C / C++ - Einführung

WS 2002/2003

Michael Baum
Dipl.-Math.
E-Mail: micbaum@de.ibm.com

1.) Einführung in die C – Programmierung

1.1 Einige grundsätzliche Begriffsbestimmungen (ref. Beispiel 1) Seite 3
1.2 Definition von Variablen und Konstanten 7

2.) Programmverzweigungen

2.1 Die if - Anweisung (ref. Beispiel 2) 11
2.2 Vergleichsausdrücke 12
2.3 Die switch - Anweisung (ref. Beispiel 3) 14
2.4 Die goto – Anweisung 16

3.) Programmschleifen

3.1 Die while - Schleife (ref. Beispiel 4) 16
3.2 Die for - Schleife (ref. Beispiel 4) 18
3.3 Die do-while - Schleife (ref. Beispiel 5) 19
3.4 Einfache Schleifenprogramme für Ein - / Ausgabe, getchar( ) und putchar(c) 20

4.) Die Entwicklungsumgebung (Einführung)

4.1 Einige grundsätzliche Möglichkeiten in der Entwicklungsumgebung 22
4.2 Spezielle Editierbefehle : Edit - und Search – Menue 23
4.3 Austesten von Programmen : Debug – Menue 23

5.) Ausdrücke und Anweisungen

5.1 Einfache und mehrfache Wertzuweisungen, Kommaoperator (ref. Beispiel 6) 23
5.2 Arithmetische Operationen 26
5.3 Bitoperationen 29
5.4 Typkonvertierung in Ausdrücken 29
5.5 Aufzählungskonstanten mit enum 31

6.) Eingabe von Konsole und Ausgabe am Bildschirm

6.1 Eingabe und Ausgabe mit cin, cout (ref. Beispiel 7) 32
6.2 Formatierte Eingabe und Ausgabe mit printf, scanf(ref. Beispiel 8) 34

7.) Felder

7.1 Eindimensionale Felder (ref. Beispiele 9, 10) 37
7.2 Mehrdimensionale Felder (ref. Beispiel 11) 40
7.3 Zeichenvektoren, die Funktionen gets(), puts() (ref. Beispiel 12) 41

8.) Strukturen

8.1 Einfache Strukturen (ref. Beispiele 13 , 14) Seite 43
8.2 Vektoren von Strukturen 46
8.3 Unions 47
8.4 Bitfelder 48
8.5 typedef structure (ref. Beispiel 16) 49

9.) Zeiger

9.1 Adressierung von Daten über Zeiger 51
9.2 Zeiger und Vektoren, Adressarithmetik 53
9.3 Char-Zeiger, Zeichenvektoren 55
9.4 Mehrdimensionale Vektoren, Vektoren von Zeigern, Zeiger auf Zeiger (ref. Beispiele 17, 18, 19, 1A) 57
9.5 Dynamische Speicherverwaltung (ref. Beispiele 20, 21, 22, 23) 59
9.6 Zeiger auf Strukturen, verkettete Listen (ref. Beispiele 24, 25, 26) 61

10.) Funktionen

10.1 Funktionsdefinition, Werteparameter und Variablenparameter (ref. Beispiele 27, 28) 63
10.2 Funktionsprototyp, Deklaration versus Definition von Funktionen 67
10.3 Speicherklassen auto, static, register, Gültigkeitsbereiche für Namen 69
10.4 Zeiger auf Funktionen, Funktionsnamen als Parameter (ref. Beispiele 29, 30) 73
10.5 Felder, Zeiger und Strukturen als Parameter (ref. Beispiele 31, 32) 74
10.6 Rekursive Funktionen 76
10.7 Inline Funktionen, Makros 77
10.8 Grafische Anwendungen 78

11.) Unterprogrammtechnik, Programmstrukturen

11.1 Include Dateien, eigene Headerdateien 81
11.2 Projekte aus mehreren Quelldateien in Borland C++ 82
11.3 Strukturierte Programmierung 82

12.) Eingabe und Ausgabe von Daten

12.1 Datei-Funktionen mit stdio.h (ref. Beispiel 33) 84
12.2 Dateifunktionen mit fstream.h 87
12.3 Zusammenfassung der Ein- und Ausgabefunktionen für C++ 90

1.) Einführung in die C - Programmierung

Das Ziel von Kapitel 1, 2 und 3 ist, möglichst schnell den Punkt zu erreichen, wo nützliche Programme geschrieben werden können; Vollständigkeit und Genauigkeit folgen in den späteren Kapiteln.

1.1 Einige grundsätzliche Begriffsbestimmungen (ref. Beispiel 1)

Die Programmiersprache C wurde in den 70-er Jahren in USA von den Autoren Brian W. Kernighan und Dennis M. Ritchie entwickelt.
C ist eine Programmiersprache für allgemeine Anwendungen. Dabei ist C eine relativ 'maschinennahe' Sprache : C geht mit den gleichen Objekten um wie der Computer : Zeichen, Zahlen und Adressen. Trotzdem ist C unabhängig von einer speziellen Maschinen-Architektur.
Der sogenannte **ANSI-Standard (American National Standards Institute)** wurde 1988 festgelegt. Ein wesentlicher Beitrag des Standards ist die Definition einer Bibliothek, welche zu C gehört.

C++ wurde in den 80-er Jahren von **Bjarne Stroustrup** entwickelt, teilweise **als verbessertes C**, zum wesentlichen Teil aber auch als Sprache für Datenabstraktion und objektorientiertes Programmieren. Die nachfolgenden Kapitel berücksichtigen die C-Erweiterungen bezw. die C - Verbesserungen in C++ , soweit sie nicht direkt mit objektorientiertem Programmieren zu tun haben.

In einem C - Quellenprogramm kann ein **Kommentar** überall dort stehen, wo auch ein Leerzeichen stehen kannn. Ein Kommentar beginnt mit /* und endet mit */ , er kann sich dabei auch über mehrere Zeilen erstrecken:

```
/*  Dieser Kommentar  gilt für eine Zeile     */
/*  Dieser Kommentar
        gilt für zwei Zeilen                  */
```

```
/*
 *   Auch dieser Kommentar
 *    ist recht gut lesbar.
 */
```

In C++ sind auch Kommentare hinter dem Zeichen // bis zum Zeilenende möglich :

```
// Dies  ist ein  C++ Kommentar
```

Der Kommentartext wird vom Compiler ignoriert. Jeder Programmbeginn und jeder neue Programmabschnitt sollte mit erklärendem Kommentar versehen sein. Nicht jede Anwendungszeile benötigt einen Kommentar.
Variablen repräsentieren Speicherstellen, deren Inhalt durch Wertzuweisungen oder Einlesen von Daten bestimmt wird. Für jede Variable wird vor ihrer ersten Anwendung ein **Datentyp** vereinbart, wobei bei dieser Definition auch ein Anfangswert festgelegt werden kann :

```
int anfangswert,  endwert;        //    zwei ganzzahlige Variablen.
int  _ziel = 100;                 //    ganzzahlige Variable mit Anfangswert
float inhalt, laenge = 20;        //  zwei reelle Variablen  (Gleitkomma Variablen )
char  zeichen, grbuchstabe =  'H' ,
                  klbuchstabe  =  'e';        //     char - Variablen
```

Das erste Zeichen eines Namens muß ein Buchstabe oder ein Unterstrich sein, die weiteren Zeichen können Buchstaben oder Ziffern sein, oder Unterstrich. Bis zu 32 Zeichen (ohne Leerzeichen) sind für die Namensgebung von Variablen relevant.
Große Buchstaben werden dabei von kleinen Buchstaben unterschieden : Ziel, ziel, ZIEL sind drei verschiedene Namen in C. Üblicherweise werden im C - Programm normale Variablennnamen klein geschrieben.
Eine Variable kann nur Werte vom vorgegebenen Datentyp annehmen. Aufgrund solcher Vereinbarungen kann der Compiler für jede Variable Speicherplatz reservieren und der Datendarstellung entsprechend Maschinenbefehle erzeugen. Variablen sollten vom Namen her ihre Bedeutung leicht erkennen lassen, z. B. zins, inhalt, flaeche; einfache Buchstaben wie i,j,r empfehlen sich bei mathem. Aufzählungen bezw. als besondere Größen in mathem. Formeln.
Im allgemeinen ist für jede neue Namensdefinition im Programm ein kurzer und erklärender Kommentar sinnvoll und nützlich.
Vereinbarungen für Namen stehen normalerweise am Anfang einer Funktion (wie main()..) oder zu Beginn eines **Programm-Blocks**, d. h. einer durch { } eingeklammerten Anweisungsfolge.
In C gibt es **reservierte Namen**, wie cin , cout, main, und auch **vordefinierte Namen**, wie sin, sqrt : **alle solche Namen sind stets klein zu schreiben** .
Konstanten sind Werte, welche im Programm unmittelbar angegeben werden. Man unterscheidet :

Integer-Konstanten : 5, 78
Gleitkomma-Konstanten : 5.0 , 78.123
Zeichen - Konstanten (Character - Konstante) : 'a' , 'A'
Textkonstanten (Stringkonstanten) : "Programm"

In einem Programm können solche Konstanten als **Literale** (direkte konstante Größe wie 78) oder auch als symbolische Konstante auftreten : dann ist ein konstanter Wert mit einem symbolischen Namen verbunden ; der Symbolwert kann im Programmablauf nicht verändert werden.
Einzelne **Zeichenkonstanten** bestehen aus einem (oder mehreren) Zeichen zwischen den Begrenzungszeichen Apostroph : 'm'

```
'M'
'\xHexadezimalzahl'
'\n'                // Escape - Sequenz für neue Zeile
```

Die Zeichenkonstante 'm' ist ein ganzzahliger Wert und stellt den numerischen Wert des Zeichens m im Zeichensatz der Maschine dar.
Die Hexadezimalzahlen liegen im Bereich x00 bis x1F, z.B. '\x41' (Buchstabe A)
Für nicht darstellbare Zeichen bietet C eine besonder Symbolik an : die Escape - Sequenzen.
Für mögliche spezielle Steuerzeichen (Escape - Sequenzen) gilt die folgende Tabelle :

'\a'	BEL	Tonerzeugung	0x07
'\b'	BS	Backspace=Rücktaste	0x08
'\f'	FF	Form Feed=Vorschub	0x0C
'\n'	LF	Linefeed=Zeilenvorschub	0x0A
'\r'	CR	Carriage Return	0x0D
'\t'	HT	Tab horizontal=Tab Taste)	0x09
'\v'	VT	Tab vertical	0x0B
'\\'	\	Backslash = Rückstrich	0x5C
'\''	'	Apostroph = Hochkomma	0x27
'\"'	"	Anführungszeichen	0x22
'\?'	?	Fragezeichen	0x3F

Dazu kommt noch das Zeichen '\0' (0x00) als Ende von intern gespeicherten Zeichenfolgen (Strings).

Textkonstanten (auch strings genannt)sind Zeichenfolgen zwischen den Begrenzungszeichen Anführunszeichen :

" Dies ist eine Textkonstante "

"\nDiese Textkonstante beginnt bei der Bildschirmausgabe in einer neuen Zeile "

Beim Abspeichern im Rechner wird an das Ende einer solchen Zeichenkette ein Nullzeichen \0' noch vom System automatisch angehängt. Die Textkonstante "x" besteht also abgespeichert aus zwei Zeichen .

Der **Dateiname** für ein C - Quellenprogramm richtet sich nach dem Betriebssystem ! Für DOS kann ein Dateiname maximal 8 Zeichen (Buchstaben oder Zahlen oder auch spezielle Sonderzeichen) enthalten, mit zusätzlicher Namenserweiterung . Zwischen Groß - und Kleinschreibung wird bei DOS - Dateinamen nicht unterschieden :

Für C++ : Aufg1.CPP
aufg1.cpp gleiche Datei wie Aufg1.CPP
1Aufg.CPP

Für C : Aufg1.C

Einige der in DOS - Dateinamen erlaubten Sonderzeichen sind : _ , $, # , & .

Unter Windows gelten die für lange Namen gültigen Regeln !

Ein einfaches **C - Programm** besteht aus einem **globalen Vereinbarungsteil** und der nachfolgenden **Hauptfunktion main().** Das runde Klammernpaar ( ) kennzeichnet hier eine leere Parameter- liste der Funktion main(). Im gleichen C - Programm darf der Funktionsname main() nicht mehrmals auftreten.

Im globalen Vereinbarungsteil stehen u. a. die **Preprozessor - Anweisungen**, wie z.B.

```
#include <iostream.h>
```

Solche Preprozessoranweisungen fügen sog. Headerdateien mit Informationen über Unterprogrammbibliotheken ein und besorgen dadurch die Zuordnung von solchen Unterprogrammbiliotheken zum Programm; sie werden zeilenweise angeordnet und nicht durch ein Semikolon abgeschlossen. Preprozessoranweisungen können überall im Programm stehen und irgendwo in einer Zeile beginnen (Borland C, C++) . Im globalen Vereinbarungsteil können auch **globale Vereinbarungen** stehen; darüber später mehr .

Die Hauptfunktion main() enthält die **lokalen Vereinbarungen** und **Anweisungen,** die festlegen, welche Aktionen nacheinander ausgeführt werden sollen. Jede lokale Vereinbarung und jede Anweisung wird in C mit ; abgeschlossen. Lokale Vereinbarungen und Anweisungen sind zwischen den Klammersymbolen { und }. Die geschweiften Klammern { und } dienen in C allgemein dazu, Vereinbarungen und Anweisungen in einem Block zusammenzufassen. Solchen Blöcken mit geschweiften Klammern ('compound statements ') werden wir noch an verschiedenen Programmstellen begegnen.

Ein allgemeines C - Programm enthält zusätzlich zu main() noch weitere Funktionen , welche innerhalb von main als **Unterprogramme** aufgerufen werden ; ein Beispiel dafür wäre die formatierte Ausgabe von verschiedenen Ergebnissen..

Jeder Programmstart beginnt in einem C - Programm mit der Funktion main() .

In viele Programmen werden **Formeln** berechnet. Dazu gibt es die **Wertzuweisungen als** besondere Anweisung :

```
z = x + y ;
```

```
z = z + 5;
```

Der Wertzuweisungoperator = kennzeichnet keine mathem. Gleichung, sondern eine Zuweisungstätigkeit. In der Wertzuweisung z = z + 5; wird zunächst der Wert von z um 5 erhöht und das Ergebnis anschließend in der Speicherstelle z abgespeichert.
Für eine übersichtliche Programmstruktur sollten bei solchen Ausdrücken die Operatoren immer mit Leerstellen umgeben werden, also nicht z=x+y; , sondern besser z = x + y ; .
Der Ausdruck

```
cout << "\nBitte, zwei ganze Zahlen ";
```

ist eine Anweisung für die Ausgabe am Bildschirm. Das << Zeichen kennzeichnet hier die Ausgabeoperation 'übertrage nach ' , cout kennzeichnet den Bildschirm.
Durch Wiederholung von << können auch mehrere Daten ausgegeben werden :

```
cout << x << "+" << y << "ergibt " << z;
```

Für die Werte x= 5 und y = 3 ergibt sich hier am Bildschirm :

```
5 + 3 ergibt 8
```

Der Ausdruck

```
cin >> x ;
```

ist entsprechenderweise eine Anweisung für Dateneingabe von der Konsole. Das >> Zeichen kennzeichnet hier die Eingabeoperation 'lese von ', cin kennzeichnet die Konsole.
Durch Wiederholung von >> sind mehrere Eingaben in einer Anweisung möglich :

```
cin >> x >> y;
```

An der Konsole muß die Dateneingabe mit cr abgeschlossen werden. Gegebenenfalls können mehrere Dateneingaben nacheinander an der Konsole, durch Leerstellen getrennt , erfolgen.

Im C - Programm werden **Leerstellen, leere Zeilen und Tabulatorstopps** vom Compiler ignoriert, sie zählen als sog. **whitespace - characters.**

Die Anweisung return 0 ; innerhalb der Hauptfunktion main()bewirkt einen Rücksprung in die Entwicklungsumgebung oder in das Betriebssystem.
Während das Format des Quellcodes für den Compiler unwichtig ist, sollte für den menschlichen Leser jeder Quellcode gut verständlich strukturiert bezw. formatiert getaltet sein. Wesentliche Aspekte der Programmstrukturierung sind Einrücken bei neuen Zeilen, Leerstellen, Positionieren von Anweisungen (gleiche / neue Zeile), Kommentare, Verwenden von großen / kleinen Buchstaben bei Namen.
Der Wert des Quellcodes ergibt sich ganz wesentlich aus seiner Lesbarkeit. Auch sollte jedes C-Programm bezüglich Code und Kommentar in einem einheitlichen Stil erstellt werden. Nur so lassen sich Programmfehler eher vermeiden und auch leichter finden **(Debugging)**.
Ein sinnvolles Beispiel von Programmstruktur ist :

* geschweifte Klammern für main() .. zu Beginn einer neuen Zeile,
* interne Definitionen und Anweisungen um eine Tabulatorstelle oder z.B. um 4 Stellen eingerückt.
* Alle Programmzeilen sollten vor dem rechten Bildschirmrand abschließen.

Beispiel :

```
#include <iostream.h>
main()
{
        float summe;
        summe = 1.6 + 3.48;
       cout << "Die berechnete Summe ist : " << summe ;
```

```
    return 0 ;
}
```

Mit Hilfe des **Editors** wird ein C - Programm in den Rechner eingegeben, und kann anschließend als Datei mit der Namenserweiterung .C (bei C) bezw .CPP (bei C++) auf Diskette oder Festplatte abgespeichert werden.

Von der **Editor - Entwicklungsumgebung** aus kann auch der **Compiler** gestartet werden. Der Compiler übersetzt das C- Programm im Hauptspeicher in ein sog. **Objektprogramm** für den Rechner (**Namenserweiterung .OBJ)**.
Der **Linker** schließlich fügt mehrere getrennt übersetzte Objektprogramme zu einem endgültigen ausführbaren **Ladeprogramm** zusammen (**Namenserweiterung .EXE).** Beispiele für solche getrennt übersetzten Programme sind die **Bibliotheksroutinen**.

Das gelinkte Programm kann dann ausgeführt werden. Auch hierfür ist in der Entwicklungs-umgebung ein spezieller Menuebefehl vorgesehen : 'Run' .

1.2 Definition von Variablen und Konstanten

In C müssen alle Namen vereinbart sein, bevor sie im Programm benutzt werden.
Bei Variablendefinitionen muß immer der Datentyp vor dem Variablennamen stehen.

```
int   summe;              // Anfangswert nicht festgelegt
int   i = 15,  j = 48;    //  Anfangswert explizit festgelegt
int  k = i;     //  Die Initialisierung kann sich auch auf bereits definierte Variablen
                //  beziehen.
```

In C müssen innerhalb eines Programmblocks alle Vereinbarungen zu Beginn dieses Blocks stehen. In C++ ist eine solche Anordnung nicht zwingend, trotzdem fasst man normalerweise die Vereinbarungen auch in C++ am Anfang eines Blocks zusammen.

Eine solche Variablendefinition kann in C / C++ innerhalb einer Funktion (z.B. main()), also **lokal** vorkommen, oder auch außerhalb jeder Funktion : dann gilt die Vereinbarung als **global.**
Global vereinbarte Variablen werden automatisch vor Programmbeginn auf Null initialisiert , sie liegen im **statischen Speicherbereich ;** lokal vereinbarte Variablen müssen in der Definition explizit initialisiert werden, sonst ist ihr Anfangswert bei Beginn der Funktion undefiniert, sie liegen im **automatischen Speicherbereich.**
Mit Hilfe des Speicherklassenspezifizierers **static** ist es jedoch möglich, auch lokal vereinbarte Variablen statisch zu machen :

```
static int  yy;    //   lokal statisch, mit 0 vorbesetzt.
```

Alle Variablen vom Typ static werden, falls nicht explizit im Programm mit Werten initialisiert, automatisch mit Null zu Programmbeginn vorbesetzt; sie liegen dann im statischen Bereich.
Allgemein gilt noch : innerhalb eines Programmblocks deklarierte Variablen können nur innerhalb dieses Blocks referiert werden, sie sind außerhalb des Blocks nicht bekannt. Der Block begrenzt ihren Gültigkeitsbereich.
Die ganzzahligen Datentypen werden häufig auch für einfache Zählaufgaben verwendet.
Rechnerintern werden ganze Zahlen als Binärzahlen dargestellt. Bei negativen ganzen Zahlen ist dabei das höchstwertigste Bit immer eine 1. N-stellige Binärzahlen haben einen Wertebereich :

n n

von -2 bis 2 - 1

In der C - Sprache sind die folgenden ganzzahligen Datentypen vordefiniert :

Datentyp	**interne Länge**	**dezimaler Zahlenbereich**	**Anwendung**
int , short int	16 bit	-32768 .. +32767	ganze Zahl
unsigned int	16 bit	0 .. 65535	ganze positive Zahl
long , long int	32 bit	-2 147 483 648 .. +2 147 483 647	ganze Zahl
unsigned long	32 bit	0 .. 4 294 967 295	ganze positive Zahl
char, signed char	8 bit	-128 .. +127	Zeichen, ganze Zahl
char, unsigned char	8 bit	0 .. +255	Zeichen, ganze Zahl

Der Datentyp **char** wird je nach Einstellung eines Compilerschalters (im Menü Options / Compiler) als signed char oder als unsigned char (Voreinstellung !) angesehen.

