"`). Es bedeuten:

- r: Die Datei wird nur zum Lesen geöffnet.
- w: Die Datei wird zum Schreiben geöffnet; existiert bereits eine Datei dieses Namens, so wird deren Inhalt gelöscht und überschrieben.
- a: Die Datei wird zum Schreiben geöffnet, existiert bereits eine Datei dieses Namens, so wird der neue Inhalt an den bestehenden hinten angefügt.

2. Die Eingabe eines Zeichens erfolgt z. B. mit

```
putc ( c, fp );
```

c ist eine Variable vom Typ int, die das Zeichen enthält, fp ein Filepointer auf die Datei C:\user\Daten.dat und putc eine Funktion vom Typ int, welche das Zeichen hinter die letzte Schreibposition setzt und das Zeichen oder die Konstante EOF (End of File) als Rückgabewert liefert.

Die Schreibposition wird durch einen Schreib-/Lesezeiger festgelegt, der bei jedem Schreib-/Lesevorgang eine Positioniterrückt, sich jedoch auch mit Hilfe der Funktion

```
fseek ( fp, offset, basis );
```

verschieben läßt. Die neue Position ist dann

offset,	falls basis = 0,
alte Position+offset,	falls basis = 1,
Dateiende+offset,	falls basis = 2.

3. Um ein Zeichen aus einer mit

```
fp = fclose ( " C:\\user\\Daten.dat", "r");
```

zum Lesen geöffneten Datei zu lesen, dient die Funktion

```
getc ( fp );
```

Sie liefert das Zeichen, auf das der Schreib-/Lesezeiger zeigt und setzt diesen eine Position weiter. Nach Lesen des letzten Zeichens erhält er den Wert EOF (= -1).

4. Sobald die Datei vom Programm nicht mehr benötigt wird, sollte sie mit der Funktion

```
fclose ( fp );
```

geschlossen werden, womit Zwischenspeicher geleert, noch nicht abgelegte Daten auf Datenträger geschrieben und die Verbindung zwischen Filepointer und Programm aufgelöst werden. Der Rückgabewert ist bei Erfolg 0, sonst EOF.

In folgendem Beispiel wird eine Datei zum Schreiben geöffnet, mit Zeichen von der Tastatur beschrieben und wieder geschlossen. Danach wird eine Datei zum Lesen geöffnet, ihr Inhalt zeichenweise gelesen und ausgegeben und die Datei wieder geschlossen

```
#include<stdio.h>
void main ( ) {
    char c;
    FILE *fp = fopen ( "C:\\user\\Datei.dat", "w" );

    while ( ( c = getchar ( ) ) != '\n')
        putc ( c, fp );
    fclose ( fp );
}
```

```

void main ( )    {
    char c;
    FILE *fp = fopen ( "C:\\user\\Datei.dat", "r" );

    while ( ( c = getc ( fp ) ) != EOF )
        putchar ( c );
    fclose ( fp );
}

```

Die Datei stdio.h liefert drei Konstanten `stdin`, `stdout` und `stderr`, welche die Filepointer der Standarddateien für die Tastatureingabe, Bildschirmausgabe und Fehlerausgabe repräsentieren. `getc (stdin)` hat dieselbe Bedeutung wie `getchar ( )` und `putc ( c, stdout )` wie `putchar ( c )`.

Von `stdio.h` zur Verfügung gestellte Funktionen für die blockorientierte Ein-/Ausgabe:

size_t fread (**void** *puffer, **size_t** groesse, **size_t** anzahl, **FILE** *dstrom);

liest anzahl Elemente der Größe groesse aus dem Datenstrom dstrom in den Speicherbereich, auf den puffer verweist. Rückgabewert ist die Anzahl der fehlerfrei gelesenen Datenelemente.

size_t fwrite (**void** *puffer, **size_t** groesse, **size_t** anzahl, **FILE** *dstrom);

schreibt anzahl Elemente der Größe groesse aus dem Speicherbereich, auf den Puffer verweist, in den Datenstrom dstrom. Rückgabewert ist die Anzahl der tatsächlich geschriebenen Datenelemente.

Für die formatierte Ein-/Ausgabe stehen die Funktionen `fscanf` und `fprintf`, bei Standardein-/ausgabe `scanf` und `printf` zur Verfügung.

Beachte: Die Funktionen sind vom Ergebnistyp **int** und liefern als Resultat die Anzahl der tatsächlich gelesenen bzw. geschriebenen Objekte.

6. 5. 2. Weitere Bibliotheksfunktionen

Neben den mathematischen Funktionen aus `<math.h>`, den Ein-/Ausgabefunktionen aus `<stdio.h>` gibt es noch weitere wichtige Funktionen z. B. zur Speicherverwaltung aus `<stdlib.h>` und zur Verarbeitung von Zeichenketten aus `<string.h>`.

Wichtige Beispiele aus `<stdlib.h>`:

void *calloc (**size_t** anzahl, **size_t** groesse);

fordert Speicherplatz für ein Array bis zu 64 Kbyte an. Das angelegte Array wird für anzahl Elemente der Größe groesse eingerichtet und mit Nullen initialisiert.

void *malloc (**size_t** groesse);

fordert einen Speicherblock der angegebenen Größe an; der zurückgegebene ***void**-Zeiger ist mit einem cast-Operator in den gewünschten Typ umzuwandeln

void *realloc (void * puffer, size_t groesse);

paßt die Größe eines mit calloc oder malloc angeforderten Speicherblockes an und gibt einen Zeiger zum neu zugeordneten Speicherblock zurück oder einen NULL-Zeiger, wenn der Speicher nicht eingerichtet werden konnte

void *free (void * addr);

gibt den mit calloc, malloc oder realloc angeforderten Speicherbereich, auf den addr verweist, wieder frei

double atof (const char * string); ascii to float

gibt den zur Zeichenkette, auf die string zeigt, korrespondierenden double-Wert zurück; dabei werden Blanks übersprungen und die Umwandlung nur bis zum ersten nicht identifizierbaren Zeichen durchgeführt

int atoi (const char * string); ascii to int

gibt den zur Zeichenkette, auf die string zeigt, korrespondierenden int-Wert zurück; dabei werden Blanks übersprungen und die Umwandlung nur bis zum ersten nicht identifizierbaren Zeichen durchgeführt

long atol (const char * string); ascii to long

gibt den zur Zeichenkette, auf die string zeigt, korrespondierenden long-Wert zurück; dabei werden Blanks übersprungen und die Umwandlung nur bis zum ersten nicht identifizierbaren Zeichen durchgeführt