Der Datentyp einer **Konstanten** ergibt sich aus ihrem Wert, der Datentyp einer Variablen aus ihrer Deklaration. Konstanten werden auch als **Literale** bezeichnet.

Ganzahlige Konstanten werden normalerweise in der int - Dartsellung intern abgespeichert, größere ganzzahlige Konstanten in der long - Darstellung. Durch Anhängen eines Kennbuchstabens u oder U für unsigned, oder l bezw. L für long an die ganzahlige Ziffernfolge läßt sich eine vorzeichenlose bezw. lange Abspeicherung (32 Bit) erzwingen :

```
40000     abgespeichert als long : 32 bit
40000u    abgespeichert als unsigned int :  16 bit

40          abgespeichert als int  :  16 bit
40L         abgespeichert als long : 32 bit
```

Außer für die Null, dürfen ganze Dezimalzahlen nicht mit einer führenden Null beginnen.

```
0        1244        -27
```

Auch Zeichenkonstanten wie 'A' sind ganze Zahlen (**ASCII** Zeichensatz : American Standard Code for Information Interchange); sie können in numerischen Operationen genau so wie andere Integerwerte verwendet werden ; sie werden jedoch am häufigsten zum Vergleich mit anderen Zeichen verwendet.

```
char Zeichen;
cout  <<  "Bitte, einen großen Buchstaben eingeben ";   cin  >> Zeichen;
cout  <<   " Das eingegebene Zeichen hat in ASCII die Position : "
      <<   Zeichen - 'A' ;
```

Ganze Hexadezimalzahlen beginnen mit dem Zeichen 0x oder 0X :

0X2B 0xff

Die in C vordefinierten **reellen Datentypen (reelle Zahlen bezw. Gleitkommazahlen) sind :**

Datentyp	**interne Länge in bit**	**Dezimaler Zahlenbereich**	**Genauigkeit**
float	32 bit	-3.4E38 .. +3.4E38	7 Dezimalstellen
double	64 bit	-1.7E308 .. +1.7E308	15 Dezimalstellen
long double	80 bit	-1.1E4932 .. +1.1E4932	19 Dezimalstellen

Die interne Darstellung ist eine normalisierte Gleitkommazahl im Dualsystem :
123.4567 wird als 1.234567E+2 abgespeichert.
Dabei wird eine bestimmte Anzahl Bits für die normalisierte Mantisse verwendet (einschl. Vorzeichen) und eine kleinere Anzahl Bits für den Exponenten (einschl. Vorzeichen). Die für die Mantisse verwendeten Bits bestimmen im wesentlichen die Genauigkeit eines reellen Zahlenwertes.
Im Rechner sind also die ganzen Zahlen keine Teilmenge der reellen Zahlen, da die interne Darstellung ganz verschieden ist.
Beim Rechnen mit reellen Datentypen können sich Rundungs - und Umwandlungsfehler zu beachtlichen Gesamtfehlern summieren.
In einer konstanten Zahl legt ein Dezimalpunkt fest, daß es sich um eine Gleitkommakonstante (reelle Konstante) handelt.
Reelle Konstanten werden normalerweise in der double - Darstellung oder bei größeren Werten in long double abgespeichert. Durch Anhängen von f oder F für float oder von l bezw. L für long double läßt sich entsprechend eine andere interne Darstellungsform erzwingen.

Mit dem **Datentyp - Modifizierer const** (auch Datentyp - Attribut genannt) können wir jedem konstanten Wert vom Typ char, int, float einen Namen geben :

```
const       laenge  =  5;          //  Typ int ,  implizit
const int   breite  = 10;          //  Datentyp int explizit
const double flaeche = 0;          //  Datentyp double explizit
```

Der Inhalt solcher Speicherstellen wird bei Programmstart vorbesetzt und kann im Programmablauf nicht verändert werden : dies wird vom Compiler kontrolliert.

```
main()
   {  double radius, umfang, flaeche;
      const double pi = 3.1415;
      cout <<"\nRadius : ";      cin   >>   radius ;
      umfang  =  2 * pi * radius;
      flaeche  =       pi * radius  * radius ;
      cout  << "\nUmfang = "   << umfang   << ", Flaeche =  "  << flaeche ;
      return 0;
   }
```

Der Gültigkeitsbereich solcher symbolischer Konstanten ist ebenfalls der umschließende Block, bezw. bei globalen Definitionen das ganze Programm.

Die sog. Eingangsdaten eines Programms sind also entweder im Quellenprogramm als Literale bezw. als symbolische Konstanten vordefiniert, oder sie werden während der Programmausführung eingelesen.

Mit Hilfe der Preprozessoranweisung

#define SYMBOL Ersatztext

ist es möglich, vor der Übersetzung das SYMBOL im Programm durch den dahinter stehenden Ersatztext ersetzen zu lassen, und zwar an allen Programmstellen außerhalb von durch " " gekennzeichneten Zeichenfolgen. Es ist üblich, den Bezeichner einer solchen Symbolkonstanten mit Großbuchstaben zu schreiben :

```
#define PI  3.1415927
 main()
 {
        double umfang, radius=1.5 ;   / /  radius vorbesetzt.
         umfang  =  2 * PI * radius;
       .....
 }
```

In diesem Beispiel wird vom Preprozessor die Zeile mit umfang = 2 * PI * radius umgewandelt in

umfang = 2 * 3.1415927 * radius ;

In **#define SYMBOL Ersatztext** darf der Ersatztext nicht mit einem Semikolon abgeschlossen werden.
Jede Preprozessoranweisung kann in weiteren Zeilen fortgesetzt werden, indem eine Zeile durch \ abgeschlossen und dann in der Folgezeile fortgesetzt wird. Der Ersatztext in #define kann auch ein beliebiger Text sein :

```
#define  FEHLER1          "Die Eingabe von der Konsole war fehlerhaft"
```

Ein nachfolgendes

```
cout  <<   FEHLER1;
```

wird vom Preprozessor übersetzt in

```
cout  <<  "Die Eingabe von der Konsole war fehlerhaft" ;
```

Allgemein bezeichnet man solche durch #define festgelegten Namen auch als **Makros :** Jeder Makroname im C Programm wird vom Preprozessor durch den entsprechenden Makrotext ersetzt. Darüber später mehr!

Der Gültigkeitsbereich eines mit #define vereinbarten Namens erstreckt sich von der Definition bis zum Ende der Quelldatei.
Die bisher erwähnten Vereinbarungen für Namen sind Namens - Definitionen und erzeugen Objekte im Speicher. Solche Definitionen sind später zu unterscheiden von Deklarationen, welche nur die Eigenschaften von solchen Objekten festlegen. Jede Definition ist auch eine Deklaration, jedoch nicht umgekehrt. Darüber später mehr !

Ein **konstanter Ausdruck** ist ein Ausdruck, in dem nur Konstanten beteiligt sind. Solche Ausdrücke werden schon vom Übersetzer bewertet, und nicht erst zur Laufzeit. In C dürfen bei dieser Compile-Time-Auswertung von Ausdrücken auch Symbole vorkommen, welche gemäß

#define mit einem Ersatztext zu ersetzen sind; dagegen gelten in C Ausdrücke mit symbolischen Konstanten (wie const int breite = 5;) in diesem Sinn nicht als konstante Ausdrücke. Insbesondere darf in C in switch-Anweisungen die Konstante nach case keine symbolische Konstante der Art const breite = 5; sein. In C++ ist diese Einschränkung aufgehoben : dort dürfen auch nach case solche symbolischen Konstanten stehen.
In diesem Sinn ist C++ vom Preprozessor weniger abhängig als C .

2.) Programmverzweigungen

Jede höhere Programmiersprache enhält zwei Arten von Anweisungen :

- was ist zu tun (z.B. Wertzuweisung, Eingabe , Ausgabe, Funktionsaufruf)
- wie ist etwas zu tun (z.B. if - Anweisung)

2.1 if - Anweisung (ref. Beispiel 2)

Programmverzweigungen bieten die Möglichkeit, in Abhängigkeit von eingelesenen oder berechneten Daten bestimmte Programmzweige zu durchlaufen. Die entsprechenden Auswahlanweisungen in C werden vom Compiler in bedingte Sprungbefehle der Maschinensprache übersetzt.

Die einfache if - Anweisung :

```
if (Bedingungsausdruck )
        Ja_Anweisung;
```

Der auf if nachfolgende Bedingungsausdruck steht meistens in runden Klammern : er wird bewertet, und sofern das Resultat dieser Berwertung nicht null ist (also positiv, oder auch negativ), so wird die Ja_Anweisung ausgeführt. Ist das Resultat gleich Null, so wird die Ja_Anweisung übergangen.

Als Bedingunsausdruck gilt jeder in C definierte mögliche Ausdruck. Vorwiegend werden dafür Vergleiche verwendet :

```
int wert;
if ( wert  == 0 )   cout   <<  "\nwert ist  0 ";
if  ( wert >= 0 )
      {  cout <<  "\nwert ist größer als 0 ";
         wert  = 0;
         cout  <<  "\nwert zurückgesetzt auf 0 ";
       }
```

Das Ergebnis eines solchen Vergleichs ist entweder 1 (bei wahr) oder 0 (bei falsch) . Wenn bei erfüllter Bedingung mehrere Anweisungen ausgeführt werden sollen, so wird die sog. **Blockanweisung** verwendet : eine durch { } eingeklammerte Folge von Anweisungen ('compound statement '). Am Ende einer solchen Blockanweisung, also nach }, erfolgt kein Semikolon ! Auch eine einfache Anweisung bei erfüllter Bedingung kann als Blockanweisung formuliert werden. Dies hat den Vorteil, daß spätere Erweiterungen leichter ohne Fehler zu machen sind.

Ist die Ja-Anweisung eine einzige (einfache) Anweisung, so kann sie in der gleichen Zeile nachfolgend zur if - Bedingung stehen. Eine komplizierte Ja-Anweisung bezw. eine Blockanweisung schreiben wir in der nachfolgenden Zeile, eingerückt um eine Tabulatorposition bezw. um 4 Leerstellen; damit läßt sich die logische Programmstruktur betonen.

Da bei if der numerische Wert eines Ausdrucks berechnet wird, sind bestimmte Abkürzungen möglich :

if (Ausdruck)	ist gleichbedeutend wie	if (Ausdruck == 1)

Die zweiseitige if - Anweisung :

if (Bedingungsausdruck) Ja_Anweissung ; **else** Nein _Anweisung ;

Wenn der Bedingungsausdruck den Wert 0 (also falsch) liefert, wird die hinter else stehende Nein_Anweisung ausgeführt. Dies kann eine einfache Anweisung oder auch eine Blockanweisung sein.

```
float  radikand,  wurzel;
 ....
 if (radikand  >=0 )
       {  wurzel = sqrt(radikand) ;
          cout <<  "nReeller Wurzelwert = "  << wurzel ;
       }
 else  { wurzel  =  sqrt( -radikand) ;
         cout  << "\nImaginärer Wurzelwert  =  "  << wurzel ;
       }
```

Unmittelbar nach else darf kein Semikolon stehen, sonst beendet es die if - Anweisung.

if-Anweisungen können auch verschachtelt werden :

```
if (ausdruck1)
      if  ( ausdruck2)
            Anweisung1;
      else
            Anweisung2;
```

Hier gilt die allgemeine Sprachregel : ein else - Teil gehört immer zum nächsten höheren if, für das noch kein else - Teil existiert. Die folgende Anweisungsgruppe wird dank der Blockanweisung anders ausgeführt als die obige :

```
if (ausdruck1)
     {  if (ausdruck2)
              Anweisung1;
     }
else  Anweisung1;              //     Hier gehört else zum ersten if.
```

Programmstruktur : else steht immer beginnend in der gleichen Spalte wie das entsprechende if .

2.2 Vergleichsausdrücke

In C sind die folgenden **Vergleichsoperatoren** definiert :

< <= > >=

== (für gleich) != (für ungleich)

Dabei haben die beiden letzten (== und !=) eine geringere Priorität als die übrigen :

if (a > b == c > d) ....; ist gleichbedeutend mit if ((a>b) == (c>d) ....;

Ein Vergleichsausdruck kann auch mehrere Vergleiche beinhalten. Das Resultat jedes einzelnen Vergleichs ist vom Typ int, entweder 1 (wahr) oder 0 (falsch). Umgekehrt gilt der Wert 0 als falsch, und jeder Wert ungleich 0 als wahr (positiv oder negativ).
Zu achten ist auf den Unterschied zwischen = (Wertzuweisung) und == (Vergleich).
Werden Werte von verschiedenem Datentyp verglichen, so wird z.B. der int - Wert in einen reellen Wert umgewandelt.
Bei Vergleich von reellen Größen auf Gleichheit (==) oder Ungleichheit (!=) kann das Resultat durch Umwandlungsfehler oder Rundungsfehler verfälscht werden. Dies betrifft besonders Nachpunktstellen und Ergebnisse, welche durch fortlaufende Addition und Subtraktion (etwa in Programmschleifen) entstanden sind. Eine Abfrage von eingelesenen oder mit Konstanten besetzten reellen Variablen ist dagegen unkritisch.

Als Vergleichsausdrücke sind auch arithmetische Ausdrücke möglich :

if (x+y > 0)... ist gleichbedeutend mit if ((x+y) > 0) ...

Die arithmetischen Operatoren haben eine höhere Priorität als die Vergleichsoperatoren.

Vergleichsergebnisse lassen sich mit **logischen Operatoren** verknüpfen :

! (Negation) && (logisches Und) || (logisches Oder)

```
if (    x==y  &&  (   a-b) > 0 )...
```

ist gleichbedeutend mit

```
if  (   (x==y)  &&  ( (a-b) > 0 )   )...
```

Die logischen Verknüpfungen haben dabei die niedrigste Priorität, && ist hierarchisch noch eine Stufe höher als || . Diesen logischen Operatoren liegen die Regeln der **Aussagelogik** zugrunde.

Zu beachten ist noch, daß die einfachen Zeichen & und | für Bitoperationen bei ganzen Zahlen verwendet werden; das Operatorzeichen ~ kennzeichnet die bitweise Negation. Die logischen Verknüpfungen werden in einem Ausdruck von links nach rechts bewertet, nur bis das logische Resultat feststeht :

A && B && C && D

Wenn hier B den Wert 0 hat, wird der >Ausdruck nicht mehr weiter untersucht.

Bei der Reihenfolge von Operationen werden also arithmetische Operationen vor den Vergleichen und diese vor den logischen Operationen durchgeführt, wenn nicht durch runde Klammerung anders festgelegt.

Besondere Beachtung verdient : if(1< y && y < 20)..
Fehlerhaft wäre hier zu schreiben : if (1<y <20)..

Das folgende Beispiel liest einen Buchstaben von der Konsole ein und wandelt Großbuchstaben in Kleinbuchstaben um :

```
char    c ;
cout  << "Bitte, einen Buchstaben eingeben  ";  cin  >>  c ;
 if  ( ( c >= 'A' )  &&  ( c  <= 'Z' ) )  //  Vergleich von einfachen ASCII - Zeichen
       c  =   'a'  +  c  - 'A';              //  ASCII Zeichensatz  angenommen
 cout  <<  c;
```

2.3 Die switch _ Anweisung (ref. Beispiel 3)

Man kann mehrere if -Anweisungen miteinander verketten, um Alternativen zu unterscheiden. Eine standardmäßige Anweisungsfolge ist :

```
if (ausdruck1)
        anweisung1;
else if (ausdruck2)
        anweisung2;
    else if (ausdruck3)
        anweisung3;
        else
          anweisung4;
```

Für die Anweisungen dürfen wieder ganze Blockanweisungen stehen, in welchen wiederum if-Anweisungen stehen können.
Ein Beispiel :

```
#include  <iostream.h>
#include  <math.h>
main()
{
      char  zeichen ;
      double  grad , bogen;
      cout  << "\nWinkel in  [°] eingeben :";          cin  >>  grad  ;
      cout  <<" s = Sinus ,  c  = Cosinus ,  t  = Tangens  -> ";  cin  >> zeichen ;
      bogen  =  grad  * M_PI  / 180 ;              // Bogenmaß     berechnen
      if  ( zeichen  == 's' )
          cout  <<  "\nSin "  <<  grad  <<  " = "    <<  sin(bogen);
      else  if  ( zeichen  == 'c'  )
              cout  << "\nCos "  << grad  << " = "  << cos(bogen);
          else  if ( zeichen  == 't')
                  cout  << "\nTan  "  << grad  << " = "  << tan(bogen);
              else  cout <<"\n\afehler : kein Kennbuchstabe  s  c  t  ";
   ..  ...  ..
}
```

Wenn dabei speziell auf konstante Werte geprüft wird, gibt es die **switch-Anweisung** anstelle von verketteten if's. Die switch - Anweisung ist eine Auswahl unter mehreren Alternativen.

```
switch  ( Auswahlausdruck )
    {
        case    Konstante1 :  Anweisungsfolge_1;   break;
        case    Konstante2 :  Anweisungsfolge_2;   break;
            . . . . . .
        case    Konstante_n :  Anweisungsfolge_n;    break;
        default :   Anweisungsfolge_s  ;   break;
     }
```

Der hinter switch stehende Ausdruck wird ausgewertet. Das Ergebnis ist (evtl. nach Umwandlung) ein Wert vom Typ int oder char. Dieser wird von oben nach unten mit den Konstanten hinter case verglichen. Diese Konstanten sind üblicherweise ganze Zahlen oder Zeichenkonstanten (z.B. 'J'). Reelle Datentypen werden nach den Regeln für gemischte Ausdrücke ganzzahlig gemacht : Stellen hinter dem Komma werden dabei abgeschnitten.
Bei angetroffener Gleichheit werden entsprechend die hinter dem **:** stehenden Anweisungen ausgeführt, einschließlich aller nachfolgenden Anweisungen hinter den anderen case-Teilen.
Wird keine Gleichheit gefunden, werden die Anweisungen hinter default ausgeführt. Der default – Teil muß aber nicht vorhanden sein. Stimmt der Wert des Auswahlausdrucks mit keiner Konstanten überein, und ist auch kein default - Teil vorhanden, so wird kein Zweig ausgeführt.
Jede Anweisungsgruppe kann man mit break abschließen : in diesem Fall wird nach break die switch-Anweisung verlassen. Fehlt ein break am Ende, so werden alle nachfolgenden Zweige durchlaufen, bis zum nächsten break oder switch - Ende.
Jede case - Konstante darf nur ein einziges Mal nach einem case vorkommen. Vor einer Anweisungsgruppe können auch mehrere case - Konstanten stehen :

```
case 0 : case 1 :  Anweisungsfolge_0_1; break;
case 2 :           Anweisungsfolge_2; break;
    . . . . . .
```

Häufig wird im default - Teil ein Fehlerfall (z.B. falsche Eingabe von Konsole) programmiert.
Die recht übersichtliche Auswahlanweisung switch hat die Einschränkung, daß nur Konstanten vom Typ int oder char als Vergleichsoperanden zugelassen sind.
Ein Beispiel :

```
#include  <iostream.h>
main()
{     int zensur ;
      ...  ....  ....
      switch  (zensur)
          {
            case  1 :   ;                  //  Semikolon ist hier eine Leeranweisung
            case  2 :   {  cout  <<  "\nEinstellen ";  break ;      }
            case  3 :   {  cout  <<  "\nGespräch ";   break;      }
            case  4 :   {  cout  <<  "\nZusatzprüfung "; break ;       }
            case  5 :                        // Programm wird nach nächstem Label fortgesetzt.
            case  6 :   {   cout   <<   "\nNicht einstellen   ";  break; }
            default :    cout  <<  "\nFalsche  Note   ";
          }
}
```

In C gibt es Einschränkungen bezüglich der Verwendung von symbolischen Konstanten nach case, wie das folgende Beispiel zeigt :

```
#define  WERT  5
main()
{     const wert  = 6 ;       //  WERT und wert sind nun verschiedene symbolische Konstanten
      .....
      switch (  ...  )
          { case WERT : ....          //   in C und in C++ erlaubt
            case  wert    :  ...      //    nicht in C, nur in C++ erlaubt
           }
       ...  ...   ...
 }
```

Programmstruktur : die geschweiften Klammern { } sind gegen switch eingerückt.