Wichtige Beispiele aus <string.h>:

void *memcpy (void * ziel, const void *quelle, size_t anzahl);

kopiert anzahl Zeichen vom Speicherbereich quelle in den Speicherbereich ziel.

void *memmove (void * ziel, const void *quelle, size_t anzahl);

verlegt anzahl Zeichen vom Speicherbereich quelle in den Speicherbereich ziel.

int memcmp (const void *puf1, const void *puf2, size_t anzahl);

vergleicht die ersten anzahl Zeichen von puf1 mit denen von puf2, bis ein Unterschied entdeckt wird, der Rückgabewert ist negativ, wenn puf1 < puf2, 0, wenn puf1 == puf2, positiv, wenn puf1 > puf2.

char *strcpy (**char** *ziel, **char** *quelle);

kopiert alle Zeichen des Speicherbereiches, auf den quelle zeigt, in den Speicherbereich, auf den ziel zeigt. Das Verhalten der Funktion ist undefiniert, wenn der Quellbereich den Zielbereich überlappt.

char *strcat (**char** *ziel, **const char** *quelle);

hängt die Zeichenkette quelle, einschließlich des Abschlußzeichens \0 an die Zeichenkette ziel an, dabei wird das Abschlußzeichen von ziel überschrieben.

int *strcmp (**const char** *string1, **char** *string2);

vergleicht, ob lexikographisch $s1 < s2$, $s1 = s2$ oder $s1 > s2$ gilt, das Ergebnis ist dementsprechend negativ, 0 oder positiv.

int *strlen (**char** *string);

liefert die Anzahl der Zeichen von string ohne das Abschlußzeichen \0.

6. 6. Makrofunktionen

Unter einem Makro versteht man eine mit Hilfe der #define-Direktive definierte Funktionsformel.

Eine wichtige Eigenschaft eines Makros ist seine Parametrisierbarkeit. Die Parameter eines Makros werden durch Komma getrennt in Klammern eingeschlossen und hinter den Makronamen gesetzt.

Als Standardbeispiel eines parametrisierten Makros dient die Berechnung des Maximums zweier Zahlen:

```
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

Der Aufruf $x = \text{MAX}(2,4)$; wird ersetzt durch

```
x = ((2) > (4) ? (2) : (4));
```

Folgendes Beispiel

```
#define SQUARE(A) ((A)*(A))
```

erlaubt die Verwendung einer Funktion SQUARE (y) und bewirkt, daß diese durch $((y) * (y))$ ersetzt wird.

Beachte: y kann ein Ausdruck sein. Deshalb ist sorgfältige Klammerung wichtig!

#define SQUARE(A) (A*A) wäre falsch, weil der Aufruf SQUARE (a + b) zu $a + b * a + b$ führen würde.

Übung 6_1: Schreiben Sie Funktionen für folgende Matrizenoperationen:

- a) Einlesen einer nxn-Matrix
- b) Anwendung des Inversionalgorithmus nach Gauß
Rückgabewerte seien die Determinante als Funktionswert, der Rang und die inverse Matrix, sofern sie existiert, als Referenzparameter
- c) Ausgabe einer nxn-Matrix
- d) Produkt einer lxm-Matrix mit einer mxn-Matrix

Die Hauptfunktion main () benutze diese Funktionen für folgende Aufgaben:

- a) Einlesen einer beliebigen 3 x 3 - Matrix und Ausgabe ihrer Determinante und ihres Ranges,
- b) Ausgabe deren inverser Matrix, sofern eine solche existiert,
- c) Berechnung und Ausgabe des Produktes der Matrix mit ihrer Inversen.

Übung 6_2: Schreiben Sie eine a) iterative, b) rekursive

Funktion zur Bestimmung von Binomialkoeffizienten

$$\binom{n}{m} = \prod (n - i) / (i + 1), \quad 0 \leq i < m \quad (\text{iterative Form})$$

$$\binom{n}{m} = \binom{n-1}{m-1} * n / m \quad (\text{rekursive Form})$$

Benutzen Sie für m und n den Datentyp int, für die Funktion den Datentyp long. Testen Sie die Gültigkeit beider Formen hinsichtlich des Definitionsbereiches.

Berechnen Sie die Anzahl der Kombinationsmöglichkeiten beim Lotto: $\binom{49}{6}$.

Übung 6_3: Schreiben Sie eine rekursive Funktion mit dem Kopf

int Vergleich (**char** *Text1, **char** *Text2),

die zwei Texte auf ihre lexikographische Anordnung vergleicht und -1 oder 0 oder +1 ergibt, je nachdem ob, *Text1 < *Text2, *Text1 = *Text2 oder *Text1 > *Text2 gilt.

Übung 6_4: Öffnen Sie zwei Dateien A:\vorname.dat und A:\hausname.dat zum Schreiben und geben Sie Ihren Vornamen und Hausnamen ein. Kopieren Sie die erste Datei in eine dritte Datei A:\name.dat, fügen noch ein Leerzeichen an und hängen Sie den Inhalt der zweiten an den der dritten.

Geben Sie den Inhalt der dritten Datei auf dem Bildschirm aus.

7. Elementare Datenstrukturen

Unter einer Datenstruktur versteht man eine Menge logisch zusammengehöriger Variablen, die unter einem einheitlichen Namen zusammengefaßt wurden. Die einzelnen Variablen werden als Komponenten bezeichnet. Zum Beispiel ist ein Feld eine Datenstruktur. Es gibt jedoch weitere als Strukturtypen bezeichnete Datenstrukturen, deren Komponenten unterschiedlichen Datentypen angehören und sogar wieder strukturiert sein können.

Der Strukturtyp entspricht dem record-Typ in Pascal und wird zum Beispiel zur Darstellung komplexer Zahlen oder Aufstellung von Datensätzen benutzt. Es können Pointer auf Strukturen zeigen oder Zeiger als Komponenten in einer Struktur enthalten sein. Dadurch lassen sich mit Hilfe von Strukturtypen lineare Listen und binäre Bäume konstruieren.

7. 1. Allgemeiner Strukturtyp

Die Deklaration eines allgemeinen Strukturtyps wird durch das Schlüsselwort **struct** eingeleitet, dem ein Name folgen kann, um ihm zur weiteren Verwendung eine eigene eindeutige Bezeichnung zu geben. Es folgt eine in geschweiften Klammern eingerahmte Komponentenliste. Jede Komponente besteht aus Variablendeklarationen, die wiederum Strukturen sein können. Dabei müssen die Variablennamen aller in einem Programm vorkommenden Strukturen eindeutig sein.

Zur Unterscheidung der Datentypbezeichner von Variablenbezeichnern empfiehlt es sich, den Typbezeichnern ein Präfix **t** voranzustellen.

Beispiele von Strukturtypen zur Darstellung

einer komplexen Zahl:	<pre> struct tComplex { float re, im; }; </pre>
eines Datums:	<pre> struct tGeburtstag { int Tag, Monat, Jahr; }; </pre>
einer Adresse:	<pre> struct tAdresse { char *Strasse; long PLZ; char *Ort; }; </pre>
einer Personalie:	<pre> struct tPerson { char *Name, *Vorname; struct tAdresse Adresse; struct tGeburtstag Geburtstag; }; </pre>

Die Deklaration von Variablen vom Strukturtyp kann auf zwei Arten erfolgen:

1. Die Variablenliste wird zwischen die schließende, geschweifte Klammer und das Semicolon geschrieben. Sollen z. B. die Variablen z1, z2, z3 vom Typ Complex und Student1, Student2 vom Typ Person sein, so kann man schreiben