2.4 Die goto - Anweisung

Die **goto -Anweisung** ist eine unbedingte Sprunganweisung :

```
goto  ziel;     // ziel ist hier eine  Programmarke,  zu   der gesprungen wird .
   . . . .
ziel : cout "\nZiel ist erreicht ";
```

Eine Marke (wie hier ziel) hat die gleiche Form wir ein Variablenname. goto - Anweisungen finden auch Verwendung in bedingten Anweisungen :

```
if (bedingung)   goto ziel;
      . . . . .
ziel : cout .. .. .. ..
```

Die Anweisung goto setzt das Programm immer an der hinter dem Sprungziel stehenden Anweisung fort.
Die Sprungmarke muß in der gleichen Funktion liegen, in der das goto benützt wird. Die Sprungziele erhalten einen eindeutigen frei wählbaren Namen (Bezeichner), welcher unmittelbar vor der nachfolgend auszuführenden Anweisung steht, durch einen Doppelpunkt getrennt.

Eine strukturierte C-Programmierung erlaubt es, ohne das goto auszukommen. In Ausnahmefällen ist es jedoch sinnvoll, mit dem goto zu arbeiten. Beispiele dafür sind fehlerhafte Eingabe von der Konsole. Auch können verschachtelte Programmschleifen mit goto auf einmal verlassen werden. Die Anwendung von break würde hier nur die innerste Schleife beenden. Das Gleiche gilt für verschachtelte switch – Anweisungen.

Code mit goto-Anweisungen ist im allgemeinen schwieriger zu verstehen. Daher sollten goto - Anweisungen nur selten verwendet werden. Insbesondere sollte ein goto nicht in enen inneren Block hinein führen : solche Programmteile sind später sehr schwierig zu ändern.

3.) Programmschleifen

3.1 Die while - Schleife (ref. Beispiel 4)

In vielen Anwendungen müssen Anweisungen mehrmals mit unterschiedlichen Daten

ausgeführt werden.
Die **while - Schleife** dient zu bedingten Schleifen, bei denen die Laufbedingung vor dem (ersten) Schleifendurchlauf auf true oder false geprüft wird .

while (Bedingungsausdruck)
Schleifenanweisung;

Der Bedingungsausdruck wird vor jedem Schleifendurchlauf neu bewertet. Im Falle von wahr (1 bzw !=0) wird die Schleifenanweisung ausgeführt. Im Falle von falsch (0) wird die Schleifenanweisung nicht mehr ausgeführt, und die while - Schleife ist damit beendet ; es folgt dann der an die while - Schleife sich anschließende Teil des Programms. Die Schleifenanweisung kann eine einzige Anweisung oder auch eine Blockanweisung sein.
Innerhalb der Schleifenanweisung bzw. Blockanweisung muß der Bedingungsausdruck so verändert werden, daß die while - Schleife ein Ende findet. Wenn die Bedingung der while - Schleife bereits vor dem Eintritt in die Schleife nicht erfüllt ist, wird der Schleifenkörper gar nicht ausgeführt.

```
int  j = 0;
while  ( j <= 200 )
    {
       cout << "\n"  << j  ;          // Laufende Werte ausgeben
       j  = j+10 ;
     }
```

Hinter der runden Klammer des Bedingungsausdrucks darf kein Semikolon stehen.

Programmstruktur : Schleifenanweisung einrücken !

Gleich noch ein Beispiel :

```
char  zeichen  =  ' ';
while (zeichen  !=  '\r' )
   {
      zeichen  = getch();
   }
```

Der Block hinter der while - Anweisung wird hier so oft durchlaufen, wie das eingelesene Zeichen nicht gleich dem ASCII Code der Return-Taste ist.
Bei reellen Variablen im Bedingungsausdruck können sich bei forlaufender Addition oder Subtraktion Umwandlungs - und Rundungsfehler so summieren, daß der beabsichtigte Endwert nicht genau getroffen wird.
Die Ungenauigkeiten der reellen Zahlendarstellung und Rechnung können besonders dann zu numerischen Problemen führen, wenn Stellen hinter dem Komma zu verarbeiten sind. Ein Beispiel ist der endliche Dezimalbruch 0.1 , der bei der Dualumwandlung einen unendlichen Dualbruch ergibt und daher nur gerundet dargestellt werden kann.

Für reelle Zählschleifen empfehlen sich folgende Maßnahmen :

- nie den Endwert auf gleich oder ungleich prüfen,
- nur auf größer, größer/gleich, kleiner oder kleiner/gleich testen,
- den Endwert gegebenenfalls mit halber Schrittweite korrigieren,
- oder reelle Größen aus ganzahligen Laufvariablen ableiten.

Ein Beispiel :

```
#include  <iostream.h>
main()
```

```
{      double  xa,  xe,  xs,  x;
       cout  <<  "\n Anfangswert  -> ";          cin   >>   xa;
       cout   << "      Endwert          -> ";      cin  >>   xe;
       cout   <<  "  Schrittweite     -> ";         cin   >>  xs;
        x  =  xa ;
        while (  x <=  xe + 0.5 * xs )       // Korrektur wegen reeller Ungenauigkeit
             {
                cout    <<  endl     <<  x ;
                x  =  x + xs ;
              }
       return 0:
}
```

Mit der **break -** Anweisung innerhalb der while-Schleife kann man die while - Schleife unmittelbar verlassen.
Mit der **continue -** Anweisung wird nur der aktuelle Schleifendurchlauf beendet und es beginnt ein neuer Durchgang, wobei zunächst die Schleifenbedingung geprüft wird.
Das folgende Beispiel zeigt den Gebrauch von break und continue im Zusammenhang :

```
        long  double  zahl;
        while  (1)                                  //   Endlosschleife
            {  cout    << "\nGib eine Zahl ein,  Ende mit  0: \n" ;
               cin     >>  zahl;
               if  ( zahl  < 0 )  continue;       //  Neuer Schleifendurchlauf
               if  ( zahl  == 0 ) break;          //    Schleifenende
               zahl   = sqrt(zahl);
               cout    <<   zahl  ;
             }
```

Im Interesse der strukturierten Programmierung sollte nach Möglichkeit auf die Verwendung von break und continue bei Programmschleifen verzichtet werden: sie können zu unübersichtlichen Programmstrukturen führen.
Mit dem Funktionsaufruf **exit(status);** läßt sich ein Programm beenden; status ist dabei eine int - Größe, der Exit - Code, dessen Wert Null normalerweise für eine fehlerfreie Ausführung steht.
Mit dem Funktionsaufruf **abort();** ,ohne Parameter, wird ein Programm ohne Fehlermeldung abgebrochen: der resultierende Exit - Code hat den Wert 3 .
Hinweis zur cout - Anweisung : in jedem Zusammenhang, in dem der Wert einer Variablen vom bestimmten Typ benutzt werden darf, kann auch ein allgemeiner Ausdruck von diesem Typ stehen, z.B. sqrt(zahl).

3.2 Die for - Schleife (ref. Beispiel 4)

Die **for -** Schleife ist meistens eine Zählschleife, bei der eine Laufvariable von einem Anfangswert bis zu einem Endwert mit einer bestimmten Schrittweite verändert wird.

```
        for (i=1;  i<100; i = i+1)
            sum = sum + i;
```

Der erste Ausdruck i = 1 initialisiert die Laufvariable i, im zweiten Ausdruck i<100 wird eine Bedingung abgefragt, im dritten Ausdruck i= i+1 wird die Laufvariable um 1 erhöht. Die for-Anweisung wird so lange durchlaufen, wie der zweite Ausdruck true ist.
Allgemeiner :

```
    for ( Laufv = Anfangswert ; Laufv verglichen mit Endwert; Laufv verändert um Schrittw)
            Anweisung bzw. Blockanweisung ;
```

Äquivalent dazu könnten wir auch die folgende while - Schleife formulieren :

```
Laufvariable = Anfangswert;
while ( Laufvariable verglichen mit Endwert )
    {  Anweisung bzw. Blockanweisung ;
       Laufvariable verändert um Schrittweite ;
    }
```

Ganz allgemein läßt sich eine for - Schleife wie folgt formulieren :

for (Anfang; Laufbedingung; Veränderung) Anweisung ;

Innerhalb der runden Klammer der for-Anweisung stehen also drei Ausdrücke, durch Semikola getrennt.

Der erste Ausdruck wird vor Beginn der Schleife ausgeführt. Er bestimmt den Anfangswert der Laufbedingung, vor Antritt der Schleife.

Der zweite Ausdruck, die Laufbedingung, wird vor jedem Schleifendurchlauf bewertet. Ergibt dieser Bedingungsausdruck den Wert 0 (falsch), so ist die for - Schleife beendet : Die Anweisung wird dann nicht mehr durchgeführt und der Programmablauf verzweigt unmittelbar zum anschließenden Programmteil. Ist die Laufbedingung bereits vor Eintritt in die Schleife nicht erfüllt, so erfolgt gar kein Durchlauf.

Der dritte Ausdruck, die Veränderung, wird nach jedem Durchlauf neu bewertet: er enthält grundsätzlich die Veränderung der Laufvariablen. Hinter diesem dritten Ausdruck innerhalb der Klammer steht kein Semikolon !

Hinter der Klammer steht die Anweisung des Schleifenkörpers, bezw. eine Blockanweisung (geschweifte Klammern { }) .

Programmstruktur : Der Anweisungsblock wird gegenüber for eingerückt.

Ein Beispiel :

```
int k,  anfang,  ende , schrittweite ;
cout << "\n Anfangswert  Endwert  Schrittweite  --> " ;
cin   >>  anfang  >>  ende  >> schrittweite;
for ( k = anfang ;  k <= ende ;  k  = k+schrittweite)
      cout  <<  "\n  "  <<  k;
```

Auch bei der for - Anweisung kommen break und continue zum Tragen; continue verzweigt hier zunächst zum Veränderungs - Ausdruck.

Einzelne Bestandteile der for -Anweisung können entfallen, nicht jedoch die Semikola.

Die Folge

```
for ( ;  ;  )  Anweisung ;
```

stellt eine endlose Schleife dar.

Unmittelbar hinter den runden Klammern der for - Schleife darf kein Semikolon stehen , sonst beendet dieses Semikolon die for-Schleife.

Innerhalb einer for-Anweisung kann eine Laufanweisung in C++ auch lokal neu definiert werden :

```
for (int i=1;  i<10;  i = i+1) ....
```

Zu beachten ist, daß hier die Variable i ihre Gültigkeit bis zum Ende des aktuellen Programmblocks hat, wie alle anderen in diesem umschließenden Block definierten Namen.

Programmstruktur : Der Anweisungsblock wird gegenüber for eingerückt.

Noch ein Beispiel :

```
double   bogen,  grad,  s;
for (grad = 20;  grad <= 60 ;  grad  = grad + 1)
```

```
{
      bogen  =  grad  *  M_PI  / 180 ;
      s          =   sin(bogen) ;
      cout   << "\n"   << s;
 }
```

3.3 Die do - while Schleife (ref. Beispiel 5)

Bei der **do - while -**Schleife liegt die Schleifenkontrolle am Schleifenende : die Anweisungen werden also mindestens ein Mal durchgeführt. In den Schleifen-Anweisungen muß die Laufbedingung so verändert werden, daß ein Ende der Schleife erreicht wird.

```
do
      Anweisungen ;
while ( Bedingungsausdruck )
```

Auf das Kennwort do folgen die Anweisungen bzw. Blockanweisung des Schleifenkörpers. Es können hier mehrere Anweisungen, ohne { } stehen, da eine Anweisungsfolge hier durch do - while eingeklammert ist. Die Blockanweisung hat allerdings den Vorteil, daß die Kennwörter do und while besser erkennbar sind. Auf das Kennwort while folgt der Bedingungsausdruck, er wird nach jedem Schleifendurchlauf neu bewertet.

```
int j = 10;
do
      {  cout << "\n"  << j;       // Laufenden Wert ausgeben
         j   = j+1 ;
      }
while  (i <= 50 ) ;                    // Schleifenkontrolle
```

Besonders zu beachten ist hier das Semikolon nach der while - Bedingung.
Auch bei do -while kommen break und continue zur Anwendung, wie bei while .
Die do-Schleife wird hauptsächlich dann benutzt, wenn eine Schleife mindestens einmal durchlaufen werden soll.
Programmstruktur : Anweisung bezw. Blockanweisung gegenüber do einrücken.
while beginnt in gleicher Spalte wie do.

Bei allen programmierten Schleifen lohnt es sich, die Bedingung für den ersten und den letzten Schleifendurchlauf sorgfältig zu beachten; hier schleichen sich gerne Fehler ein, welche unter anderem zu endlosen Schleifen führen können.

3.4 Einfache Schleifenprogramme für Ein - und Ausgabe, getchar() und putchar(c)

Für die Eingabe und Ausgabe von einzelnen Zeichen dienen die Funktionen getchar() und putchar(c), welche in stdio .h deklariert sind. getchar() hat keinen Parameter, die Funktion liest ein Zeichen von der Standardeingabe stdin und liefert es als Datentyp int ab. Die Eingabe einer Zeichenfolge über die Tastatur wird gepuffert bis 'Enter' gedrückt wird.

Die Funktion putchar(c) gibt ein Zeichen vom Typ int bezw. char in die Standardausgabe stdout aus.

Beispiel : Ein einfaches Kopierprogramm.

```
#include  <stdio.h>
main()
{
      int c;
      printf("\nBitte, eine Zeichenfolge eingeben, Abschluß mit  ^Z ->");
      c  =  getchar();                        //  Eingabe von der Konsole   (stdin)
      while  (  c !=  EOF)
            {    putchar (c);                        //  Ausgabe am Bildschirm   (stdout)
                 c  = getchar();
             }
      return 0;
}
```

Beispiel : Eingelesene Zeichen zählen :

```
#include <stdio.h>
main()
{
     long  nc  = 0;
     printf("\nBitte, eine Zeichenfolge eingeben, Abschluß mit  ^Z    ->");
     while (getchar()  !=  EOF)              //     EOF resultiert aus Eingabe von  ^Z
            nc  =  nc + 1 ;
     printf("%ld\n",  nc);                   // enspricht cout <<   nc   <<  endl ;
     return 0 ;
}
```

Beispiel : Wiederholende Kontrollschleife bei Eingabe :

```
#include  <iostream.h>
main()
{
     double   xa ;              //   Einzulesende  reelle Größe
     int fehler;                // Schaltervariable

     do
        {
          cout  << "\nReelle Größe  0 < xa  < 100    ->" ;    cin  >> xa ;
                                  // cin : führende Leerstellen werden überlesen,
                                  //       Ende bei erster nachfolgender Leerstelle
          fehler  =  !( 0 < xa    &&    xa  <  100 ) ;
        }
     while (fehler) ;           // Wiederhole ,  solange fehlerhafte Eingabe
     return 0;
}
```

Beispiel : Wiederholende Leseschleife mit dem Wert 0 als Endemarke :

```
#include <iostream.h>
main()
{
     int  wert;                          //  einzulesender Wert
     int  sum = 0,  n = 0;               //  Summe  sum  und  Zaehlvariable  n

     do
          {
             cout  << "\nGanzzahlige Größe,  Ende mit 0  "  << n+1  <<". Wert -> ";
             cin  >> wert;
             if ( wert  != 0 )
                 {     sum  = sum + wert;   n  =  n+1;
                  }
          }
      while ( wert  != 0) ;              // Wiederhole ,  solange  keine   0 -Eingabe
      return 0;
}
```

4.) Die Entwicklungsumgebung (Einführung)

Die folgenden Unterkapitel behandeln einige häufig benutzte Möglichkeiten der C++ Entwicklungsumgebung. Sie sind aus dem 'Borland C++ Benutzerhandbuch ' entnommen ; das hier referierte Benutzerhandbuch beschreibt noch eine große Menge von weiteren Bedienungsmöglichkeiten .
Alle Bedienungsmöglichkeiten können auch online im C++ Hilfemenue (F1) nachgelesen werden.

4.1 Einige grundsätzliche Möglichkeiten in der Entwicklungsumgebung

Über die Menuezeile erreichen Sie alle Menuepunkte. Bei einem Befehl mit Fortsetzungspunkte n (...) wird ein Dialogfenster geöffnet. Bei einem mit Pfeilspitze markiertem Befehl gelangt man in ein Untermenue. Befehle ohne Zusatzzeichen werden unmittelbar ausgeführt.

In der Entwicklungsumgebung können viele Fenster offen sein. Öffnen Sie dieselbe Datei in mehreren Fenstern, wirken sich Aktionen in einem Fenster auf alle anderen Fenster aus. Auch können mehrere Dateien auf einmal geöffnet sein.

Wurde eine Datei im Editor - Fenster geändert, erkennt man dies an einem Sternchen (*) links unten neben der Spalten - und Zeilennummer.

Mit File | DOS Shell kann man C++ verlassen, um in der DOS - Ebene DOS-Befehle einzugeben. Dabei bleibt C++ im Speicher : mit dem DOS Befehl Exit kommt man wieder zur C++ - Entwicklungsumgebung zurück.

Einige häufig verwendete Tastenkürzel in der C++ Entwicklungsumgebung sind :

F1	Hilfe	
F2	File | Save	Speichert editierte Datei ab, mit vorgegebenem Namen
F3	File | Open	Öffnet Dialogfenster zum Laden einer Datei
F4	Run | Go to Cursor	Programmausführung bis zu Cursorpunkt.
F7	Run | Trace into	Programmstart im Debuggermodus: Einzelschritt
F8	Run | Step over	Wie F7, Funktionen werden übersprungen.

4.2 Spezielle Editierbefehle : Edit - und Search - Menue

Menue Edit :

Cursorbewegungen : Zeilenanfang Taste = Home (Pos1)
Zeilenende Taste = End (Ende)
Programmanfang Taste = ^ Home (^Pos1)
Programmende Taste = ^ End (^Ende)

Markierten Text in Zwischenablage : Menue = Edit | Cut (ausschneiden !)
Edit | Copy (kopieren)

Text von Zwischenablage in Cursorpos. einfügen: Menue = Edit | Paste

Letzten Edit Befehl rückgängig machen : Menue = Edit | Undo

Kopiere Programmbeispiel aus dem
aktuellen Hilfefenster in Zwischenablage: Menue = Edit | Copy Example

Menue Search : (Untermenues mit Dialogfenster)

Text in editierter Datei suchen : Menue = Search | Find
Text in editierter Datei ersetzen: Menue = Search | Replace

4.3 Austesten von Programmen : Debug - Menue

Steuern von Watch - Ausdrücken :

Neuer Watch - Ausdruck in Watch-Fenster : Menue = Debug | Watches | Add Watch
Watch-Ausdruck wieder löschen : Menue = Debug | Watches | Delete Watch
oder Debug | Watches |Remove all Watches

Steuern von Breakpoints :

Breakpoint in Cursorpos. setzen / löschen : Menue = Debug | Toggle Breakpoint
Alle Breakpoints anzeigen : Menue = Debug | Breakpoint
(Dialogfenster ermöglicht Löschen von Breakpoints sowie auch Setzen von neuen Breakpoints)

Anzeige von Stack : Funktionsaufrufe ---> Menue = Debug | Call Stack

5.) Ausdrücke und Anweisungen

5.1 Einfache und mehrfache Wertzuweisungen, Kommaoperator (ref. Beispiel 6)

In einer Programmiersprache wie C sind die einzelnen Operanden eines Ausdrucks von jeweils bestimmten Datentyp. Jede Variable in einem Ausdruck hat ihrem Datentyp entsprechend einen begrenzten Wertebereich. Der Wert eines Ausdrucks ist wiederum von bestimmten Datentyp und liegt entsprechend in einem für den Rechner festgelegten Wertebereich. Das genaue Resultat eines berechneten Ausdrucks hängt auch von der Reihenfolge der einzelnen Berechnungen innerhalb dieses Ausdrucks ab.

Eine einfache Anweisung enthält in der Regel Wertzuweisungen und Funktionsaufrufe :

```
x = y;          x = y + z * w ;
funct();
```

Ein allgemeiner Ausdruck wird erst durch das nachfolgende Semikolon zu einer Wertzuweisung.