```

struct tComplex    {
    float re, im;
} z1, z2, z3;

struct tPerson      {
    char *Name ;
    char *Vorname;
    struct tAdresse;
    struct tGeburtstag Geburtstag;
} Student1, Student2;

```

2. Nach Definition des Strukturtyps schreibt man **struct** <Strukturname> <Variablenliste>;
z. B:

```

struct tComplex z1, z2, z3;
struct tPerson Student1, Student2;

```

Der Zugriff auf eine Strukturkomponente erfolgt durch Angabe des Variablennamens, gefolgt von einem Punkt und Angabe des Komponentennamens z. B.

```

z1.re = 2; z1.im = 3;    // bedeutet "symbolisch" z1 = 2 + 3j

z3.re = z1.re + z2.re;   // ergibt die Summe zweier komplexer Zahlen
z3.im = z1.im + z2.im;

```

Die Personaldaten im letzten Beispiel lassen sich mit Hilfe einer Eingabefunktion folgendermaßen zuweisen:

```

void Eingabe ( char *Zeiger; char *Text)    {
    char *Laufzeiger = Text;

    while (*Laufzeiger) // Ausgabe eines Textes zur Information
        putchar(*Laufzeiger++);

    Laufzeiger = Zeiger; // zeigen auf den Ort des 1. einzulesenden Zeichens
    while ((*Laufzeiger++ = getchar ( )) != '\n');
    *--Laufzeiger = '\0'; // zeigt auf den Ort des Abschlusszeichens
}

void main ( )    {
    Student.Name = ( char * ) malloc ( sizeof ( char ));
    Eingabe ( Student.Name, "Name: ");
    ...
}

```

7. 2. Bitfelder

Unter Bitfeldern versteht man spezielle Strukturkomponenten, deren Werte bitweise nebeneinander in einem Wort abgelegt werden. Auf diese Weise können Informationen sehr speicherplatzsparend untergebracht werden.

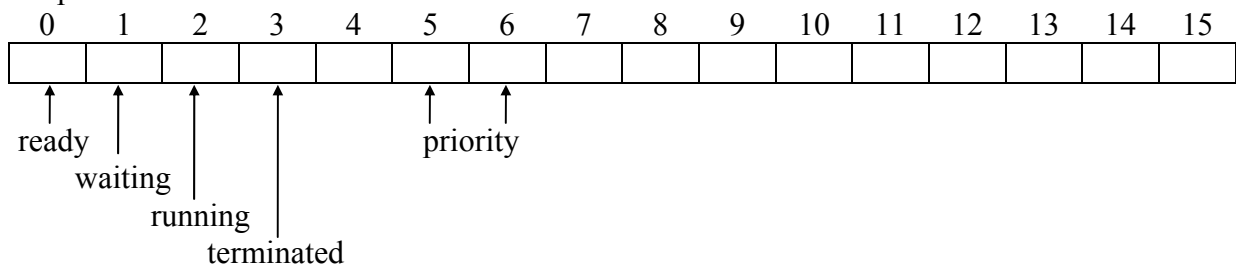
Als Datentyp eines Bitfeldes wird in der Regel nur unsigned oder int zugelassen.

Zur Definition eines Bitfeldes setzt man hinter dessen Namen einen Doppelpunkt, gefolgt von einem konstanten Ausdruck. Dieser gibt die Anzahl der Bits an, in denen die Komponente abgespeichert wird und liegt rechnerabhängig z. B. zwischen 1 und 16.

Beispiel mit 4 Bitfeldern der Länge 1 und einem Bitfeld der Länge 2:

```
struct Zustand {
    unsigned ready : 1;
    unsigned waiting : 1;
    unsigned running : 1;
    unsigned priority : 2;
}
```

Bitpositionen:

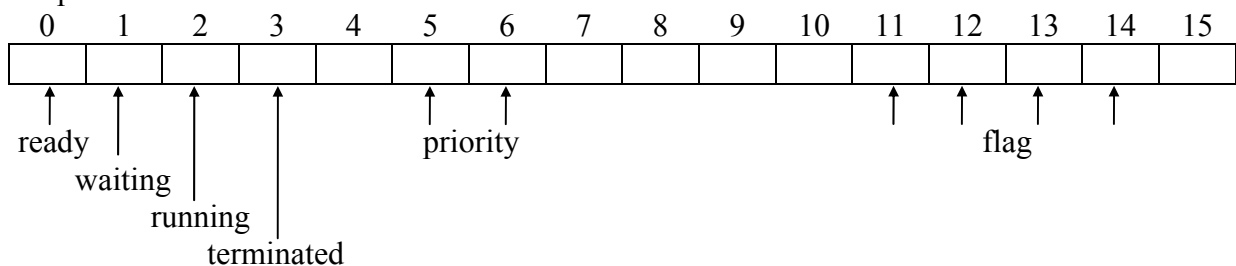


Durch Verwendung unbenannter Bitfelder können auch leere Bitfelder erzeugt und damit die Positionen beliebig festlegen.

Beispiel:

```
struct Zustand {
    unsigned ready : 1;
    unsigned waiting : 1;
    unsigned running : 1;
    unsigned priority : 2;
    unsigned : 4;
    unsigned flag : 4;
}
```

Bitpositionen:



7. 3. Varianten

Eine Variante unterscheidet sich von einer allgemeinen Struktur darin, daß allen Komponenten der gleiche Speicherplatz zugewiesen wird. Er ist gerade so groß, daß er die Komponente mit dem größten Speicherbedarf aufnehmen kann. Diese Speichersparnis hat zur Folge, daß immer nur eine Komponente mit einem Wert belegt werden kann. Weist man einer anderen Komponente einen Wert zu, so wird damit die alte Komponente überschrieben.

Zur Definition einer Varianten braucht man statt des Schlüsselwortes **struct** lediglich das Schlüsselwort **union** zu schreiben.

Beispiel:

```
union Variante    {
    char Zeichen;
    unsigned Zahl;
    int i;
    float x;
} v;
```

In diesem Beispiel habe x den größten Speicherbedarf von 4 Bytes. Dann werden der Variantenvariablen v 4 Bytes zugewiesen. Die Zuweisung v.Zeichen = 'A'; schreibt das Zeichen 'A' in das erste Byte, die restlichen 3 Bytes bleiben leer. Folgt eine weitere Zuweisung v.x = -2.4; so wird -2.4 in die 4 Bytes geschrieben und damit 'A' gelöscht.

7. 4. Initialisierung

Eine Datenstruktur, die sich wie z. B. der Feld- oder Strukturtyp aus einzelnen Komponenten zusammensetzt, wird als **Aggregat** <aggregate> bezeichnet.

Aggregate werden entweder automatisch mit Nullen initialisiert, oder sie können extern durch Zuweisung einer Liste von Konstantenausdrücken initialisiert werden, wenn sie nicht zu den Speicherklassen **auto** oder **register** gehören. Die in geschweiften Klammern eingeschlossenen, durch Kommata getrennten Konstanten werden den Komponenten der angegebenen Reihenfolge entsprechend zugeordnet. Reicht ihre Anzahl nicht aus, so werden die restlichen Komponenten mit Nullen initialisiert.

Beispiel:

```
struct Test      {
    double x, y;
    char z;
    int a[2][3];
};
```

```
struct Test IniBeispiel = { 10, 20, 'D', { 1, 2, 3 } };
```

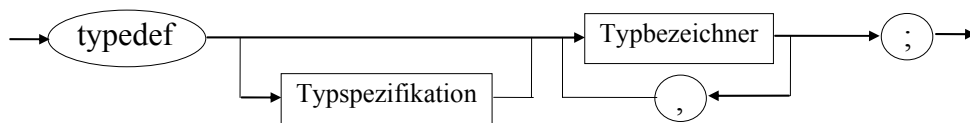
Die Komponenten x, y, z erhalten die Initialwerte 10, 20 und 'D'.

a [0] [0], a [0] [1], a [0] [2] werden mit 1, 2, 3 und die restlichen Feldelemente jeweils mit 0 initialisiert.

7. 5. Eigene Typbezeichner

Die Datentypen struct und union erlauben die Verwendung eigener Bezeichner für einen Datentyp. Z. B. wurden bei der Definition einer komplexen Zahl oder Personaldatei die eigenen Namen tComplex und tPerson benutzt.

Mit Hilfe des Schlüsselwortes typedef lassen sich beliebige Typspezifikationen mit einem eigenen Namen benennen, der anstelle des alten Typbezeichners verwendet werden kann. Dazu schreibt man hinter das Schlüsselwort typedef die Typspezifikation und den eigenen Typbezeichner.



Beispiele:

```
typedef float reell; // definiert das Synonym reell für float, danach darf statt float x, y, z;
                     // reell x, y, z; geschrieben werden.
```

```
typedef struct      {
    double re, im;
} tComplex;
```

```
tComplex Feld [ 10 ]; // definiert ein Feld von 10 komplexen Zahlen
```

```
typedef struct      {
    char *Name;
    char *Vorname;
    ...
} tPerson; // definiert den Strukturtyp tPerson
```

Der Vorteil derartiger eigener Typdefinitionen besteht in der Schaffung

- einprägsamerer Namen,
- von Abkürzungen für umfangreichere Typen (**struct**, Zeiger auf Funktionen)
- leichter wartbarer Programme. Alle Variablen des Typs können bei Bedarf durch die Änderung der Typdefinition angepaßt werden (statt sämtliche Variablendefinitionen aufsuchen zu müssen).

7. 6. Zeiger auf Strukturen

Auf Strukturen sind nur wenige Operationen erlaubt z. B. der Zugriff auf eine Komponente mit dem Punktoperator, der Adreßoperator **&** und die Funktion **sizeof**. Ansonsten darf eine Strukturvariable nicht als Funktionsparameter verwendet werden.

Die Deklaration eines Zeigers auf eine Struktur erfolgt analog wie bei einfachen Variablen durch Anwendung des Dereferenzoperators auf den Strukturnamen z. B. werden mit

```
struct complex    {
    double x, y;
} *z1, *z2, *z3;
```

drei Zeigervariablen z1, z2, z3; auf die Struktur complex deklariert.

Die Speicherverwaltung erfolgt mit Hilfe der Funktionen malloc und free z. B. wird z mit

```
z = ( struct complex *) malloc ( sizeof ( struct complex ));
```

definiert.

Der Zugriff auf die Komponenten einer in Zeigerform deklarierten Struktur kann entweder unter Verwendung der Dereferenz mit dem Punkt in der Form (*z1).x oder kürzer unter Verwendung des Zeigers mit dem Operator -> in der Form z1->x erfolgen. Eine Wertzuweisung z. B. mit einer komplexen Zahl $2 + 3j$ sieht folgendermaßen aus

```
(*z1).x = 2;           oder           z1->x = 2;
(*z1).y = 3;           z1->y = 3;
```

7. 6. 1. Lineare Listen

Als Strukturkomponenten sind in Zeigerstrukturen wie in gewöhnlichen Strukturen alle Datentypen, also auch Felder, weitere Strukturen und Zeiger erlaubt, so daß sich tief geschachtelte, strukturierte Datentypen konstruieren lassen.

Im Gegensatz zu gewöhnlichen Strukturen ist es jedoch bei Zeigerstrukturen erlaubt, als Komponententyp rekursiv den eigenen Strukturtyp zu deklarieren.

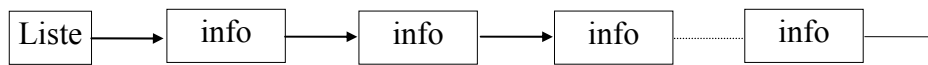
Beispiel: Folgende Deklaration

```
struct tListe      {
    char *info;
    struct tListe *next;
} *Liste;
```

deklariert eine derartige rekursive Struktur tListe, die eine assoziierte Information in Textform und eine Zeigerkomponente next verwendet, die ihrerseits vom Typ * tListe ist.

Beachte: Rekursive Strukturen sind nur mit Zeigerkomponenten erlaubt; eine Deklaration tListe next; wäre falsch!

Zur Definition einer linearen Liste werden fortlaufend Strukturen vom Typ tListe erzeugt und in die entstehende Liste eingefügt. Der letzte next-Zeiger kann mit NULL abgeschlossen werden oder im Falle einer zyklischen Liste auf ein vorhergehendes Element verweisen.



Zur Bearbeitung solcher Listen benutzt man Funktionen für folgende Aufgaben:

1. Erzeugung eines neuen Elementes, das an den Listenanfang, an das Listenende oder im Falle einer geordneten Liste an einer bestimmten Stelle eingefügt wird
2. Löschen des ersten, letzten oder eines bestimmten Elementes
3. Ausgabe der in einzelnen oder allen Elementen enthaltenen Information

Eine lineare Liste wird als Stack oder Kellerspeicher bezeichnet, wenn sie mit Hilfe der Funktionen push, top und pop bearbeitet wird, als Queue oder Schlange bezeichnet, wenn sie die Funktionen append, top und pop verwendet:

1. Die push-Funktion erzeugt ein neues Element, das an den Listenanfang gesetzt wird.
2. Die append-Funktion erzeugt ein neues Element, das an das Listenende angefügt wird.
3. Die top-Funktion gibt die assoziierte Information des ersten Elementes aus.
4. Die pop-Funktion löscht das erste Listenelement.

Durch Auf- und Abbau eines Stacks kann die Information in der Reihenfolge last-in-first-out, kurz LIFO genannt, ein- und ausgegeben werden. Bei einer Queue gilt die Reihenfolge first-in-first-out, kurz FIFO genannt.

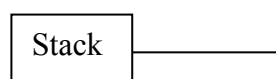
Beispiel zur Implementation eines Stacks:

Das Hauptprogramm **void main ()** erwartet eine Eingabe zur Entscheidung, ob „gepusht, getopt, gepopt“ oder die ganze Liste gelöscht werden soll.

```
// Stackoperationen
```

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
```

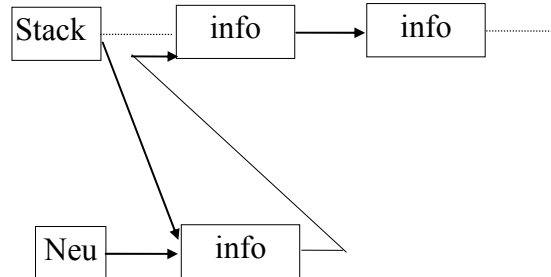
```
struct tStack {
    char *info;
    struct tStack *next;
} *Stack = NULL;
```



```

void push ( )    {
    struct tStack *Neu = ( struct tStack *) malloc ( sizeof ( struct tStack ));
    Neu->info = ( char *) malloc ( sizeof ( char ));
    printf("Bitte Text eingeben: ");
    scanf("%s", Neu->info);
    getchar ( );
    Neu->next = Stack;
    Stack = Neu;
}

```



```

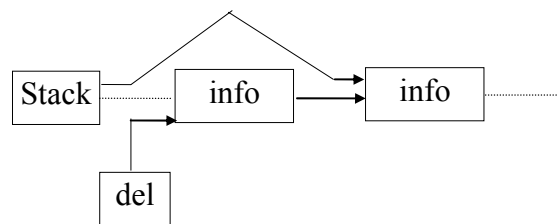
void top ( )    {
    char *Laufzeiger;
    if (!Stack) printf("Der Stack ist leer!\n");
    else        printf("%s\n", *Stack);
}

```

```

void pop ( )    {
    struct tStack *del;
    if (!Stack)
        printf("Der Stack ist leer!\n");
    else {
        del = Stack;
        Stack = Stack->next;
        free ( del->info );    // Beachte:
        free ( del );          // Die beiden Speicherbereiche müssen in der
                                // richtigen Reihenfolge freigegeben werden!
    }
}

```



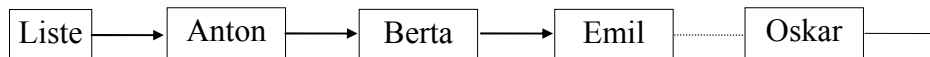
```

void main ( )    {
    char Op, Ende = '1';
    do {
        printf("Stackoperationen: +(push), -(pop), 0(top), sonst Programmende: ");
        Op = getchar ( );
        getchar ( );
        switch (Op) {
            case '+': push ( );
                       break;
            case '0': top ( );
                       break;
            case '-': pop ( );
                       break;
            default: while (Stack) pop( );
                       Ende = '\0';
        }
    } while ( Ende );
}

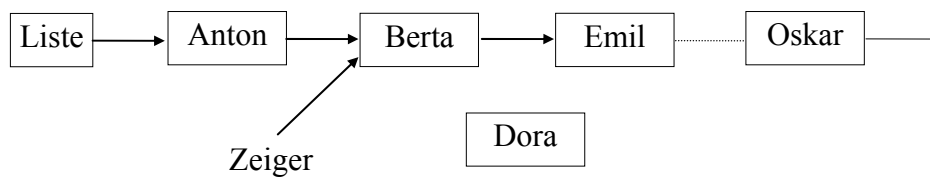
```

Eine lineare Liste heißt geordnet, wenn die Elemente einen Schlüsselwert enthalten, der die Elemente nacheinander auf- oder absteigend numeriert. Der Schlüssel kann entweder eine

zusätzliche int-Komponente oder die alphabetisch geordnete, assoziierte Information selbst sein.