Eine Wertzuweisung (=) wird in C genau wie jeder andere Operator betrachtet. Dadurch sind zum Beispiel Konstruktionen der folgenden Form möglich :

```
v3 = v1 + (v2 = 5);
```

Hier wird zunächst der Wert von v2 auf 5 gesetzt, entsprechend der Anweisung v2=5, dann wird die Summe von v1 und v2 im v3 gespeichert.
Ein weiteres Beispiel ist :

```
v3 = (v1 = 4) + (v2 = 5);
```

Hier erhalten zunächst v1 und v2 die Werte 4 bezw. 5 zugewiesen; die Summe (4+5) wird schließlich der Variablen v3 zugewiesen.

Entsprechend können wir formulieren :

```
if ( (c = getchar() ) != EOF ) .....
```

Hier wird zunächst durch die Elementarfunktion getchar() ein Zeichen von der Konsole eingelesen und in c abgespeichert; anschließend wird der Wert von c mit EOF verglichen.

Zu beachten ist noch der Unterschied zwischen Initialisierung (z.B. int i = 5;) und Wertzuweisung (i = x + y ;) . Die Initialisierung spielt sich bei der Definition einer Variablen ab, bei der Wertzuweisung hat die Varible links vom Gleichheitszeichen schon einen Speicherplatz.
Oft verwendet werden auch die möglichen **Mehrfachzuweisungen :**

```
x1 = x2 = x3 = 2.2 ;    // x3 = 2.2;   x2 = x3;   x1 = x2;
```

Von **rechts nach links** wird hier allen drei Variablen der Wert 2.2 **zugewiesen.** Solche Mehrfachzuweisungen eignen sich zum Initialisieren von Variablen .
Diese sogenannte **Rechts - Assoziativität** gilt bei allen Arten von Zuweisungen.

Ein Ausdruck kann also aus allen arithmetischen, logischen und Vergleichs-Operatoren gebildet werden, und sogar Wertzuweisungen enthalten.

Die Klammerung innerhalb von Ausdrücken dient
-- zur Festlegung von Prioritäten
-- Erhöhung der Lesbarkeit.

Es sind hierzu nur runde Klammern zugelassen.
Wir können auch mehrere Ausdrücke durch den **Kommaoperator** verknüpfen.
Der Kommaoperator dient zur Verkettung von allgemeinen Ausdrücken, er darf nicht verwechselt werden mit dem Komma zur Trennung von Funktionsargumenten.

```
while  ( x = getchar(), x != EOF)       z  = z+1;
```

Hier wird mit jedem Schleifendurchlauf die Funktion getchar() zum Einlesen der Tastatureingabe ausgeführt und das Ergebnis an x zugewiesen. Anschließend wird ermittelt, ob x ungleich EOF ist (bei Eingabe von ^Z). Solange dies der Fall ist, wird z erhöht.

Die Ausdrücke eines Kommaoperators werden also von links nach rechts ausgewertet.

Ein entsprechendes Beispiel mit cin, cout :

```
int eingabe ;
while (cout <<"\nganze positive Zahl eingeben : " ,  cin >> eingabe,  eingabe  > 0 )
        {
            cout  << "\nDie eingegebene Zahl ist "  << eingabe   <<endl;
        }
```

Noch ein Beispiel : Aufaddieren der Zahlen 1 bis 100 :

```
                int i, ende, summe;
                for (i=1, ende = 100, summe = 0;                 //   Initialisierung
                        i  <= ende;                              //   Endebedingung
                        i = i+1, summe = summe + i );            //   Veränderung
```

Allerdings macht ein zu extensiver Gebrauch vom Kommaoperator ein Programm gerne unübersichtlich.
Der Kommaoperator ist links-assoziativ: er wird von links nach rechts abgearbeitet. der Wert des Gesamtausdrucks entspricht demnach dem Wert des am weitesten rechts stehenden Ausdrucks:

```
int    i1,  i2,  i3;
double   betrag;
betrag  =  ( i1 = 5 ,  i2  =  i1  *  3  ,   i3  =  i2  %  4);
```

Hier wird zunächst der Wert 5 der Variablen i1 zugewiesen, anschließend 15 (5*3) der Variablen i2, dann 3 (15 % 4) der Variablen i3, und schließlich wird dieser Wert 3 auch der Variablen betrag zugewiesen, als Wert 3.0 .
Eine Gruppe von Anweisungen kann mit dem Klammerpaar { } zu einem Programmblock (oder Block genannt) zusammengefasst werden. Hinter dem abschließendem Zeichen } steht kein Semikolon. Die Klammern { und } stellen eine Methode der Strukturierung von Programmen dar. Am Anfang eines solchen Blockes können Variablen definiert werden, deren

Gültigkeit auf diesen Block und alle evtl. inneren Blöcke begrenzt ist. Dies hat große Vorteile, wann immer eine Variable nur lokal in einem begrenzten Programmabschnitt benötigt wird. Sollte eine andere Variable desselben Namens außerhalb des Blocks aktiv sein, wird diese innnerhalb des neuen Blocks unwirksam. Ein bereits außerhalb des Blocks definierter Bezeichner läßt sich also in einem Unterblock neu definieren.

Ein Beispiel für neue Definitionen in einem inneren Block :

```
main ( )
{
   int   x = 1 ;
    {                                   // Anfang Unterblock
          cout  << "\ n"  << x ;         //  altes  x = 1
           double   x  = 2;             //  neue Definition
           cout  << "\n"    << x ;      //   neues  x = 2
             {                                  //   Anfang    Unter - Unterblock
                 cout  <<"\n"   <<  x;          //    neues  x = 2
             }                                  //    Ende  Unter - Unterblock
      }                                     //  Ende Unterblock
       cout  << "\n"  << x ;               altes  x = 1
       return 0;
}
```

5.2 Arithmetische Operationen

Die arithmetischen Operatoren * / und % haben eine höhere Priorität der Ausführung als die Operatoren + und -, und diese wiederum eine höhere als die Wertzuweisung.

x = a + b * c ; ist also gleichbedeutend wie x = (a + (b * c)) ;

Die normale Reihenfolge der arithmetischen Operatoren ist von links nach rechts :

a + b + c ist gleichbedeutend mit (a + b) + c

Auch der Divisionsoperator / ist **links - assoziativ** :

a / b / c ist gleichbedeutend wie (a / b) / c

Diese Reihenfolge ist z.B. im folgenden Programmteil bedeutsam :

```
int wert;
wert  =  4 * 10000 / 2;
```

Hier wird zunächst 4 * 10000 berechnet, was zu einem Überlauf bei int - Werten führt.

Auch in den folgenden Beispielen kommt es auf die Reihenfolge an :

(1.0 + 1.0e10)-1.0e10--> 0.0 , jedoch gilt : 1.0 + (1.0e10 - 1.0e10) ---> 1.0 !

Der Modulo-Operator % kann nur auf ganze Zahlen (char, int, long) angewendet werden und liefert den Rest bei der ganzzahligen Division. Bei der Division muß der Programmierer dafür sorgen, daß **keine Division durch Null** stattfindet.

Hat eine arithmetische Operation ganzzahlige Operanden, so wird eine ganzzahlige Operation durchgeführt. Ist mindestens ein Operand vom Typ der reellen Zahlen, so wird eine Gleitkommaoperation durchgeführt; wenn nötig, wird der ganzzahlige Operand in ein reelles Format umgewandelt.

Bei ganzzahliger Division entsteht ein ganzzahliger Quotient ohne Stellen hinter dem Komma. Daher wird 5/9 ganzzahlig berechnet und ergibt den Wert 0; dagegen wird 5.0/9 bezw. 5 / 9.0 bezw 5.0 / 9.0 in Gleitkommaarithmetik berechnet und ergibt den Wert 0.555555 . Die Division -10 /3 ergibt -3 .

Ganzzahlige Bereichsüberschreitung wird in C nicht erkannt. Dagegen führen Bereichsüberschreitungen bei reellen Operationen in der Regel zu einem Fehlerabbruch.
In der Sprache C gibt es keinen besonderen Operator für die Exponentiation. Potenzen können allgemein mit der Funktion

pow(double x, double y)

durchgeführt werden, wobei sich als Ergebnis der Wert x hoch y, in double, ergibt.
(mit #include <math.h> zu benutzen).

Verwandte Exponentialfunktionen sind :

exp(double x) liefert e hoch x, vom Typ double,

und

pow10(int x) liefert 10 hoch x, ganzzahlig.

Anstelle einfacher if- else - Anweisungen kan man das **Operatorpaar ? :** einsetzen. Es ist ein dreistelliger Operator (auch **bedingter Ausdruck** genannt) :

var = ausdruckv ? ausdruck1 : ausdruck2 ;

Hier wird ausdruckv bewertet (Ergebnis 0 oder nicht 0) ; Ist der Wert für ausdruckv != 0, dann erhält var den Wert von ausdruck1, andernfalls den Wert von ausdruck2.

var = (a>b) ? a : b; ist also gleich wie var = max(a,b).

Berechnung von ln | x | :

```
float x, y;
if ( x != 0 )
    y = x > 0 ? log( x) : log ( -x );
```

Noch ein Beispiel : der folgende Programmteil gibt den Inhalt einer Speicherstelle k entweder mit einem positiven oder negativen Vorzeichen aus :

```
cout <<"k = " << ( ( k < 0 ) ? '-' : '+' ) << abs(k);
```

Die Zähloperatoren ++ und - - :

In der Regel wird der int Datentyp zum Zählen bei sich wiederholenden Programmabläufen verwendet. Zum Inkrementieren einer Zählvariable gibt es den Operator ++, zum Dekrementieren den Operator - - . Diese Operatoren können auf Zählvariablen angewendet werden :

```
int zaehl;
++zaehl;          //  zaehl  = zaehl + 1
- - zaehl;        //    zaehl = zaehl - 1;
```

Solche Operatoren können vor oder hinter den Variablen stehen. Die Ausführung ist entsprechend unterschiedlich :

```
int neu,  i;
neu  =  ++i;     //  i = i+1;  neu  = i;    Prefix Operator ++
neu  =  i++;     //  neu = i;    i = i+1;   Postfix Operator ++
```

Der Operator - - reagiert genau so .

Allgemein gilt : steht einer dieser Operatoren ++ bezw. - - in einem Ausdruck vor einer Variablen, wird zuerst die Variable entsprechend verändert und dann im Ausdruck weiter verwertet; steht ein Operator ++ bezw. - - hinter einer Variablen, so wird die Variable im Ausdruck erst verwertet, und dann anschließend entsprechend modifiziert .

Beispiel : berechne die Summe der ganzen Zahlen 1 bis 100.

```
i = 100;
while (i > 0 )  summe = summe +  i- - ;
```

Viele for-Anweisungen lassen sich mit solchen Zähloperatoren eleganter formulieren :

for (int i = 0; i < 50; i++) ....

Der Zuweisungsoperator kann entsprechend der folgenden Zusammenstellung mit arithmetischen Operatoren kombiniert werden. Dabei wird die arithmetische Operation vor der Wertzuweisung ausgeführt.

```
x  +=  y ;    wie  x = x + y; wir sagen : addiere y zu x
x   -=  y ;    wie  x = x - y;
x  *=  y ;    wie  x = x * y;
x  /=  y ;    wie  x = x / y;
x %= y ;    wie  x = x % y;
```

In einem Programm sollte eine der beiden Formen der Wertzuweisung konsistent benutzt werden . Die folgenden Wertzuweisungen sind äquivalent :

```
++x;          und          x  += 1;
- - x ;       und          x   -= 1;
```

Für Postfixoperanden gibt es keine entsprechende Äquivalenz !
Die folgende Tabelle faßt die Regeln für Vorrang und Assoziativität aller Operatoren zusammen; sie enthält auch Operatoren, die bisher noch nicht behandelt wurden. Innerhalb einer Zeile haben Operatoren den gleichen Vorrang.

Operatoren	**Assoziativität**
`()  []  ->  .`	von links her
`! ~  ++ --  +  - *  & (type) sizeof`	von rechts her
`*  /  %`	von links her
`+  -`	von links her
`<<  >>`	von links her
`<  <=  >  >=`	von links her
`==  !=`	von links her
`&`	von links her

```
^                                              |  von links her

|                                              |  von links her

&&                                             |  von links her

||                                             |  von links her

?:                                             | von rechts her

= += -=  *=  /=  %=  ^=  |=  <<=  >>=          | von rechts her
,                                              |  von links her
```

Die folgenden mathematischen Standardfunktionen werden mit #include <math.h> dem C-Programm zugeordnet :

Funktion	**Wirkung**	**Beipiel**
abs(i)	|i| Betrag ganzzahlig	j = abs(i);
fabs(x)	|x| Betrag reell	z = fabs(x);
pow(x,y)	x hoch y, x,y reell	z = pow(x, 0.5);
pow10(x)	10 hoch x, x ganz	z = pow10(i);
exp(x)	e hoch x, x reell	z = exp(5.2);
sqrt(x)	Quadratwurzel von x, x >=0 reell	z = sqrt(x);
log(x)	ln x Basis e , x > 0	z = log(x);
log10(x)	log x Basis 10, x > 0	z = log10(u/v);
M_PI	Zahlenwert 3.1415927...	bogen = wink*M_PI/180;
sin(x)	Sinus im Bogen 0 bis 2pi	z = sin(wink*M_PI/180);
cos(x)	Cosinus im Bogen 0 bis 2pi	z = cos(wink*M_PI/180);
tan(x)	Tangens im Bogen 0 bis 2pi	z = tan(wink*M_PI/180);

5.3 Bitoperationen

C verfügt über 6 Bitoperatoren, welche auf int- / char-Operanden angewendet werden können:

&	UND - Verknüpfung von Bits
|	ODER - Verknüpfung
^	Exklusives ODER
<<	Bitverschiebung nach links
>>	Bitverschiebung nach rechts
~	Bit - Komplement

Beispiele : & : unsigned char byte = 0x81,
maske = 0x80 ;
byte = byte & maske ; // Ergebnis : byte = 0x80
// Mit UND-Operation wird ausgewähltes Bit auf 0 gesetzt.

| : unsigned char byte = 0x80,
maske = 0x01;
byte = byte | maske; // Ergebnis: byte = 0x81
// Mit ODER-Operation wird ausgewähltes Bit auf 1 gesetzt.

<< : x << 2 bewegt den Wert von y um 2 Positionen nach links und schiebt Null-Bits nach; dies ist eine Multiplikation mit 4 !

>> : Rechtsshift in Borland C : ist der Operand positiv bezw. vom Typ unsigned, bekommen wir bei jeder Bitverschiebung eine Division durch 2 mit Verlust des Restes. Ist dagegen die Zahl negativ (signed), wird beim Rechtsshift immer von links eine 1 nachgeschoben, so daß man nicht mehr von einer Division durch 2 reden kann.

5.4 Typkonvertierung in Ausdrücken

Im Gegensatz zu vielen anderen Programmiersprachen erfolgt in C keine Typprüfung bei der Manipulation von Daten. So können Datentypen beliebig untereinander kombiniert werden. Aber Vorsicht, bei den Umwandlungen können Rundungsfehler bezw. Umwandlungsfehler entstehen. Innerhalb eines Ausdrucks können verschiedenartige Datentypen verwendet werden:

```
int  zahl;
float  betrag,  summe;
summe  =  zahl  + betrag ;
```

Für jeden Operator (hier +) müssen die beiden Operanden auf einen gemeinsamen Datentyp gebracht werden, bevor die eigentliche Berechnung durchgeführt wird. Im Beispiel wird zunächst der Wert der Variablen zahl in das float - Format umgewandelt, dann erst findet die Addition von zwei float - Werten statt. Vor der Wertzuweisung wird das Resultat des Ausdrucks wenn nötig nochmals umgewandelt : in das Format der **Lvalue-Variablen** (hier summe).

```
int r;   float  s;  double  t,  res;
res  =  r  *  s  +  t;
```

Hier wird zunächst r in float umgewandelt, dann das Produkt r*s als float Wert berechnet; anschließend wird dieser Produktwert in double umgewandelt, um die Addition in double durchzuführen , und schließlich wird der double -Wert der Variablen res zugewiesen. In C ist die folgende Hierarchie für solche **Typkonvertierungen** festgelegt :

long double
double
float
unsigned long
long
unsigned int
int

Bei arithmetischen Operationen werden alle char -, unsigned char-, signed char- und short - Typen unabhängig von ihrer Kombination mit anderen Datentypen zunächst in ein int - Typ verwandelt.

```
float   fvalue1,   fvalue2;

fvalue1  =   40 / 3 ;
```

```
fvalue2   =  40.0/3;
cout    <<" 40/3 ergibt   "   <<  fvalue1   <<"40.0/3 ergibt "  << fvalue2;
```

Die Ausgabe für fvalue1 ist gleich 13.000000 , da bei 40/3 zunächst der ganzzahlige Bruch ermittelt wird : 13 ; dieser Wert wird dann in float umgewandelt und fvalue1 zugewiesen: 13.000000 .

Die Ausgabe für fvalue2 ist gleich 13.333333, da bei 40.0/3 der Wert 3 zunächst in die Gleitkommazahl 3.0 umgewandelt und dann erst der Dezimalbruch berechnet und als float value der Variablen fvalue2 zugewiesen wird.

Die folgende Zusammenstellung zeigt die grundsätzlichen Regeln bei der Typumwandlung innerhalb eines Ausdrucks :

unsigned char --> int,long unsigned int --> long	Die höherwertigen Bytes im neuen Typ sind immer Null.
char --> int,long int --> unsigned long, long	Das Vorzeichen wird in die höherwertigen Bytes propagiert.
int, long --> float, double, long double	Dezimalen können abgeschnitten werden !
float --> double, long double double --> long double	Es werden Nullen angehängt, der Wert bleibt erhalten.

Soweit wie möglich sollte der Programmierer mit gleichem Datentyp in einem Ausdruck arbeiten. Bei ganzzahligen Datentypen wird ein Datenüberlauf bei Berechnungen bezw. Umwandlungen nicht angezeigt.

Während innerhalb eines Ausdrucks alle Werte der beteiligten Variablen immer auf den hierarchisch höchsten Typ umgewandelt werden, gibt es bei der **Zuweisung** des Wertes eines Ausdrucks an die Ergebnisvariable (Lvalue - Variable) praktisch alle Möglichkeiten der Typumwandlung.Dies führt zu der folgenden Ergänzung der obigen Zusammenstellung :

long double --> double, float double --> float	Wert wird auf 16 bezw. 7 gültige Ziffern gerundet.
long,unsigned long-->int,unsigned int	Linkswertige Bits werden abgeschnitten, der Wert kann sich ändern.
float,double,long double-->int,long	Dezimalen hinter dem Komma weggelassen: ganzzahliger Wert bleibt.

Neben der bisher besprochenen **impliziten Typumwandlung** gibt es in C auch die Möglichkeit der **expliziten Typumwandlung** mittels des **cast-operators :**

(Datentypbezeichner) Ausdruck

Als Datentypbezeichner werden vordefinierte Datentypen wie int, double, verwendet.

```
int  i=3,   j=6;
double  d;
d  =  i / (double) j;        // Ergebnis d = 0.5
```

Noch einige Beispiele :

20 / 3	hat den Wert	6
(float) 20 / 3		6.333333
20.0/ 3		6.333333
(float)(20 / 3)		6.0

In C++ gibt es für den cast - Operator sowohl diese C- Notation, als auch eine funktionelle Notation :

```
float (20)
float (20/3)    usw.
```

5.5 Aufzählungskonstanten mit enum

Eine Aufzählung ist eine Folge von ganzzahligen symbolischen Konstanten vom Typ int.
Die grundsätzliche Deklaration sieht wie folgt aus :

```
enum   {  FALSCH,  WAHR  } ;
```

Der erste Name in einer solchen enum - Liste hat den Wert 0, der nächste 1, und so weiter, wenn kein expliziter Wert angegeben wird. Wir können aber die Werte individuell zuweisen :

```
int  monat;
enum  {JAN = 1,  FEB,  MRZ,  APRL  };     // FEB  hat  den Wert 2, usw.
....
monat  =  MRZ;
....
if  (monat  == APRL) ...
```

enum Bezeichner sind Konstanten.
In der folgenden Vereinbarung erhält jedes Symbol den explizit festgelegten int - Wert :

```
enum   { ROT = 3,  GRUEN = 5,  GELB  = -1  };
```

Solche Aufzählungssymbole sind auch eine Alternative zu #define : sie können in C in jedem Konstantenausdruck verwendet werden, beispielsweise auch nach case in switch- Anweisungen. Während allerdings die über #define vereinbarten Texte über das ganze Quellprogramm gültig sind, haben Aufzählungskonstanten einen Gültigkeitsbereich, der dem von normalen Variablen entspricht : innerhalb von umfassenden Blöcken hat ein lokal definierter Namen Gültigkeit.
Aufgrund dieser besonderen Alternativen zu #define schreiben wir solche Symbole immer groß; dadurch werden sie auch unterschieden von den durch const festgelegten Namen.