Zum Ein- oder Ausfügen eines Elementes läßt man einen Zeiger die Liste durchlaufen, der genau dann stoppt, wenn er auf das Element zeigt, hinter das ein- oder ausgefügt werden soll. Danach lassen sich die Zeiger dem Ein- oder Ausfügeprozeß entsprechend unter Beibehaltung der Ordnung umweisen.



Um z. B. Dora einzufügen muß ein Zeiger nach Berta geschickt werden. Das folgende Programm enthält die zum Aufbau einer linearen Liste

```

struct tListe {
    char *info;
    struct tListe *weiter;
} *Liste = NULL;
  
```

notwendigen Funktionen:

1. Die rekursive Funktion **int** Vergleich (**char** *Text1, **char** *Text2) testet, ob bei lexikographischer Anordnung Text1 > Text2, Text1 = Text2 oder Text1 < Text2 gilt und liefert als Funktionswert 1, 0 oder -1.
 2. die Zeigerfunktion **struct** tListe *SuchVorg (char *Text) liefert einen Zeiger auf das Element, hinter das ein neues mit dem Inhalt Text eingefügt werden soll. Falls ein solches Element nicht existiert, weil Text kleiner als das 1. Element oder die Liste leer ist, wird der NULL-Zeiger zurückgegeben.
 3. Die Funktion **void** Fuegein () erzeugt ein neues Element und fügt es hinter die mit SuchVorg () ermittelte Stelle in die lineare Liste ein.
 4. In der Funktion **void** Loesche () wird ein Text eingelesen und das Element, das diesen Text enthält, sofern vorhanden, ausfügt.
 5. Die Funktion **int** Ausgabe () schreibt den Inhalt der Liste alphabetisch geordnet aus und liefert als Funktionswert die Listenlänge.
 6. Das Hauptprogramm **void** main () erwartet eine Eingabe zur Entscheidung, ob eingefügt, ausgegeben, ausgefügt oder die ganze Liste gelöscht werden soll.
- ```

int Vergleich (char *Text1, char *Text2) {

```

```

 if (*Text1 > *Text2) return 1;
 else
 if (*Text1 < *Text2) return -1;
 else
 if (!*Text1 && !*Text2) return 0;
 else return Vergleich (++Text1, ++Text2);
}

struct tListe *SuchVorg (char *Text) {
 struct tListe *lauf = Liste;

 if (!Liste) return NULL; // Die Liste ist leer.
 else
 if (Vergleich (Text, lauf->info) <= 0) return NULL; // Text vor 1. El.
 else
 while (lauf->weiter && Vergleich (Text,lauf->weiter->info > 0)
 lauf = lauf->weiter;

 return lauf;
}

void Fuegein () {
 struct tListe *Neu = (struct tListe *) malloc (sizeof (struct tListe)), *Pos;

 Neu->info = (char *) malloc (sizeof (char));
 printf("Bitte Text eingeben: ");
 scanf("%s", Neu->info);
 getchar ();
 Neu->weiter = NULL;
 Pos = SuchVorg (Neu->info);
 if (!Pos) { // Neues Element wird 1. Element
 Neu->weiter = Liste;
 Liste = Neu;
 }
 else { // Neues Element wird hinter dem 1. eingefügt
 Neu->weiter = Pos->weiter;
 Pos = Neu;
 }
}

void Loesche () {
 struct tListe *Pos, *del;
 char *Text = (char *) malloc (sizeof (char));

 if (!Liste) printf ("Die Liste ist leer!\n");
 else {
 printf ("Bitte auszufuegenden Text eingeben: ");
 scanf ("%s", Text);
 getchar ();

 Pos = SuchVorg (Text);
 }
}

```

```

 if (!Pos || !Pos->weiter) // der Text ist im 1. Element oder nicht in der Lis-
te
 if (!Vergleich (Liste->info, Text)) { // Text im 1. Element
 del = Liste;
 Liste = Liste->weiter; // wird geloescht
 free (del);
 }
 else printf ("Kein Element mit diesem Text vorhanden!\n");
 else { // Text steht hinter dem 1. Element
 del = Pos->weiter;
 Pos->weiter = del->weiter; // wird geloescht
 free (del);
 }
 }
}

int Ausgabe () { // gibt alle Texte geordnet aus und liefert die Laenge der Liste
 struct tListe *lauf = Liste;
 int i = 0; // Zaehler
 while (lauf) {
 printf ("%s\n", lauf->info);
 lauf = lauf->weiter;
 i++;
 }
 return i;
}

void main () {
 char Op, Ende = '1';
 do {
 printf("Listenoperationen: +(Einfuegen), 0(Ausgabe), -(Loeschen), sonst Ende:
");
 Op = getchar ();
 getchar ();
 switch (Op) {
 case '+': Fuegein ();
 break;
 case '0': printf ("Laenge der Liste: %d\n \n", Ausgabe ());
 break;
 case '-': Loesche ();
 break;
 default: while (Liste) {
 del = Liste;
 Liste = Liste->weiter;
 free (del)
 Ende = '\0';
 }
 }
 } while (Ende);
}

```

## 7. 6. 2. Binäre Bäume

Durch Verwendung mehrerer Folgezeiger lassen sich beliebige, gerichtete Graphen konstruieren, deren Operationen besonders einfach durch rekursive Programme ausgeführt werden können.