Die enum-Deklaration läßt sich auch noch dadurch erweitern, daß wir in der enum – Deklaration gleichzeitig noch Variablennamen angeben :

```
enum    { FALSCH,  WAHR } variable1, variable 2 ;
```

Damit sind auch noch zwei Variablen vom Typ enum vereinbart; beide Variablen sind vom Typ int mit dem Wertebereich FALSCH und WAHR (für 0 und 1). Wir können nun formulieren :

```
variable1  = FALSCH;
.....
```

if (variable2 == WAHR) ...

Zusätzlich kann man mit enum noch eine **Datentypdefinition** verbinden :

enum beurteilung {SEHR_GUT = 1, GUT, BEFRIEDIGEND) note1, note2;
enum beurteilung note3; // in C

Hier ist enum beurteilung ein neu definierter Datentypname, die Variablen note1, note2 und note3 haben alle den zwischen den Klammernsymbolen { und } festgelegten Wertebereich.
In C wird hier der Name beurteilung als Etikett ('tag') des neuen Datentyps enum beurteilung bezeichnet.
In C++ ist beurteilung selbst der neue Datentyp, hier können wir formulieren :

beurteilung note4; // in C++

6.) Eingabe von Konsole und Ausgabe am Bildschirm

6.1 Eingabe und Ausgabe mit cin, cout (ref. Beispiel 7)

Für die Ein-/Ausgabe von Daten kann in C++ Versionen eine Bibliothek verwendet werden, die mit #include <iostream.h> zugeordnet wird. Die älteren Funktionen in den Bibliotheken stdio und conio bleiben erhalten (siehe Kapitel 6.2).
Für die **Ausgabe auf der Konsole** werden entsprechend dem Datentyp voreingestellte Formate (Umwandlungsverfahren) verwendet. Der Ausgabeoperator << läßt sich aneinanderreihen :

cout << Ausdruck_1 << Ausdruck_2 << Ausdruck_3 ;

Die Bewertung der Ausdrücke beginnt (hier) rechts mit dem Ausdruck_3; auf der Bildschirmausgabe erscheinen die Ausdrücke von links nach rechts beginnend mit Ausdruck_1.

Anstelle eines Ausdrucks kann auch einer der folgenden **Manipulatoren** verwendet werden ; diese führen entweder spezielle Ausgabeoperationen durch, oder steuern die nachfolgende Ausgabe :

Manipulator	**#include**	**Auswirkung**
dec	<iostream.h>	dezimale E/A (voreingestellt)
hex	<iostream.h>	hexadezimale E/A
endl	<iostream.h>	Zeilenvorschub cr, wie "\n"
setw(n)	<iomanip.h>	folgende Ausgabe mit n Stellen Breite.
setprecision(n)	<iomanip.h>	alle reellen Ausgaben mit n Nachpunktstellen.

Die Ausgabebreite nach setw(n) gilt nur für die anschließende Ausgabeoperation, sie wird danach wieder auf den Vorgabewert 0 zurückgesetzt. Reicht die Feldbreite nicht aus, was bei 0 immer der Fall ist, so werden nur die Ziffern ohne führende oder folgende Nullen ausgegeben.
Die Bibliothek iostream erlaubt auch noch die folgenden **speziellen Funktionsaufrufe :**

Ergebnis	Funktion	Aufgabe der Funktion

```
int      cout.width(n)   Folgende Ausgabe n Stellen breit
int      cout.precision  Alle reellen Ausgaben mit n Nachpunktstellen
int      cin.get()       Liefert das nächste Zeichen und entfent es.   (cout.put ( c )    )
int      cin.peek()      Liefert das nächste Zeichen, das im Puffer bleibt.
int      cin.fail()      Liefert Fehlerstatus: == 0 kein Fehler
                                               != 0 Eingabefehler
int      cin.eof()       Liefert Endestatus : == 0 kein Ende
                                               != 0 Endemarke ^Z
         cin.clear()     Setzt Fehler- und Endestatus zurück.
         cin.ignore(n,c) Überliest n Zeichen, leert Puffer  bis Zeichen c
-----------------------------------------------------------------
```

In der voreingestellten Formatierung erscheinen bei der Ausgabe von Zahlen nur die Ziffernfolgen, Leerzeichen sind als Texte oder Zeichen einzufügen.
Bei der Ausgabe von Tabellen sind entsprechend der Größe der auszugebenden Zahl führende Leerzeichen voranzustellen . Das folgende Beispiel formatiert die Ausgabe von x auf eine Feldbreite von 8 Stellen mit 2 Nachkommastellen :

```
#include   <iostream.h>
#include  <iomanip.h>
main
{
      double  x  =  12.3456789;
      cout.precision(2);              //     2  Nachkommastellen für alle reellen Ausgaben
      cout   <<  "\nWert = "   << setw(8)   <<  x     << endl;
 }
```

In einer neuen Zeile erscheint nun : Wert =   12.34
(rechtsbündig, auch bei Textausgabe)

Auch für die **Eingabe von der Konsole** werden entsprechend dem Datentyp voreingestellte Formate verwendet. Der Übernahmeoperator >> läßt sich ebenfalls aneinandereihen :

cin >> Bezeichner ;
cin >> Bezeichenr_1   >> Bezeichenr_2;

So können auch mehrere Werte, in der Eingabe durch Leerzeichen getrennt, eingelesen werden. Die eingegebenen Werte erscheinen auch auf dem Bildschirm. In einfachen Anwendungen sollte man mit cin nur einen Wert einlesen. Zusammen mit dem abschließendem Wagenrücklauf cr gelangen die Eingabezeichen in einen Pufferspeicher und werden dort entsprechend dem Datentyp ausgewertet. Alle Whitespace-Zeichen (Leerzeichen sowie Wagenrücklauf, Zeilenvorschub, Tabulator) werden überlesen. Die Eingabe von Zahlen erfolgt mit oder ohne Vorzeichen. Die Auswertung endet mit dem ersten Zeichen, das nicht zur Auswertung gehört.
Auch bei der Eingabe von Zeichen für den Datentyp char werden bei cin >> .. Whitespace-Zeichen überlesen, das erste Nicht-Whitespace- Zeichen wird als Ergebnis gespeichert. Im Gegenatz dazu liefert die Funktion **cin.get()** das nächste Zeichen der Eingabe, auch ein Whitespace-Zeichen. Tritt ein Eingabefehler auf oder werden mehr Daten als erforderlich eingegeben, so verbleibt der Rest der Eingabezeile zusammen mit dem Endezeichen cr im Eingabepuffer und wird bei der nächsten Eingabe ausgewertet. War die Eingabezeile nicht länger als 80 Zeichen, und wurde mit einem Wagenrücklauf die Eingabe abgeschlossen,, so kann der

Eingabepuffer mit **cin.ignore(80,10);** gelöscht werden, so daß die nachfolgende Eingabe einen leeren Puffer vorfindet.
Eine Eingabe von Buchstaben anstelle von Ziffern, bei Zahlen, bewirkt einen **Fehlerzustand.** Dieser Fehlerzustand kann mit dem Funktionsergebnis von **cin.fail()** ungleich 0 erkannt werden. Der Fehlerzustand muß mit einem Aufruf von **cin.clear()** wieder zurückgesetzt werden, da er sonst die gesamte Eingabe blockiert :

Ein Beispiel :

```
double   r,  s;
cout  << "\nReellen Wert  --> ":   cin  >>  r;
if ( cin.fail()   != 0 )                    // bezw. if  ( cin.fail()   ) ...
   {     cin.clear();     cin.ignore( 80, 10 );        cout  <<"??";
    }
cout   <<"\nReellen Wert  -->";
if   (cin   >>  s    == 0)                          //   >> rangiert vor  ==
    {     cin.clear();     cin.ignore(80, 10 );   cout   <<  "??";
     }
```

Die Tastenkombination STRG und Z (^Z) wird als Endemarke von Leseschleifen verwendet und kann mit dem Funktionsergebnis cin.eof() ungleich 0 erkannt werden. Auch diese EOF - Marke ist mit cin.clear wieder zurückzusetzen.

6.2 Formatierte Eingabe und Ausgabe mit printf, scanf (ref. Beispiel 8)

Zur formatierten Eingabe und Ausgabe dient die Bibliothek stdio. Die **formatierte Ausgabe** geschieht mit der Funktion

printf(" Formatangaben ", Liste von Ausdrücken);

Ein Beispiel :

```
int   a  = 1,  b  = 5;
printf( "\n%i  +   %i   =  %i",   a ,  b,   a+b );
```

Die Formatangaben stehen zwischen Hochkommazeichen in einer Textkonstanten. Sie bestehen aus Steuerzeichen (z.B. \n) , Texten und Umwandlungsvorschriften wie z.B. %i für int - Zahlen. Jede Umwandlungsvorschrift entspricht in der Reihenfolge einem Ausdruck, wobei die einzelnen Ausdrücke durch Kommas zu trennen sind. Für jeden Ausdruck ist also eine durch % eingeleitete Umwandlungsvorschrift erforderlich.
Alle Zeichen in der Formatangabe, die weder zu einer mit \ beginnenden Escape-Sequenz, noch zu einer mit % beginnenden Umwandlungsvorschrift gehören, werden als Text zwischen den umgewandelten Zahlenwerten ausgegeben.
Fehlen Umwandlungsvorschriften, so werden die entsprechenden Ausdrücke nicht ausgegeben. Überflüssige Umwandlungsvorschriften resultieren in zufälligen ausgegebenen Werten. Die printf-Funktion hängt an das Steuerzeichen \n (Zeilenvorschub) automatisch noch das Zeichen \r für den Wagenrücklauf an, so daß auf den Anfang der nächste Zeile gegangen wird. Die Formate für Eingabe und Ausgabe simd im folgenden zusammengestellt :

Datentyp	**Eingabeformat(scanf)**	**Ausgabeformat(printf)**
unsigned char	%c (nur Zeichen)	%c(Zeichen) %u %x(Zahl)
unsigned int	%u (%x für hex)	%u (%x für hex)

```
unsigned long  %lu ( %lx für hex)    %lu       (%lx für hex )

signed char    %c  (nur Zeichen)     %c(Zeichen)  %i %d(Zahl)

int,short int  %d  %i  (auch hex)    %d  %i     (%x  für hex)
long,long int  %ld %li (auch hex)    %ld %li    (%lx für hex)

float          %e  %f  %g  %E  %G        %e  %f  %g  %E  %G
double         %le %lf %lg %lE %lG       %le %lf %lg %lE %lG
long double    %Le %Lf %Lg %LE %LG       %Le %LF %Lg %LE %LG
---------------------------------------------------------------
```

Bei printf() ist besonders wichtig, daß die Formatbeschreibungen im Formatstring auch wirklich mit den Typen und der Reihenfolge der danach folgenden Ausgabekonstanten und Variablen übereinstimmen.
Bei der Ausgabe von reellen Zahlen in der Formaten %f, %lf, %LF erscheint der Wert mit Stellen vor und hinter dem Dezimalpunkt. Die Formate %e, %le, und %Le arbeiten in der Mantisse-Exponent Darstellung mit einer Vorpunktstelle.
Bei der Ausgabe in den Formaten %g, %lg und %LG richtet sich die Darstellung nach der Größe der Zahl.
Bei der Ausgabe einer Zahl innerhalb eines Textes verwendet man vorzugsweise %i und %lg (double) . Für die Ausgabe von **Zahlentabellen** ist es nötig, Angaben über die Breite des Feldes und gegebenenfalls die Anzahl der Nachpunktstellen zu machen ; sie stehen unmittelbar hinter dem % Zeichen :

%5i (rechtsbündige Ausgabe in Feld der Breite 5)
%10.2lf (rechtsbündige Ausgabe in Feld der Breite 10, mit 2 Nachpunktstellen)

Die Funktion printf() gibt normalerweise die Zahl der ausgegebenen Bytes zurück.
Die formatierte Eingabe geschieht mit der Funktion

scanf("Formatangaben", Variablenliste);

Die Formatangaben stehen üblicherweise zwischen Anführungszeichen in einer Textkonstanten. Die Bezeichner der Variablen sind durch ein **vorangestelltes & als Zeiger** zu kennzeichnen (Adressen der Variablen) :

```
int  x,  y;
scanf("%i  %i",  &x,  &y);        //  Formatangaben  für int
```

Fehlen Formatangaben, so erhalten die entsprechenden Variablen keine neuen Werte zugewiesen.
Die Formatangaben in scanf werden üblicherweise durch ein Leerzeichen getrennt; **hinter der letzten Angabe darf kein Leerzeichen mehr stehen,** da sonst bei der Eingabe ein weiteres Datenfeld erwartet wird.
Bei der Eingabe von double -Größen muß in der Formatangabe l mit verwendet werden.

Eine Eingabezeile ist erst gültig, wenn sie mit cr abgeschlossen ist. Die Zahlen der Eingabezeile werden durch ein Leerzeichen oder Tabulatorzeichen getrennt. Werden weniger Daten als erwartet eingegeben, so fordert das System mit dem Cursor weitere an; werden mehr eingegeben, so verbleiben die überzähligen in der Eingabdatei stdin im Puffer und werden beim nächsten scanf - Aufruf ausgewertet. Der die Eingabe abschließende Wagerücklauf cr verbleibt immer im Pufferspeicher, stört aber bei der nachfolgenden Eingabe von Zahlen nicht mehr. Die Eingabedatei stdin kann mit **fflush(stdin) oder reset(stdin)** geleert werden. Dies bereinigt die Eingabe von überzähligen Daten.

Häufig wird eine Eingabe wegen Eingabefehler ohne Fehlermeldung abgebrochen. Dabei können unendliche Schleifen entstehen, oder es wird mit alten oder zufälligen Werten weitergerechnet. Für eine Fehlerkontrolle liefert die scanf-Funktion die Anzahl der richtig eingelesenen Parameter als Funktionsergebnis zurück :

Anzahl = scanf ("Formatangaben", Variablenliste);

Ein Beispiel für die Kontrolle des Fehlerfalls :

```
int  x  eins;
eins =  scanf("%i",  &x);
if  (eins  != 1 )  printf("Eingabefehler ");
```

Bei der Eingabe von Strg und Z (^Z) anstelle von Daten liefert die scanf-Funktion den vordefinierten Wert EOF (meist -1). Damit lassen sich Leseschleifen steuern :

```
while  (  scanf("%i",  &x )   !=  EOF )
{
    Anweisungen
}
```

Insbesondere bei der fehlerhaften Verwendung eines Eingabe- oder Ausgabeformats können ungeprüfte Fehler entstehen, wobei sich dann Programme wie eben gezeigt sehr hilfreich erweisen.

In stdio.h sind auch die beiden Funktionen getchar() und putchar(ch) deklariert.

Die Funktion **getchar()** hat keinen Parameter, sie liest ein Zeichen von der Standardeingabe (Konsole) stdin und stellt es über den Return-Wert zur Verfügung :

```
char zeichen = getchar();          // Aufruf von getchar()
```

Die Eingabe wird mit Return abgeschlossen. Das unmittelbare Drücken der Return - Taste wird als line feed = 0x0A zurückgegeben, bei Eingabe der Taste ^Z (Taste F6) ist der Returnwert EOF (-1), hexadezimal 0xFF .
Die Funktion **putchar (ch)** gibt das Zeichen ch an die Standardausgabe stdout aus. Der Rüchgabewert ist das Zeichen selbst, bei fehlerhafter Ausgabe wird EOF (-1) zurückgeliefert.

Für die Eingabe von einzelnen Zeichen verwendet man auch die in **<conio.h>** definierten Funktionen

Zeichenvariable = getch();
Zeichenvariable = getche();

Beide Funktionen warten auf das Betätigen einer Taste. getch() liefert den Code ohne Echo zurück, getche() mit Echo. Beide Funktionen können auch ohne Wertzuweisung aufgerufen werden, um ein Programm bis zum Betätigen einer Taste warten zu lassen.
Ein Beispiel :

```
#include   <stdio.h>
#include    <conio.h>
#include    <math.h>
main()
{
    double   radikand;
     do
      {
          printf("\nEingabe von Radikand    : ");  scanf("%lg",  &radikand);
          printf(" Wurzel  aus  %lg   =  %lg",  radikand,  sqrt(radikand)   );
```

```
            printf(" \n Weiter ?     n = nein  ->");
      }  while  ( getche()   !=  'n');                  // Antwort lesen und auswerten .
      return  0 ;
}
```

Es gibt übrigens auch eine **leere Anweisung,** wenn Sie nämlich nur ein Semikolon in eine Zeile setzen. Solche Konstruktionen werden beispielsweise in Schleifen benutzt, bei denen im Schleifenkörper nichts passieren soll, weil alle Aktionen schon im Ausdruck des Schleifenkopfes stattfinden:

```
double  nc;                                    // nc ist number of characters
for (nc = 0;  getchar()  !=  EOF;  ++nc)     // die eingegebenen Zeichen werden gezählt.
      ;                                        // leere Anweisung
printf("%.0f\n", nc );
```

in printf sorgt %.0f dafür, daß die Ausgabe keinen Dezimalpunkt und keine Dezimalstellen enthält, da sie Null sind.

7.) Felder

7.1 Eindimensionale Felder (ref. Beispiele 9, 10)

Große zusammenhängende Datenmengen wie Matrizen oder Messreihen werden bei vielen technischen Anwendungen verwendet. Ein einfaches Beispiel ist die Aufgabe, aus mehreren Meßwerten den Mittelwert zu bilden und von jedem Wert die Abweichung vom Mittelwert auszugeben.
In C kann man wie in anderen Programmiersprachen mehrere Daten gleichen Typs zu einem **Feld (Array)** zusammenfassen :

```
float  r[80] ;          //  Feld  aus 80  float - Zahlen.
```

In eckigen Klammern teilt man dem Compiler mit, wie groß das Feld sein soll, d.h. wieviele Elemente des angegebenen Typs in dem Array Platz haben.
Solche **eindimensionale Felder** bezeichnet man auch als **Vektoren** oder **indizierte Variablen.**
Die Elemente eines Feldes werden durch die Angabe der Indexposition in eckigen Klammern angesprochen :

```
r[0]                    r[24]
```

Der Zugriff auf ein Array-Element erfolgt hier durch die Angabe des Arraynamens und der Position in eckigen Klammern; diese Art des Zugriffs nennt man **Indizieren.** Als Indexwerte dürfen nur ganzzahlige Werte vom Typ int verwendet werden.
Die Indexposition ist hier ein ganzzahliger Ausdruck von [0] bis [79]. C benutzt als ersten Index immer die Null. Die maximale Indexposition ist immer um 1 kleiner als die bei der Deklaration festgelegte Elementanzahl: ein Array der Größe N hat also einen Index von 0 bis N-1.
Reelle Größen als Index werden entsprechend der automatischen Typumformung ganzzahlig gemacht, Stellen hinter dem Komma werden abgeschnitten. Als Index können auch arithmetische Ausdrücke verwendet werden. **Der Indexbereich wird weder bei der Übersetzung noch bei der Ausführung überwacht !!**
Ein Beispiel :

```
 int    i,  j[5];
double  x[100],    ri  = 4.99,  y;
x[0]   =  47.11;
for  ( i=0;  i < 100;  i++ )
      x[i]   =  0.1  * i ;
```

```
i  = 1;
x[i-1 ] = 47.12 ;                    //    Indexrechnung  x[0]
j[0]    =   10;
x[ j[0]  ]    =  47.13 ;             //  Indexschachtelung   x[10]
x[ ri ]  =  47.14  ;                 //   reeller Index    x[4]

y  =  (  x[14] + 6.88 )  *  x[99];          // Allgemeiner Ausdruck
```

Oft wird die Anzahl der Feldelemente durch eine Symbolkonstante festgelegt :

```
#define   N  5
main()
{    double   x[N];      ....
}
```

Bei der Deklaration lassen sich Felder auch mit Anfangswerten vorbesetzen :

```
double   p[5]   =  {  4.1,   4.2,   4.3,  4.4,   4.5  };
```

Enthält die Liste weniger Elemente als vereinbart, so werden die letzten Elemente nicht besetzt und haben je nach Speicherklasse entweder einen zufälligen Wert (lokal) oder den Wert 0 (global , static). Enthält die Liste zuviele Elemente, so erfolgt eine Fehlermeldung vom Compiler.
Wenn Sie eine Initialisierung vornehmen und bei der Deklaration den Größenwert in der Klammer weglassen, dann ermittelt der Compiler selbst aus der Anzahl der Initialisierungselemente, wieviel Platz benötigt wird :

```
int  a[ ]  =  { 1, 2, 3,  4, 5 } ;
```

Ein **char - Vektor (Stringvariable)** kann so dimensioniert werden, daß er genau die gewünschten Zeichen aufnehmen kann :

```
char  nachricht[7 ]    =  {'F',  'e',  'h',  'l',  'e',   'r' ,  '\0' };
```

Oder auch so :

```
char nachricht[ ]      =   {'F',  'e',  'h',  'l',   'e',   'r',   '\0\' } ;
```

Oder auch einfach so :

```
char nachricht[7]      =  "Fehler";   // abschließendes '\0' - Zeichen
                                      // wird vom System am Stringende eingefügt.
```

Oder ganz einfach so :

```
char  nachricht[ ]       =  "Fehler";  // Das System errechnet die Feldgröße.
```

Solche char - Vektoren werden zur Verarbeitung von Zeichenketten viel verwendet, insbesondere deshalb, weil C keinen eigenen Datentyp für solche Strings hat.

<u>Ein Beispiel : Mittelwert und Abweichung berechnen</u>

```
#include  <iostream.h>
#include  <iomanip.h>
#define  N  5                          // Anzahl der Elemente
main()
{
     int i;
     double   x[N],  summe = 0,  xmittel;
     /*  Feld  mit   N Elementen  einlesen und aufsummieren      */
    cout  <<"\nFeld aus  "       << N        <<"  Werten eingeben  \n";
    for  (i  = 0;  i <  N  ;   i++ )
```

```
        {   cout << "x["    << i     <<"] -> ";
            cin >> x[i];  cin.ignore(80,10);
            summe += x[i];                                  // aufsummieren
        }
     xmittel = summe / N;                                   // Mittelwert berechnen
    cout  "\nSumme = " << summe   <<",  Mittelwert = "      << xmittel;
    cout <<"\n\n              Wert              Abweichung";
     /*  Wert und Abweicung vom Mittel ausgeben       */
    for (i = 0;  i < N;  i++ )
          cout <<"\n"  << setw(10)  << x[i]  << setw(12)   << x[i] - xmittel;
}
```

Zu beachten ist in diesem Beispiel in der for-Anweisung die Bedingung i < N ;
eine Formulierung i <= N würde zu einem Bereichsüberlauf führen.
Bei der Deklaration eines Feldes muß für die Dimension ein Konstantenausdruck stehen.
In C++ können wir formulieren :

```
const int max = 50;
int  a[max] ;
```

In C ist dies nicht möglich, wir müssen hier formulieren :

```
#define  max  50
...  ...
 int  a[max];
```

Dies ist ein Beispiel dafür, daß in C die const Deklarationen weniger nützlich sind als in C++ : C ist stärker vom Preprozessor abhängig.

7.2 Mehrdimensionale Felder (ref. Beispiel 11)

Für zweidimensionale Anordnungen **(Matrizen)** verwendet man den Doppelindex :

```
double  x[2]  [3];       // Felddeklaration
```

In der Mathematik beschreibt der erste Index die Zeile und der zweite die Spalte. Entsprechend liegen auch im Speicher die Elemente zeilenweise :

```
x[0] [0]        x[0] [1]        x[0] [2]
x[1] [0]        x[1] [1]        x[1] [2]
```

Beispiel : Nullsetzen eines zweidimensionalen Feldes mit zwei verschachtelten for-Schleifen.

```
int  i,  j ;
```

```
double   wert [10]  [50];
for (i = 0;  i < 10;  i++)
      for (j = 0;  j < 50;  j++ )
            wert[i] [j]  = 0;
```

Die Größen für die Anzahl der Elemente können auch mit leicht veränderbaren Symbol- konstanten festgelegt werden :

```
#define  NZ  10
#define NS  50
double  wert[NZ][NS];
```

Bei der Definition von vorbesetzten Feldvariablen stehen die Werte entweder in einer Gesamtliste (zeilenweise abgespeichert !) oder sie werden zeilenweise auf Teillisten aufgeteilt :

```
double  a[2][3]  =  {1, 2,  3,  4,  5,  6 };
double   b[2][3]  =  {  {1,  2,   3 },  { 4,  5  6  }  };
```

Beim Lesen von Matrizen lohnt sich die Eingabe der Werte einer Matrixzeile auf einer Eingabezeile .

<u>Beispiel für zeilenweise Eingabe und Ausgabe :</u>

```
#include  <iostream.h>
#include  <iomanip.h>
#define  NZ  3                                  //  3  Zeilen
#define  NS  4                                  //  4 Spalten
main()
{     int   i,  j ;
       double    x[NZ][NS];
        cout   <<  "\nMatrix zeilenweise eingeben\n";
        for  (i  = 0;  i < NZ;  i++)
            {    cout <<  i+1    <<".Zeile :     "  << NS   << "  Spaltenwerte  -> ";
                  for (j = 0;  j  < NS;  j++)     cin  >>  x[i] [j] ;
                  cin.ignore( 80, 10);
            }
        cout  <<   "\nMatrix zeilenweise ausgeben \n                    ";
        for  ( j = 1;  j  <= NS;  j++)  cout  <<  setw(3)    << j    <<  ".Spalte";
        for  (i = 0;  i < NZ;  i++)
              {    cout  << "\n"   << i+1     << ".Zeile:   ";
                    for  ( j = 0;  j <  NS;  j++)    cout   <<  setw(5)    <<  x[i] [j]   << "     ";
              }
}
```

<u>Ausgabe am Bildschirm :</u>

```
Matrix zeilenweise eingeben
1.Zeile:          4   Spaltenwerte -> 11  12  13  14
2.Zeile           4   Spaltenwerte -> 21  22  23  24
3.Zeile           4   Spaltenwerte -> 31  32  33  34
Matrix zeilenweise ausgeben
                     1.Spalte       2.Spalte       3.Spalte       4.Spalte
1.Zeile               11               12               13               14
2.Zeile               21               22               23               24
3.Zeile               31               32               33               34
```

7.3 Zeichenvektoren, die Funktionen gets() und puts() (Beispiel 12)

Texte werden allgemein als **Strings** oder **Zeichenketten** oder auch als**Zeichenvektoren** bezeichnet. In C ist kein eigener Datentyp für solche Strings vordefiniert, daher werden hier Strings als Felder (Arrays) aus Zeichen vereinbart :

```
char  Stringbezeichner[Größe]  ;
char   Stringbezeichner[Größe]  =  "Text";
 char  Stringbezeichner[ ]          = "Text";
```

Im Gegensatz zu einem normalen Feld wird der Text durch eine zusätzliche Endemarke '\0' (‘\0‘ = 0x00) im Speicher abgeschlossen. Das zusätzliche Byte ist bei der Dimensionierung des Feldes sowie bei der Indizierung einzelner Stellen im Feld zu berücksichtigen.
Eine Textkonstante " Peter" erhält vom Compiler zusätzlich die Endemarke '\0' und wird ohne die Begrenzungszeichen " im Speicher abgelegt. Als einzelne Textzeichen sind alle Zeichenkonstanten einschließlich der Escape-Sequenzen zugelassen. Strings können sowohl durch Adressierung ihrer Elemente als auch durch ihren Bezeichner in einer Schleife bearbeitet werden.

```
int  k;
char  r[5]  =  "1234“;                          //  Stringvariable
cout  << "\nText = "  << r;                     // Ausgabe :   Text = 1234
for ( k=0; k < 5; k++)
    cout  << "\n["  << k  <<"] Zeichen : "  << r[k]    << " Zahl :  "  <<  (int)  r[k] ;
      // Ausgabe : 5 Zeilen wie folgt :  [0]    Zeichen: ‘1‘     Zahl: 49
      //                                 [1]    Zeichen: ‘2‘     Zahl: 50
```

Die iostream-Ausgabe mit cout läßt sich auch auf Strings anwenden, die bis zur Endemarke ausgegeben werden. Bei der Eingabe mit cin beginnt die Speicherung mit dem ersten Nicht-Whitespace-Zeichen und wird mit dem nächsten Whitespace-Zeichen abgebrochen. Daher ist cin für die Eingabe von Texten, die Leerzeichen enthalten, nicht verwendbar.
Bei der Eingabe mit der Funktion cin.getline werden als Parameter eine Stringvariable (Zeiger auf ein char-Feld), die maximale Anzahl der Zeichen und das Endezeichen übergeben, bis zu dem die Eingabe ausgewertet werden soll. Übergibt man \n als Endemarke, so werden auch Leerzeichen gespeichert :

```
cout   <"\Eingabe von Text mit Leerzeichen :  ";
cin.getline(r,  80, '\n') ;
cout  << "Kontrollausgabe --> "  <<  r;
```

Eine Eingabe von **Anita Koller** ergibt die gleiche Ausgabe **Anita Koller**, mit den Leerzeichen.
Mit printf lassen sich Strings mit der Formatangabe %s zusammen mit Zahlen und Zeichen ausgeben.
Bei der Eingabe (mit stdio) ist es zweckmäßig, nicht scanf (mit %s) , sondern die **Funktion gets (String-Variable)** zu verwenden, wobei die Texteingabe durch cr beendet wird. Entsprechend kann auch für die Ausgabe die **Funktion puts(String-Variable)** verwendet werden.
Für gets gibt man als Argument den Namen des Feldes an, wohin der Text gespeichert werden soll. Dieser Feldname kennzeichnet auch die Anfangsadresse dieses Feldes. Der Rückgabewert von gets ist wiederum diese Anfangsadresse des Feldes. Im Fehlerfall oder bei Eingabeende ist der Rüchgabewert Null.
Für puts gibt man als Argument den Namen des char-Feldes an, dessen Inhalt (Text) durch puts ausgegeben werden soll. Statt eines solchen Feldnamens kann auch eine Textkonstante als Argument stehen : puts(" Ausgabetext"); . Im Fehlerfall ist der Rüchgabewert EOF (-1), ansonsten ein positiver Wert.

Beispiel für gets und puts :

```
#include  <stdio.h>
main()
{    char  s[81];                    //   maximal  80 Zeichen  + 1 Zeichen für Stringende  '\0'
     while(  gets (s)   !=  NULL )   //   NULL bei Eingabeende  (Null-Zeiger  vordefiniert )
              puts( s ) ;            //  gelesene  Zeile wird ausgegeben.
}
```

Noch ein Beispiel für gets, mit atof (atoi) :

```
#include <stdio.h>
main()
  {  double km;  char string[128];
     printf("Wie viele km ")  gets(string);
     km = atof(string);  // Funktion   double atof(const char * str);
                         // Funktion     int     atoi( const char * str);
     printf("kilometer = %f\n", km);
  }
```

Operationen mit Strings lassen sich nur über die Feldelemente der Stringvariablen oder mit vordefinierten Funktione (#include <string.h>) durchführen.
Beispiel : Funktion **strcpy(Ziel,Quelle)** :

```
#include  <string.h>
main()
{
    char   s[81];                    // Stringvariable für 80 Zeichen
    strcpy(s,  "Firma ");           //   Funktion kopiert String  "Firma"  nach Feld  s
    cout  <<  s;                     // Ausgabe der Stringvariablen
}
```

Die einfache Operation der Wertzuweisung mit s = "Firma" ; ist in C nicht definiert.

Die Funktion **strcmp(String_1, String_2)** vergleicht die beiden Strings : Das Vergleichsergebnis ist Null bei Gleichheit, sonst größer oder kleiner Null
(zeichenweise verglichen, ASCII Tabelle).

Die Funktion **strcat(Ziel,Quelle)** hängt den String Quelle an den String Ziel an : der String Ziel wird dabei
um die Länge des Strings Quelle größer.

Die Funktion **strlen(String)** gibt die Länge des angegebenen Strings als int zurück. Dabei wird das Stringendezeichen ('\0') nicht mitgezählt.
Das folgende Beispiel zeigt Leseschleifen für Texte:

```
#include <iostream.h>
#include  <string.h>
#define  N  81                          // Länge der Textzeile,  Feldgröße
#define  ENDE  "fertig"                 // Endemarke  für Eingabe
main()
{
  const char  ein[] = "\nEingabe -> ",  aus[] = "Ausgabe  => ";  // Textkonstanten ein und  aus
  char  text[N];                                                 // Stringvariable   text
```

```
  cout  << "\nLeseschleife  Ende mit  " <<  ENDE   << endl;  // ENDE  ist hier ein Makroname
  while (cout <<  ein,  cin.getline(text, N,  '\n'),
              strcmp(text,  ENDE)    )      // Vergleichsfunktion strcmp ergibt 0 bei Gleichheit !
           cout  << aus     << text ;

   cout  << "\n\nLeseschleife Ende mit Strg und Z";
   while ( cout << ein,  cin.getline(text, N,  '\n'),
                          !cin.eof()   )        //   cin.eof liefert  !0 bei Eingabe von    Strg und Z
        cout << aus   << text;
  cin.clear;                                        //  cin.clear löscht Endbedingung  eof
}
```

8.) Strukturen

8.1 Einfache Strukturen (ref. Beispiele 13 , 14)

Bei vielen Anwendungen müssen Elemente verschiedener Datentypen unter einem Namen (Bezeichner) zusammengefasst werden, insbesonders im Zusammenhang mit Datendateien .
In C ist dies möglich durch den zusammengesetzten Datentyp **struct.** Die Verwendung solcher Strukturen ist nicht zwingend nötig, sie erhöht jedoch oft die Übersichtlichkeit und vereinfacht die Bearbeitung der Programmdaten.
Die einzelnen Elemente (Komponenten) können beispielsweise aus ganzen Zahlen und reellen Zahlen und auch aus Feldern für Strings bestehen :

```
struct
      {   int   nummer;
          double  preis;
          char name[81];
       } artikel ;                          // artikel ist  Name für Strukturvariable
```

Man zählt den Datentyp **struct** zu den benutzerdefinierten Datentypen. Auf das Kennwort struct folgt zwischen geschweiften Klammern eine Beschreibung der Komponenten, aus denen die Struktur zusammengesetzt ist. Variablen innerhalb einer Strukturdeklaration können nicht einzeln initialisiert werden.
Die Syntax der struct Datentypdeklarationen und Variablendeklarationen ist ganz analog zu enum Datentypen und Variablen !
Komponenten gleichen Datentyps lassen sich wie bei einfachen Variablendefinitionen in einer Liste zusammenfassen :

```
struct   { int  reell, imag  ;
         }     kozahl;
```

Wie einfache Variablen erhalten die Komponenten einen frei wählbaren Namen, welcher nur mit der Strukturvariablen verwendet werden kann. Strukturvariablen werden über ihre Komponenten angesprochen :

```
kozahl.reell  =  15;
kozahl.imag= 3;
```

```
strcpy( artikel.name, "Schraube");
cout << kozahl.reell    << "+ "   << kozahl.imag   <<  "i" ;
```

Dabei verbindet der Auswahl - Operator "." den Namen der Strukturvariable mit dem Komponentennamen.
Die Bezeichner der Elemente dürfen auch in anderen Strukturen oder Variablen verwendet werden, da sie immer nur an einen ganz bestimmten Strukturtyp gebunden sind.
In einer Strukturdefinition können auch mehrere Variablen für den gleichen Strukturtyp definiert werden:

```
struct    {
              KOMPONENTENLISTE ;
          }  var1, var2, var3;
```

Eine Struktur kann als Element auch Felder haben. Umgekehrt kann man auch Felder definieren, deren Elemente aus Strukturen bestehen :

```
struct
   {    int   nummer;
        double  preis;
        char name[81];
    }  artikel[200] ;
```

Hier ist artikel[200] ein Feld aus 200 Strukturen, wobei jedes Feldelement (Struktur) unter anderem auch ein Feld (name[81]) enthält. Darüber mehr im nächsten Kapitel !

Allgemein unterscheidet man zwischen **Deklaration** der Struktur (das ist die Komponentenbeschreibung) und der **Definition**, welche Variablen mit den deklarierten Eigenschaften festlegt. Die Deklaration beschreibt den Musterbauplan der Struktur.

Für eine Struktur kann man auch einen Datentypnamen 'als Etikett' festlegen :

```
struct  complex   {                  //   Datentypname complex ist in C ein Etikett zu struct
                      int reell, imag;
                  } ;                //  Ohne Variablennamen ist hier das Semikolon nötig
```

Hier ist (in C) der Datentyp durch struct festgelegt, der Datentypname complex dient hier nur zur Vervollständigung (engl. tag) der Datentypbezeichnung. Eine Variablendefinition sieht nun (in C !) folgendermaßen aus :

```
struct complex  kzahl1,  kzahl2;     //    in C
```

In C++ gilt bei obiger Deklaration der Name complex als Datentyp. Daher definieren wir in C++ Strukturvariablen so :

```
complex  kzahl1 ,  kzahl2;          //   in C++
```

Vergleichsoperationen zwischen Strukturen und Wertzuweisungen von Strukturen, die zwar gleich aufgebaut sind , aber nicht zusammen definiert wurden, sind nicht zulässig. Im obigen Beispiel wurden kzahl1 und kzahl2 zusammen vereinbart, daher ist eine Wertzuweisung möglich :

```
kzahl2  =  kzahl1 ;
```

Die Ein- und Ausgabe von Strukturvariablen erfolgt normalerweise nur über die Komponenten. Bei der Ausgabe mit printf ist es allerdings möglich, alle Komponenten (außer Strings) auch durch Angabe der Strukturvariablen auszugeben.

<u>Ein Beispiel :</u>

```
struct   {
              int  nummer;
              double  wert;
           }  x;
cin     >> x.nummer  <<  x.wert ;    //  nur Komponenten in cin
cout   << x.nummer  << x.wert;     //   nur Komponenten in cout
printf("\n%i  %lg",  x);                // auch  Strukturvariable in printf
```

Auch **geschachtelte Strukturen** sind in C möglich ! Bei der Deklaration einer Struktur kann eine weitere Struktur innerhalb eingebettet werden. Dabei kann es sich um eine an anderer Stelle deklarierte Struktur handeln, oder man verwendet an der entsprechenden Stelle nochmals das Schlüsselwort struct und deklariert eine Struktur innerhalb einer anderen.

Ein Beispiel :

```
#include   <iostream.h>
#include   <string.h>
struct  datum
       {
           int  jahr;
           char  monat[10];
            int  tag;
        } ;

struct wetter
        {
            float temperatur;
            struct datum  zeitpunkt;
         };

struct wetter heute[365];

main()
{
     heute[0].temperatur     =  12.6;
     heute[0].zeitpunkt.jahr = 1998;
     heute[0].zeitpunkt.monat = "januar";
     heute[0].zeotpunkt.tag     = 1;
          ....   ...
}
```

Im Zusammenhang mit Strukturen ist der **sizeof-Operator** von besonderem Interesse. Dieser Operator ist ein unärer Operator und wirkt auf eine Variable, das Schlüsselwort für einen Datentyp, einen Ausdruck, ein Feld oder auch eine Struktur. Der sizeof-Operator liefert als Ergebnis die Länge des Operanden in Bytes.

```
                              double     dvariable ;
                               int intvar[10], anzahl;
                               float farray[] = {2.85, 27.123, 8.025,  10 };
                               struct  k
                                     {
```

```
            char  name[40];
             int  zahl1;
             double  d2;
          } konto;
int laenge ;

laenge =  sizeof(dvariable);        // in laenge steht nun  8
laenge =  sizeof(double);           // in laenge steht nun wiederum 8

laenge = sizeof (intvar);           // in laenge steht nun 80  ( = 40 * 2 )
laenge = sizeof (intvar[5] );// in laenge steht nun 2 ( für int Datentyp)

anzahl  = sizeof( farray)  / sizeof(float); // Anzahl der Feldelemente

laenge = sizeof( konto) ;           //  Strukturlänge
laenge = sizeof( k );               // gleiche Strukturlänge !
```

Mögliche Operanden in sizeof-Operator sind : Datentyp, Variable, Konstante, Ausdruck

8.2) Vektoren von Strukturen

Die Elemente einer Struktur können auch Felder sein. Ebenso können die Komponenten eines Feldes Strukturen sein.