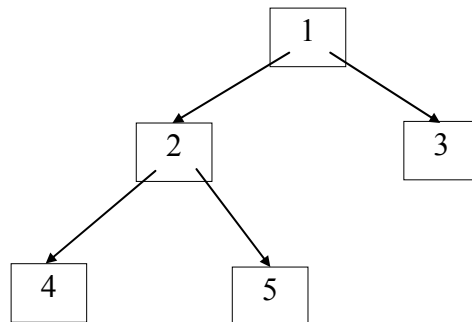
Eine elementares Anwendungsbeispiel ist der Baum. Das ist ein einfach zusammenhängender, zyklensfreier Graph, dessen Knoten alle von einem sogenannten *Wurzelknoten* auf einem eindeutig bestimmten Weg erreichbar sind.

Bis auf die Wurzel hat jeder Knoten einen Vorgänger. Knoten ohne Nachfolger heißen *Blätter* des Baumes.

Man bezeichnet die Anzahl der von einem Knoten ausgehenden Kanten als Grad des Knotens und den maximal auftretenden Grad als Grad des Baumes.

Ein Baum vom Grade 2 heißt *binärer Baum* oder *Binärbaum*

Die Länge des Weges von der Wurzel zu einem Knoten ist die Anzahl der auf dem Wege liegenden Kanten und heißt *Tiefe* des Knotens. Die Tiefe des längst möglichen Weges in einem Baum heißt *Höhe* des Baumes. Dieser führt offensichtlich von der Wurzel zu einem Blatt.



Beispiel: Baum in schichtenweise, sequentieller Anordnung

Ein Binärbaum der Höhe  $h$  besitzt mindestens 1 und höchstens  $2^h$  Blätter und mindestens  $h+1$  und höchstens  $2^{h+1} - 1$  Knoten. Im ersten Fall liegt ein zu einer Liste entarteter Baum vor, im zweiten Fall besitzt der Baum eine optimale Struktur mit schichtenweiser, sequentieller Anordnung, bei der noch jede Schicht ihre maximale Knotenzahl aufweist.

Umgekehrt gilt

$$h \geq \text{ld} (\text{Knotenzahl} + 1) + 1$$

Der Vorteil bei der Verwendung binärer Bäume gegenüber linearer Listen besteht offensichtlich darin, dass Aufwand zum Aufsuchen eines Knotens lediglich wie die Logarithmusfunktion statt linear mit steigender Knotenzahl wächst.

In folgendem Beispiel zum Auf- und Abbau eines Binärbaumes in schichtenweiser, sequentieller Anordnung formuliert. Die Verwendung rekursiver Funktionen führt zu besonders einfachen Funktionsformulierungen.

```

// Beispiel7_3: Binaerer Baum
//
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

struct tBaum
{
 int key; // Sortierschluessel
 char *info; // assoziierte Information
 struct tBaum *Nachf[2];
} *Wurzel = NULL;

struct tBaum *GenKnoten(int Nr) { // Erzeugung eines Knotens
 struct tBaum *Knoten = (struct tBaum*)malloc(sizeof(struct tBaum));
 char *Text = (char*)malloc(sizeof(char));

 printf("%d. Text: ", Nr);
 scanf("%s", Text);
 getchar();

 Knoten->key = Nr;
 Knoten->info = Text;
 Knoten->Nachf[0] = Knoten->Nachf[1] = NULL;
 return Knoten;
}

struct tBaum *Vorg (int i) { // Vorgaenger eines Knotens
 if (i<4) return Wurzel;
 else
 return Vorg (i/2)->Nachf[i%2];
}

void Fuegein(int Nr) {
 if (Nr==1) Wurzel = GenKnoten (1);
 else
 Vorg (Nr)->Nachf[Nr%2] = GenKnoten (Nr);
}

void Loesch(int Nr) {
 if (Nr==1) {
 free(Wurzel);
 Wurzel = NULL;
 }
 else {
 free(Vorg (Nr)->Nachf[Nr%2]);
 Vorg (Nr)->Nachf[Nr%2] = NULL;
 }
}

void preorder(struct tBaum *Baum) {
 if (Baum) {
 printf("Nr.%d %s\n", Baum->key, Baum->info);
 preorder(Baum->Nachf[0]);
 preorder(Baum->Nachf[1]);
 }
}

void inorder(struct tBaum *Baum) {
 if (Baum) {
 inorder(Baum->Nachf[0]);
 printf("Nr.%d %s\n", Baum->key, Baum->info);
 inorder(Baum->Nachf[1]);
 }
}

```

```

void postorder(struct tBaum *Baum) {
 if (Baum) {
 postorder(Baum->Nachf[0]);
 postorder(Baum->Nachf[1]);
 printf("Nr.%d %s\n", Baum->key, Baum->info);
 }
}

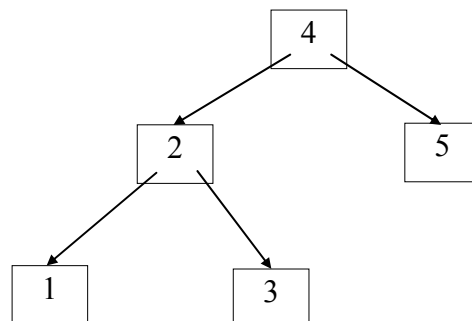
void main() {
 char Op, Ende = '1';
 int Nr = 1;

 do {
 printf("Baumoperationen: +(Einfuegen), -(Ausfuegen),\n");
 printf("1 (Ausgabe pre-), 2 (in-), 3 (postorder), sonst Ende: ");
 Op = getchar();
 getchar();

 switch (Op) {
 case '+': Fuegein(Nr++);
 break;
 case '-': Loesch(--Nr);
 break;
 case '1': preorder(Wurzel);
 break;
 case '2': inorder(Wurzel);
 break;
 case '3': postorder(Wurzel);
 break;
 default: Ende = '\0';
 } while (Ende);
 }
}

```

Bei einem geordneten, binären Baum, einem sogenannten Suchbaum, ist der Schlüsselwert eines jeden Knotens größer als der seines linken und kleiner als der seines rechten Sohnes. In diesem Fall lassen sich die Knoten auf besonders kurzen Wegen erreichen, insbesondere dann, wenn der Baum austariert ist, d. h. eine bei vorgegebener Knotenzahl minimale Höhe hat ( sog. AVL-Bäume ).