Ein Beispiel :

```
struct   messwert
       {
           float  x,  y;
           float  temperatur;
        }  messfeld[50];
             .....
  messfeld[26].x  =  12.8;
  messfeld[26].y  =   58.34;
  messfeld[26].temperatur =  45.1;
```

Dieses Beispiel deklariert zunächst eine Struktur mit den Elementen x, y und temperatur; Mit der Zusatzangabe der Variablendefinition für messfeld[50] wird schließlich ein Feld aus 50 solchen Strukturen erzeugt. Mit dem deklarierten Datentyp struct messwert lassen sich nachfolgend auch weitere Variablen definieren :

```
struct  messwert  m1,  m2[8],  m3 ;
```

Mit solchen Deklarationen und Definitionen erhält man eine gute Übersicht bezw. Ordnung über alle verwendeten Variablen und Felder. Dies wird bei dem folgenden Beispiel besonders deutlich :

```
struct  adresse
       {
          char  name[30];
          char  ort[40];
          char strasse[70];
        }  vertreter [100] ;
```

Wenn wir eine bestimmte Stelle im Namensfeld vom 30. Vertreter referieren wollen, dann sieht das folgendermaßen aus :

```
vertreter[30].name[18]  =  'r' ;
```

Wenn Sie eine Struktur deklarieren, hat sie normalerweise keine definierten Inhalte. Lediglich bei static-Variablen und bei globalen Variablen ist die Struktur schon mit Nullen vorbelegt.

Die Initialisierung von Strukturvariablen hat viel Ähnlichkeit mit dem Vorgehen bei Feldern. Nach dem Namen folgt ein Zuweisungszeichen, gefolgt von geschweiften Klammern. Innerhalb dieser Klammern kommt die durch Kommas getrennte Werteliste :

```
struct  fehler
       {
           int nummer;
           char text[30];
        }  einzelfehler = {  46, "Falsche Eingabe "  } ;

struct fehler fliste[3] =      {
                                  { 0, "Alles  OK"   },
                                  {1, "Allgemeiner Fehler" },
                                  {2, "Unbekannter Fehler"}
                                } ;    // Die inneren geschweiften Klammern sind
                                       //  nicht notwendig !
```

8.3) Unions

Der **Datentyp union** ist eine Zuammenfassung von Varianten (Komponenten) unterschiedlicher Datentypen, die im Gegensatz zu einer Struktur alle den gleichen Speicherplatz belegen (teilen). Es gelten die gleichen Regeln wie für die Vereinbarung von Strukturen, jedoch tritt an die Stelle des Kennworts struct das Kennwort **union.**
Das folgende Beispiel zeigt drei Strukturelemente mit unterschiedlichen Datentypen : alle drei liegen beginnend auf den gleichen Speicherstellen.

```
union   {
            float   f ;                  //  4 Bytes für float
            double d ;                   //  8 Bytes für double
            char ganz[10];               //   10 Bytes  für Zeichenfolge
         }  zahl;
```

Die Länge einer union entspricht der Länge des Datentyps mit der größten Länge. Die Variablen einer union lassen sich genauso ansprechen wie die von Strukturen.

```
zahl.f  = 12.34 ;
```

Ein Wert aus jedem der obigen drei Typen kann der union - Variablen zahl zugewiesen und dann in Ausdrücken verwendet werden, solange die Benutzung konsistent ist : der Datentyp , welcher entnommen wird, muß der Typ sein, der als letzter zuvor gespeichert wurde.
Unions können innerhalb von Strukturen und Feldern auftreten und umgekehrt.

```
struct   {      int  utype ;      // Inhalt von utype gibt aktuellen Datentyp in union an.
                union   {
                            int ival;
                            float  fval;
```

```
                    char  sval[20] ;
                }
        } un ;
```

Eine union kann nur mit einem Wert initialisiert werden, der zum Typ der ersten Alternative paßt ; die oben beschriebene union kann also nur mit einem int- Wert initialisiert werden.
Eine Anwendung von unions sind Symboltabellen bei Textverarbeitung : Die programmtechnische Verwaltung solcher Tabellen ist einfacher, wenn ein Wert (in einer Spalte) unabhängig von seinem Datentyp jeweils gleich viel Speicher benötigt. Die union ist eine Variable, welche legitimerweise einen beliebigen von mehreren Typen aufnehmen kann.

8.4) Bitfelder

Ein Sonderfall der Struktur ist das Bitfeld : dies ist eine Struktur, deren einzelne Variablen vom Typ int oder unsigned int sind. Die Länge der Variablen wird in Bits angegegben :

```
struct   frage
    {
        unsigned  int   druckein : 1;
        unsigned int  ega  :  1;
        int  retcode : 2;
        int zustand : 8;
        unsigned int  halbbyte : 4;
     } flags;
```

Ein Bitfeld hat genau die Länge eines Wortes (16 Bit bei 80x86) . Die 16 Bits können hintereinander in einzelne Felder aufgeteilt werden, die Summe darf 16 nicht überschreiten.
Im vorangehenden Beispiel kann ega den wert 0 oder 1 annehmen, die Variable retcode die Werte -2, -1, 0, 1 .
Man darf innerhalb eines Bitfeldes auch Lücken lassen :

```
struct   {
        int feld1 : 4;
        int : 4;          //  ungenutzt, da ohne Name
        unsigned  imt feld 2 :  6;
         int  feld 3 : 2;
    }  bfeld;
```

Eine Anwedung von Bitfeldern ist beispielsweise in Symboltabellen, wo in einem Wort (2 Bytes) bitweise Information etwa für symbolische Namen festgehalten werden.
Der Zugriff auf einzelne Bitfelder ist wie bei Namen in gewöhnlichen Strukturen.
Denken Sie daran, daß ein Programm mit Bitfeldern unter Umständen nicht mehr portabel ist !

8.5) typedef struct

Mit Hilfe von **typedef** können in C neue Typnamen vereinbart werden. Beispielsweise wird in

```
typedef  float  Gleikomma ;
```

der Name Gleikomma synonym zu float. Die Typbezeichnung Gleitkomma kann nun bei nachfolgenden Deklarationen verwendet werden :

```
Gleikomma  radius, durchmesser;     //  radius und durchmesser sind nun vom Typ float.
```

Im großen und ganzen hat typedef den gleichen Effekt wie #define, jedoch wird die Ersetzung in typedef vom Compiler vorgenommen, während #define vom Preprozessor bearbeitet wird. Für die mittels typedef definierten Variablen gelten die normalen Regeln bezüglich Gültigkeitsbereich. Allgemeine Syntaxfehler kann der Compiler bei typedef eher finden als bei #define !!
Zu beachten noch, dass die entsprechende Formulierung in #define wie folgt lautet :

```
#define GLEITKOMMA   float    // ohne Semikolon
```

In typedef ist Gleitkomma der zweite Name, in #define ist GLEITKOMMA der erste Name.
Ein Beispiel :

```
typedef  int BOOL;

#define  FALSCH  0
#define  WAHR  1
main()
{
      BOOL     a;
      a  =  WAHR;
      cout  <<   a;
}
```

Mit typedef kann man in C einer Struktur einen neuen Datentypnamen verpassen :

```
typedef  struct  adresse      // Die  Typ-Etikettierung adresse kann auch weggelassen
                              //    werden.
                  {
                     char  name[30];
                     char  ort[25];
                   }  Anschrift ;
```

Hier ist sowohl adresse als auch Anschrift eine Typenbezeichnung für die deklarierte Struktur :

```
struct  adresse   Einwohner;       //  in C
Anschrift  vertreter , kunde ;     //  in C  !
```

Nun sind vertreter und kunde zwei Strukturvariablen .
Eine typedef - Vereinbarung konstruiert niemals einen neuen Datentyp, sonder es wird lediglich ein zusätzlicher Name für einen existenten Typ eingeführt. Im obigen Strukturbeispiel wird die Struktur mit typedef implizit deklariert.

Auch für Aufzählungen lassen sich mit typedef neue Datentypen deklarieren :

```
typedef  enum  { FALSCH,  WAHR  }  bool;          // Typdeklaration für bool;
bool  x,  y;                                      //   Logische Variablen  x,  y  .
```

Die Verwendung von typedef für Felder sieht so aus :

```
typedef  char feld[30];
feld    ausgabe  ;                 //  ausgabe ist nun ein Feld der Länge  30
```

```
     /*********************************************************/
     /*                                                       */
     /*   Einführung in ANSI C :  Beispiel 1        Nov 2002  */
     /*                                             M. Baum   */
     /*********************************************************/

       // Beispiel 1 :Zwei ganze Zahlen summieren.

#include <iostream.h>                /* System I/O-Funktionen */

main()                               // Hauptfunktion

{
    int x,y,z;                       // Variablenvereinbarungen
    int bruch,                       // Ganzzahliger Quotient
        rest ;
// Rest bei ganzzahliger Division

    cout  << "\nBitte, zwei ganze Zahlen eingeben :"  ;
    cin >>  x     >>  y;

    z = x + y;              // Summe bilden
    cout << x <<"+" << y <<" ergibt " << z ;//Zahlen und Summe ausgeben

    bruch  =  y  /  x ;     // Ganzzahliges Ergebnis bei Division
    cout <<'\n' <<  y  <<'/'  << x << " ergibt " << bruch ;

    rest   =  y  %  x ;     // Rest bei ganzzahliger Division: 'modulo'
    cout <<  '\n' << y <<'%'  << x <<" ergibt "  << rest ;

    return(0);  //  Rücksprung nach System
}
```

```
/*        Programmablaufplan :
 *       =====================
 *                |
 *      ___________V___________
 *     | Ganze Zahlen x,y      |
 *     |    einlesen           |
 *     |_______________________|
 *                |
 *                |
 *      ___________V___________
 *     | Summe x+y berechnen   |
 *     | und ausgeben          |
 *     |_______________________|
 *                |
 *                |
 *      ___________V___________
 *     | Quotient x/y berechnen |
 *     | und ausgeben          |
 *     |_______________________|
 *                |
 *                |
 *     ____________V___________
 *     | x % y berechnen und   |
 *     | Quotientenrest ausgeben|
 *     |_______________________|
 */
```

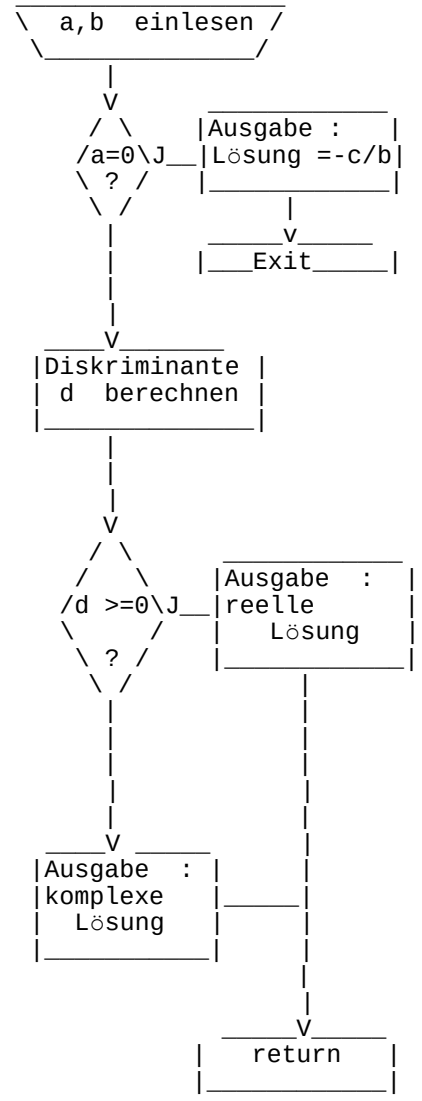
Struktogramm :
==========

Ganze Zahlen x, y
einlesen
Summe x+y berechnen und ausgeben
Quotient x/y berechnen und ausgeben
x % y berechnen und Quotientenrest ausgeben

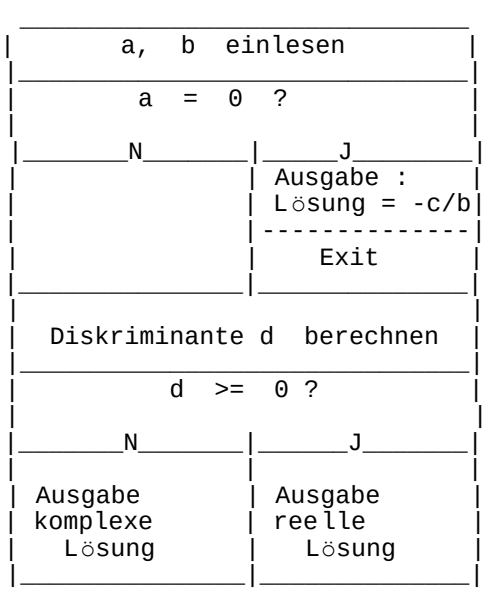
```
     /*********************************************************/

     /*                                                       */
     /*   Einführung in ANSI C :  Beispiel 2        Nov 2002  */
     /*                           if - Anweisung    M. Baum   */
     /*********************************************************/

     // Beispiel 2 :  Quadratische Gleichung lösen.

#include <iostream.h>             /* System I/O-Funktionen */
#include <math.h>
#include <process.h>

#define  C  25.0                  // Konstante der quadratischen Gleichung
main()                            // Hauptfunktion

{
    float  a , b ,            // Konstanten der quadr. Gleichung
           d ;                // Diskriminante der quadr. Gleichung

    cout <<"\nBitte, reelle Werte "
         <<" für die Konstanten a und b eingeben :" ;

    cin  >> a   >> b ;

    if(a == 0 )
      {  cout << "\nLösung = "  <<  -C / b ;
           exit(1);
       }

    d  =  b * b  -  4 * a * C  ; // Berechnung der Diskriminante

    if( d >= 0 )                 // Nur reelle Lösungen
        { cout <<"\nLösung 1 = "  << (-b + sqrt(d) ) / (2 * a ) ;
          cout <<"\nLösung 2 = "  << (-b - sqrt(d) ) / (2 * a ) ;
        }
    else                      // Komplexe Lösungen
        { cout <<"\nLösung 1= " <<-b/(2*a) <<'+' << sqrt(-d)/(2*a) <<" i";
        cout <<"\nLösung 2= " <<-b/(2*a) <<'-' << sqrt(-d)/(2*a) <<" i";
        }

    return 0;
}
```

/*

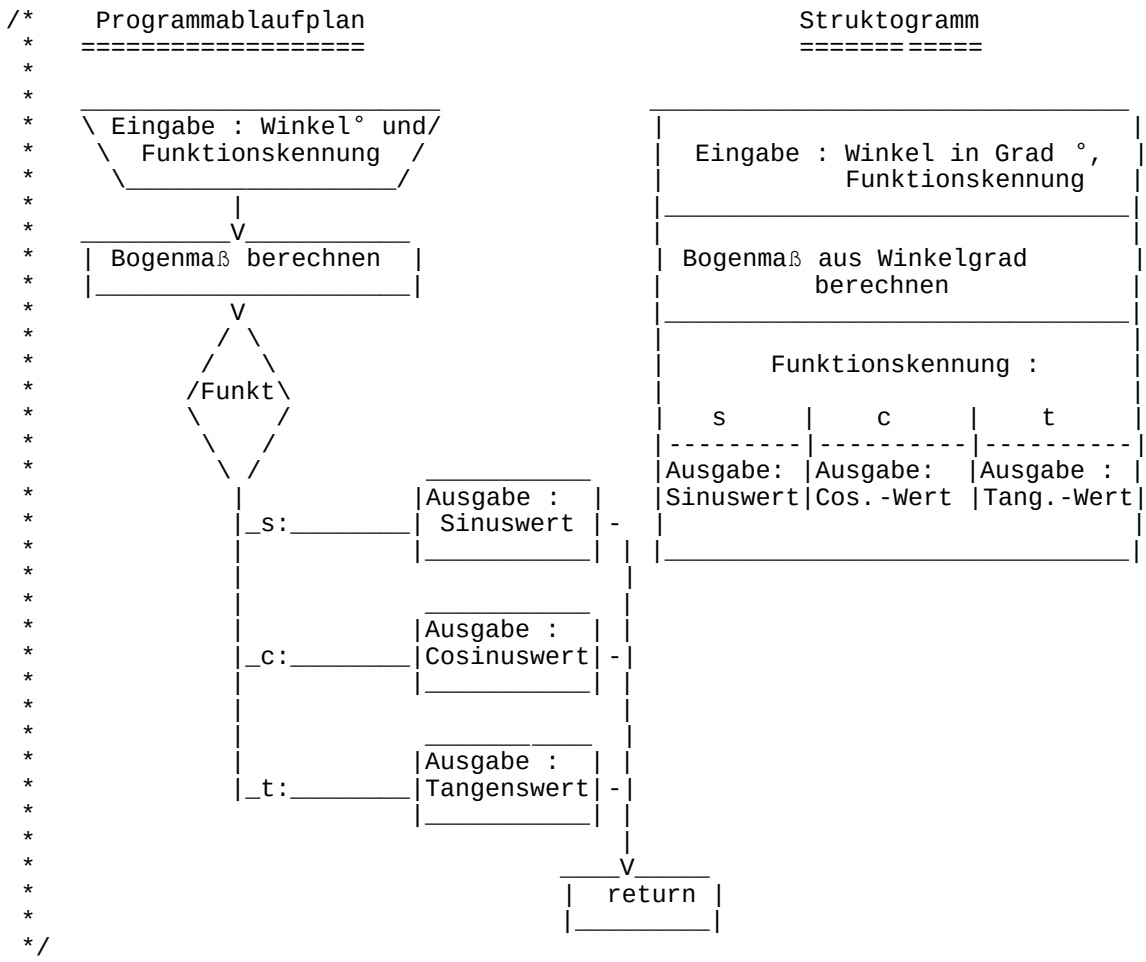
Programmablaufplan



Struktogramm



```
     /*********************************************************/
     /*                                                       */
     /*   Einführung in ANSI C :  Beispiel 3        Nov 2002  */
     /*                          switch-Anweisung    M. Baum   */
     /*********************************************************/

#include <iostream.h>                /* System I/O-Funktionen */
#include <math.h>

main()                               // Hauptfunktion
{
   char  funktion;              // Aufzurufende Winkelfunktion

   double winkel,               // Winkelmaß in Grad
          bogen ;               // Bogenmaß

   cout << "\nEingabe von Winkel in Grad  -> ";      cin >> winkel;

   cout << "s = Sinus, c= Cosinus, t = Tangens  ->"; cin >> funktion;

   bogen = winkel * M_PI / 180 ;    // Umrechnen in Bogenmaß

   switch(funktion)
       {
        case 's' :  cout << "\nSin " << winkel <<" ° = "  << sin(bogen);
                    break;

        case 'c' :
        case 'C' :  cout << "\nCos " << winkel <<" ° = " << cos(bogen);
                    break;

        case 't' :  cout << "\nTan " << winkel <<" ° = " << tan(bogen);
                    break;

        default  :  cout << "\n Fehler : kein Kennbuchstabe s  c  t ";

       }

            return(0);                       // Rücksprung nach System
}
```