Beispiel: Suchbaum

Übung 7\_1: Eine Queue ( Schlange ) unterscheidet sich von einem Stack darin, daß statt der Operation push ( ) die Operation append ( ) verwendet wird, die ein neues Element nicht an den Anfang, sondern an das Ende der linearen Liste anfügt. Dazu wird ein zusätzlicher Zeiger verwendet, der auf das letzte Listenelement zeigt und nach jedem Anfügevorgang eine Position weiter gesetzt wird. Durch eine top-pop-Folge lässt sich die Information in der Reihenfolge FIFO ( first in first out ) wieder ausgegeben.

Ändern Sie das Stackprogramm in ein Queueprogramm um, indem Sie

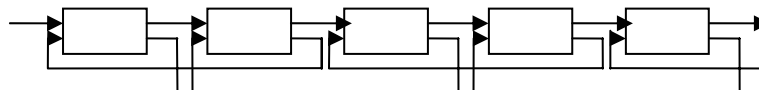
- den Typbezeichner tStack durch tQueue und den Zeiger Stack durch Head ersetzen,
- einen mit NULL initialisierten Zeiger Tail hinzufügen, der auf das letzte Element zeigt,
- die Funktion push durch die Anfügefunktion append ersetzen.

Übung7\_2: In dieser Aufgabe soll ein Programm mit Standardfunktionen zur Bearbeitung einer zweifach verketteten, geordneten, linearen Liste geschrieben werden, deren Elemente zwei Zeiger mit den Namen \*vor und \*rueck enthalten.

```
struct tListe {
 char *info;
 struct tListe *vor, *rueck;
} *Liste = NULL;
```

Die Zeiger sollen so angeordnet sein, dass die \*vor-Zeiger die Liste wie bei der behandelten einfachen Verkettung in Richtung lexikographisch steigenden Textinhaltes durchlaufen.

Dagegen sollen die \*rueck-Zeiger die Liste in entgegengesetzter Richtung, also lexikographisch fallenden Textinhaltes durchlaufen.



Schreiben Sie zwei Ausgabefunktionen, deren eine einen Vorwärtslauf ( Tastatureingabe 0 ) und deren andere einen Rückwärtslauf ( Tastatureingabe 1 ) um einen Zyklus bewirkt, sodass die Texte einmal in steigender und einmal in fallender lexikographischer Anordnung ausgegeben werden können.



Übung 7\_3: In dieser Aufgabe soll eine lineare, geordnete, einfach verkettete Liste aufgebaut werden, deren Elemente jeweils ein deutsches Wort und dessen englische und französische Übersetzung enthalten.

Ferner gehen von jedem Listenelement drei Zeiger aus, die jeweils auf das Listenelement zeigen, welches das hinsichtlich der lexikographischen Anordnung folgende deutsche, englische bzw. französische Wort enthält.

Auf die Listenelemente mit dem ersten deutschen, dem ersten englischen und dem ersten französischen Wort zeigt je ein Anfangszeiger, der als Startzeiger zum Durchlauf der Liste nach einer der drei Sprachen dient.

Benutzen Sie folgende Datenstruktur mit vordefiniertem  $N = 3$ :

```
struct tListe {
 char *info[N];
 struct tListe *weiter[N];
} *Liste[N];
```

Das Programm soll durch die Eingabe eines Zeichens folgendermaßen gesteuert werden:

+-Zeichen ruft die Funktion Fuegein auf, die ein neues Listenelement erzeugt, zur Eingabe von  $N-1$  Vokabeln auffordert, und dieses in die Liste einfügt,

eine Ziffer zwischen 0 und  $N-1$  gibt die Tabelle nach der Sprache deutsch (0), englisch (1) oder französisch (2) sortiert aus,

ein sonstiges Zeichen löscht die Liste und gibt die Speicherplätze frei.

Beispiel:

|          |            |               |
|----------|------------|---------------|
| Amboss   | anvil      | enclume       |
| Abfuhr   | removal    | enlevement    |
| Ahnung   | suspicion  | pressentiment |
| Adel     | nobility   | noblesse      |
| Affe     | monkey     | singe         |
| Aas      | bait       | charogne      |
| Agraffe  | clasp      | agrafe        |
| Aktie    | share      | action        |
| Achtung  | attention  | attention     |
| Allotria | tomfoolery | frasques      |

Zur weiteren Verwendung kann die Liste um Vokabeln weiterer Sprachen ergänzt, in einer Datei gespeichert und diese fortlaufend ergänzt werden. Ferner lassen sich statt Vokabeln auch beliebige Objekte mit beliebigen Attributen verwenden, die sich als Sortierschlüssel verwenden lassen.

#### Übung 7\_4: Formulieren Sie Funktionen zum Auf- und Abbau eines Suchbaumes folgender Struktur

```
struct tBaum {
 char *info;
 struct tBaum *Nachf[2];
}
```

Das Hauptprogramm ruft nach Eingabe eines Pluszeichens die Funktion Fuegein zur Erzeugung und Einfügung eines Knotens statt eines Blattes auf, wobei die Suchbaumeigenschaft erhalten bleibt.

Entsprechend führt die Eingabe eines Minuszeichens zum Löschen eines durch den Eingabetext bestimmten Knotens.

Zur Ausgabe sollen die in letztem Beispiel verwendeten rekursiven Funktionen verwendet werden:

**void** preorder ( **struct** tBinBaum \*Baum ) werde nach Eingabe von 1 gestartet, gibt zuerst den Text des Wurzelknotens aus, ruft sich dann für den linken und zuletzt für den rechten Nachfolger des Wurzelknotens auf,

**int** inorder ( **struct** tBinBaum \*Baum ) werde nach Eingabe von 2 gestartet, ruft sich rekursiv für den linken Nachfolger auf, gibt dann den Text des Wurzelknotens aus und ruft sich dann rekursiv für den rechten Nachfolger auf,

**int** postorder ( **struct** tBinBaum \*Baum ) werde nach Eingabe von 3 gestartet, ruft zuerst rekursiv den linken, dann den rechten Nachfolger auf und gibt zuletzt den Wert des Wurzelknotens aus.

Zusätzlich soll nach Eingabe von 0 die mit der Funktion

**int** Knotenzahl ( **struct** tBinBaum \*Baum )

bestimmte Anzahl der Knoten ausgegeben werden.