Programmablaufplan
Struktogramm
Eingabe : Winkel° und/
Funktionskennung
Bogenmaß berechnen
Funkt
Ausgabe : Sinuswert
s:
Ausgabe : Cosinuswert
c:
Ausgabe : Tangenswert
t:
return
Eingabe : Winkel in Grad °, Funktionskennung
Bogenmaß aus Winkelgrad berechnen
Funktionskennung :
s
c
t
Ausgabe: Sinuswert
Ausgabe: Cos.-Wert
Ausgabe : Tang.-Wert

```
     /*********************************************************/
     /*                                                       */
     /*   Einführung in ANSI C :  Beispiel 4        Nov 2002  */
     /*                           while,  for       M. Baum   */
     /*********************************************************/

#include <iostream.h>                 /* System I/O-Funktionen */
#include <iomanip.h>           // Für setw()
#include <stdio.h>             // Für printf
      /*  Umwandlung von Fahrenheit in Celsius
              für Fahr = 0, 20, ... , 160.             */

main()                         // Hauptfunktion
{
   float  fahr,  celsius ;     // Wärmemaße

   int    unter = 0,           // Untere Grenze
        ober  = 160,         // Obere Grenze
        schritt = 20 ;       // Schrittbreite in Schleife

   fahr = unter ;

   cout.precision(2);
   cout << endl << setw(10) <<"Fahrenheit" << setw(10) <<"Celsius"
        << endl ;

   while ( fahr  <= ober )
      { celsius = ( 5.0 / 9.0) * (fahr - 32) ;
        cout << endl << setw(10)  << fahr  << setw(10)  << celsius ;
        fahr = fahr + schritt;
      }

   printf("\n\n");

   for ( fahr = unter; fahr <= ober; fahr = fahr + schritt )
      { celsius = ( 5.0 / 9.0) * ( fahr - 32);

        printf ("\n %10.2f %10.2f", fahr, celsius );

      }

   return(0);                         // Rücksprung nach System
}
```

```
 *      Programmablaufplan                    Struktogramm
 *       ==================                    =============
 *
 *
 *          ________________                 ____________________________
 *         | unter  =   0  |                |  unter = 0,  ober =  160, |
 *         | ober   = 160  |                |  schritt = 20,fahr = unter|
 *         | schritt=  20  |                |___________________________|
 *         | fahr  = unter |                |                           |
 *         |_______________|                |   Headerzeile ausgeben    |
 *                 |                        |___________________________|
 *                 |                        | Solange  fahr  <= ober    |
 *         ________|_______                 |    ______________________ |
 *         |Headerzeile   |                 |    | Berechne celsius    |
 *         |   ausgeben   |                 |    |    aus  fahr        |
 *         |______________|                 |    |_____________________|
 *                 |                        |    | Ausgabe : fahr,     |
 *     --->-------->|                        |    |           celsius   |
 *     |            V                        |    |_____________________|
 *     |           /\                        |    | fahr = fahr + schritt|
 *     |          /  \     ________          |____|_____________________|
 *     |         /fahr\ N  |        |        |für alle fahr, von unter   |
 *     |        /  <=  \___| return |        | bis ober, in schritt-Weite|
 *     |        \ ober /   |________|        |    ______________________ |
 *     |         \    /                      |    | Berechne celsius    |
 *     |          \  /                       |    |     aus   fahr      |
 *     |           \/                        |    |_____________________|
 *     |          Ja |                       |    | Ausgabe : fahr,     |
 *     |      ________V_________             |    |           celsius   |
 *     |     |Berechne celsius |             |____|_____________________|
 *     |     |   aus  fahr     |
 *     |     |_________________|
 *     |              |
 *     |      ________V_________
 *     |     |Ausgabe : fahr,  |
 *     |     |          celsius |
 *     |     |_________________|
 *     |              |
 *     |      ________V_________
 *     |     |                 |
 *     -<<<-|fahr=fahr+schritt|
 *          |_________________|
 *
 *
 *     Hinweis : Im Programmablaufplan
 *               ist die for-Anweisung
 *                nicht extra gezeichnet,
 *                das Bild wäre gleich wie
 *                für die while - Schleife.
 */
```

```
     ******************************************************/
     /*                                                    */
     /*       Einführung in ANSI C :  Beispiel 5  Nov 2002  */
     /*                              do - while      M. Baum   */
     /******************************************************/
       /*  Beispiel für Reihenentwicklung :
        *          1
        *  s  =  ----- = 1 - x + x² - x³ + ...   für |x| < 1
        *        1 + x
        */
#include <stdio.h>                /* System I/O-Funktionen */
#include <math.h>
main()
{
    const double epsilon = 1E-8;    // Genauigkeitsgrenze
    double  term  = 1,          // Reihen - Glied
            summe = 1,          // Zwischensumme
            x          ;        // Variable
    printf(" Bitte Wert für x eingeben, mit |x| < 1 :");
    scanf(" %lf",&x); printf("x =  %10.3lf ", x);
    do
      {
         term  = -term * x;
         summe = summe + term;
      }
    while (epsilon < fabs(term) ) ;
    printf("\nNäherung für x = %6.3lf ist %10.3lf", x, summe);
    return(0);    // Rücksprung nach System
}
```

```
/*      Programmablaufplan
 *      ==================
 *
 *       _________________
 *       | epsilon= 1E-8 |
 *       | term   = 1    |
 *       | summe  = 1    |
 *       |_______________|
 *               |
 *        ________V________
 *        \   Wert für x  /
 *         \   einlesen  /
 *          \___________/
 *               |
 *    --->>-----<-|
 *    |     ______V_________
 *    |     | term = -term * x|
 *    |     | summe=summe+term|
 *    |     |_________________|
 *    |           |
 *    |           V
 *    |          /\
 *    |         /  \          _________
 *    |        epsilon    | Ausgabe |
 *    |__<__<_ /    <  \_N_| summe   |
 *          Ja  fabs(term) |_________|
 *               \    /         |
 *                \  /          V
 *                 \/         return
 */
```

Struktogramm

```
_____________________________
| epsilon = 1E-8,  term = 1 |
| summe = 1                 |
|___________________________|
| Wert für int x einlesen   |
|___________________________|
|  |                        |
|  |term  =   -term * x     |
|  |summe =  summe + term   |
|  |                        |
|  |________________________|
|solange                    |
|   epsilon < fabs(term)    |
|___________________________|
|                           |
|       Ausgabe  summe      |
|___________________________|
```

```
/*********************************************************/
/*                                                       */
/*   Einführung in ANSI C :  Beispiel 6        Nov 2002  */
/*                           Ausdrücke         M. Baum   */
/*********************************************************/


       //  Allgemeine Ausdrücke und Wertzuweisungen

#include <iostream.h>                 /* System I/O-Funktionen */
#include <stdio.h>
main()
  {
  char c;     // einzulesendes Zeichen
  long nc = 0 ;    // Anzahl der eingelesenen Zeichen

 // Kopierprogramm,   Version 1
  cout << "Bitte, Zeichenfolge_1 eingeben, Abschluß mit  ^Z\n ";

  c = cin.get();         // Lies  ein Zeichen von Konsole

  while(c != EOF)
     { cout << c;        // Gib das Zeichen aus
       c = cin.get();
     }



 // Kopierprogramm,     Version 2

   cin.clear();  rewind(stdin);
   cout <<  "\nBitte, Zeichenfolge_2 eingeben,  Abschluß mit ^Z\n";

   while ( (c = cin.get())  != EOF)
        cout << c;




 // Eingabezeichen zählen und ausgeben

    cin.clear(); rewind(stdin);
    cout << "\nBitte, Zeichenfolge_3 eingeben,  Abschluß mit  ^Z\n";

    while (cin.get()  !=  EOF )
      ++nc;

    cout  << endl  <<"Anzahl der Zeichen = " << nc ;


  return(0);                        // Rücksprung nach System

  }
```

```
     /*******************************************************/
     /*                                                     */
     /*   Einführung in ANSI C :  Beispiel 7        Nov 2002  */
     /*                           cin,  cout        M. Baum   */
     /*******************************************************/

            /*   Leseschleife  für Wurzelberechnung     */


#include <iostream.h>       //   System I/O-Funktionen
#include <math.h>           //   für sqrt
main()
{
    double  radikand ;

    do
      { cout << "\nWert zur Wurzelberechnung eingeben -> ";
        cin  >> radikand;


        cout.precision(2);   // Ausgaben mit 2 Nachpunktstellen
        cout <<" \nQuadratwurzel aus "   << radikand   <<" = "
             << sqrt(radikand );

        cout << "\nNochmal? n = nein -> ";

        cin.ignore(80, 10 );              // Eingabepuffer leeren !!

       }

    while (cin.get()  !=  'n');             // Antwort lesen und auswerten

    return 0;
}

   /* Im Gegensatz zu cin >> und  scanf liefern  cin.get(),  getchar(),
      getch()  und getche() das nächste eingelesene Zeichen           */
```

```
     /*********************************************************/
     /*                                                       */
     /*   Einführung in ANSI C :  Beispiel 8        Nov 2002  */
     /*                           printf, scanf     M. Baum   */
     /*********************************************************/


#include <stdio.h>               /* System I/O-Funktionen */

main()
{
    double gehalt;

    printf("\n\t\tSteuerberechnung ") ;
    printf("\nBitte Gehalt eingeben ");
    scanf("%lf", &gehalt);

      while (gehalt > 0 )
        {
        if(gehalt < 500  )
             {
             printf("\n Keine Steuern für Gehalt  < 500 ") ;
           }
        if(gehalt >= 500   && gehalt < 3000 )
           {
             printf("\n%.2lf DM Steuern für Gehalt = %lf",gehalt/100*18.5,
                                              gehalt );
           }
        if(gehalt >= 3000 )
           {
             printf("\n%.2lf DM Steuern für Gehalt = %lf",gehalt/100*40,
                                                    gehalt);
           }

        printf("\nBitte Gehalt eingeben  ");
        scanf("%lf", &gehalt);

         }
    return  0;
}
```

```
     /*********************************************************/
     /*                                                       */
     /*   Einführung in ANSI C :  Beispiel 9        Nov 2002  */
     /*                           Felder            M. Baum   */
     /*********************************************************/


            //  Mittelwert und Abweichung berechnen

#include <iostream.h>                /* System I/O-Funktionen */
#include <iomanip.h>       // für setw
#define N   5              // Symbolkonstante für Anzahl der Elemente

main()
{
    int i;          // Laufvariable
    double x[N],    // Feld aus N Elementen,  Index 0 bis N-1
           Sum = 0, // Summe der Elemente
         Xm;      // Mittelwert

        // Feld aus N Elementen einlesen

    cout <<"\nFeld aus "  << N  <<" Werten eingeben\n";
    for (i=0; i < N; i++)  // Vorsicht, keine Eingabekontrolle !
       {
          cout  <<"x[" << i <<"]  -> ";
          cin   >> x[i]; cin.ignore(80,10);
       }

      // N Elemente des Feldes summieren :

    for (i=0; i < N; i++)  Sum  = Sum + x[i];

      // Summe und Mittelwert ausgeben :

    Xm = Sum / N;
    cout <<"Summe = " << Sum   <<"Mittewert = " << Xm  << endl;
    cout << "\n       Wert     Abweichung";

      // Wert und Abweichung vom Mittelwert ausgeben :

    for ( i = 0;  i < N;  i++)
        cout  <<"\n"  <<setw(10)   << x[i]  <<setw(10)  << x[i] - Xm;

    cout << " \n\nWeiter -> "; cin.get();

    return 0;     // Rücksprung nach System

}
```

```
     /*********************************************************/
     /*                                                       */
     /*   Einführung in ANSI C :  Beispiel 10       Nov 2002  */
     /*                          Felder,printf,scanf  M. Baum  */
     /*********************************************************/


   // Histogramm über Häufigkeit von verschiedenen Eingabezeichen

#include <stdio.h>         //   System I/O-Funktionen
#include <conio.h>         //   Für getch()
#include <ctype.h>         //   Für isprint(i)

#define MAXHIST  15    // maximale Länge des Histogramms
#define MAXCHAR  128   // maximale Zahl verschiedener Zeichen
main()
{
   int c,  i,          // Eingabezeichen c, Zählvariable i
       len,            // Länge eines Balkens
       maxvalue = 0,   // Maximale Wert für cc[]

       cc[MAXCHAR] ;   // Zählerfeld für die Zeichen

   for (i=0; i < MAXCHAR; ++i)  cc[i] = 0 ;    // Zählerfeld initialisieren

   while ( (c = getchar() ) != EOF )           // Zählerfeld auffüllen
      if (c < MAXCHAR)
           ++cc[c] ;

   for (i=1; i < MAXCHAR;  ++i) //maxvalue auf den größten cc[i]-Wert setzen
       if ( cc[i] > maxvalue)
           maxvalue = cc[i];

   for (i=1; i < MAXCHAR; ++i)
      {
      if ( isprint(i) )       // sichtbares Zeichen,  auch Leerzeichen
        printf("%5d - %c - %5d : ", i, i,  cc[i] );
      else
        printf("%5d -    -%5d : ", i,  cc[i] );

      len = cc[i] * MAXHIST / maxvalue;
      while ( len > 0 )
          {
            putchar('*');
            --len;
          }
      putchar('\n');
        if ( i % 20  == 0 ) getch();
      }

   return 0;     // Rücksprung nach System
}
```

```
/*
 *                                  Struktogramm
 *                                  ============
 *  ____________________________________
 * |       maxvalue  = 0                |
 * |------------------------------------|
 * |Für alle i von 0 bis MAXCHAR-1      |
 * |    ____________________________    |
 * |    |                           |   |
 * |    |    cc[i]   = 0            |   |
 * |------------------------------------|
 * |    c  =   getchar()                |
 * | Solange c ungleich EOF             |
 * |    ____________________________    |
 * |    |       c < MAXCHAR  ?      |   |
 * |    |           |               |   |
 * |    |___JA______|_____NEIN______|   |
 * |    |cc[c] =    |               |   |
 * |    |  cc[c]+ 1 |               |   |
 * |    |___________|_______________|   |
 * |    |  c  =   getchar()         |   |
 * |___|____________________________|   |
 * |Für all i von 1 bis MAXCHAR-1       |
 * |    ____________________________    |
 * |    |    cc[i]  >  maxvalue ?   |   |
 * |    |           |               |   |
 * |    |___JA______|_____NEIN______|   |
 * |    |maxvalue =  |               |   |
 * |    |    cc[i]   |               |   |
 * |___|____________|_______________|   |
 * |Für alle i von 1 bis MAXCHAR-1      |
 * |    ____________________________    |
 * |    | Ist i ein sichtbares Zeichen? |
 * |    |           |                |   |
 * |    |___JA______|____NEIN________|   |
 * |    | Ausgabe mit| Ausgabe ohne  |   |
 * |    |  Zeichen   |   Zeichen     |   |
 * |    |____________|_______________|   |
 * |    |len = cc[i] * MAXHIST /maxvalue|
 * |    |____________________________    |
 * |    | Solange  len  > 0          |   |
 * |    |    ________________________    |
 * |    |    | Ausgabe '*' am Bildschirm|
 * |    |    |________________________|   |
 * |    |    |    len  = len  - 1     |   |
 * |    |____|________________________|   |
 * |    | Neue  Zeile  beginnen      |   |
 * |___|____________________________|   |
 *
 */
```

```
     /*********************************************************/
     /*                                                       */
     /*   Einführung in ANSI C :  Beispiel 11      Nov 2002   */
     /*                          Zweidim. Felder    M. Baum   */
     /*********************************************************/

            //  Konstantenlisten und Anordnung im Speicher

#include <iostream.h>                 /* System I/O-Funktionen */
#include <iomanip.h>       // für setw

#define NZ  3              // Anzahl der Zeilen
#define NS  4              // Anzahl der Spalten

main()
{
    int i, j, k;          // Laufvariablen
    double a[NZ][NS]  = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
    const double b[NZ][NS] = { {0,1,2,3} , {4,5,6,7} , {8,9,10,11 } };
    cout <<"\nMatrix a zeilenweise ausgeben  C++ Index\n    ";

    for (k=1; k <= NS; k++)  cout << setw(7)  << k   <<" .Spalte";

    for (i=0; i < NZ; i++)
       {
         cout <<"\n" <<i  <<".Zeile:";
         for (j = 0; j < NS; j++)
           cout <<"  A[" <<i <<"] [" << j <<"] =" << setw(3)<< a[i][j];
       }

    cout <<"\n\nMatrix b zeilenweise ausgeben, mathem. Index\n    ";
    for ( k=0; k < NS; k++) cout << setw(7)  << k << ".Spalte ";

    for ( i=0; i < NZ; i++)
       {
         cout << "\n"  << i+1  <<".Zeile:";
         for (j = 0; j < NS; j++)
           cout <<"  B_" <<i+1 <<j+1 <<" =" <<setw(3) << b[i][j] <<"    ";
       }

    cout <<"\n\nWeiter ->";  cin.get();


    return 0;     // Rücksprung nach System

}
```

```
     /*********************************************************/
     /*                                                       */
     /*   Einführung in ANSI C :  Beispiel 12      Nov 2002   */
     /*                           gets,  puts       M. Baum   */
     /*********************************************************/

#include  <stdio.h>

main()
{
    char zkette[21] ;         // String für maximal 20 Zeichen
    int  iz  = 0;            // Schleifenvariable  iz


    printf("\n\nGib einen Text ein, max. 80 Zeichen:\n");

    gets (zkette);    // String einlesen
    puts (zkette);    // String auf Bildschirm ausgeben



      // Nachfolgend werden Zeichen einzeln eingelesen, mit Bereichsprüfung:


    puts("\n\nGib nochmals einen Text ein, ");


    puts("max. 20 Zeichen werden eingelesen:\n");

    while (iz < 20 )   // Schleife über max.  20 Zeichen
      {
        zkette[iz]  =  getchar();
        if ( zkette[iz] =='\x0A' )     // Wenn Enter  gedrückt,
             break;                    // dann Schleife verlassen
          ++iz;

        }

    zkette[iz]  =  '\x00' ;      // Stringende setzen

    puts(  zkette);              // String wieder ausgeben

    return 0;     // Rücksprung nach System

}
```

```
     /*********************************************************/
     /*                                                       */
     /*   Einführung in ANSI C :  Beispiel 13       Nov 2002  */
     /*                          Strukturen         M. Baum   */
     /*********************************************************/


#include <iostream.h>                /* System I/O-Funktionen */
#include <stdio.h>

main()
{
    float Kontogebuehr = 0;

    struct {          // Deklaration der Struktur Konto
             char Name[30];
           char Adresse[50];
           long int Kontonr;
           float Wert;        // Kontostand
             int Anzbew;        // Zahl der Kontobewegungen

         } Konto;      // Ende der Struktur mit Namen Konto

    for(;;)     // Kontodaten eingeben, Ende mit Return
       {
         cout <<"\nName (Programmende mit 'nur Return')";gets(Konto.Name);
       if (Konto.Name[0] == '\0' ) break;   // Ende der for-Schleife

         cout << "\nAdresse:";  gets( Konto.Adresse );

       cout << "\nKontonummer :"; cin >> Konto.Kontonr;
       cout << "\nKontostand :";  cin >> Konto.Wert;
       cout << "\nAnzahl der Bewegungen :"; cin >> Konto.Anzbew;

       Kontogebuehr = Konto.Anzbew * 0.85;  // Kontogebuehr
       Konto.Wert   = Konto.Wert - Kontogebuehr;

       cout << "\n\n\n ***** Kontodaten ausgeben   *** ";
       cout << "\n Kontonummer = " << Konto.Kontonr;

       cout.precision(2);
       cout << "  Kontostand = " << Konto.Wert;

       cout << "\n "  << Konto.Name << "   " << Konto.Adresse;
       cout << "\n Kontogebuehr = " << Kontogebuehr ;

        }  /*  Ende der  for-Schleife   */


    return 0;     // Rücksprung nach System

}
```

```
     /*******************************************************/
     /*                                                     */
     /*   Einführung in ANSI C :  Beispiel 14       Nov 2002  */
     /*                          Struktur           M. Baum   */
     /*******************************************************/

           //  Struktur für Bauteileverwaltung
#include <iostream.h>                /* System I/O-Funktionen */
#include <iomanip.h>
#include <string.h>    // für strlen() Funktion

#define N 10          //  Max. Anzahl der Bauteile

main()
{   struct {
             char Name[81];    //  String für Bauteilbezeichnung
             int  Menge;       //  Lagerbestand
           double Preis;     //  Stückpreis
           }  Lager[N];       //  Feld aus Strukturen

    int  i,j,          // Laufvariablen
         Anzahl;       // Anzahl der Bauelemente

    double Msteu,      // Mehrwertsteuer
           Summe = 0,  // Gesamtpreis aller Bauteile
         Wert;       // Gesamtpreis gleicher Bauteile

    cout << "\nAnzahl (maximal "  << N <<") --> "; cin >> Anzahl;
    for (i=0;i < Anzahl; i++)
        {
        cout << "Nr."  << i+1  <<": Bezeichnung ----> ";
        cin  >> Lager[i].Name; cin.ignore(80,10);
        cout <<"      Lagerbestand ---> "; cin >> Lager[i].Menge;
        cout <<"      Stückpreis[DM] -> "; cin >> Lager[i].Preis;
      }
    cout << "\n          Bauteil    Menge     Wert\n";
    for (i=0; i < Anzahl; i++)
        {
        Wert = Lager[i].Menge * Lager[i].Preis;
          Summe = Summe + Wert;
        for (j = 1; j < (14-strlen(Lager[i].Name) ); j++) cout.put(' ');
        cout << Lager[i].Name  <<setw(12)  << Lager[i].Menge
             << setw(10)  << Wert  << endl;
        }
    cout  <<"================================";
    cout  <<"\n               Gesamtwert  "  << setw(8)  << Summe;
    cout  <<"\n\n        Mehrwertsteuersatz in % ->",  cin >> Msteu;
    cout  <<"   Bewertung mit Mehrwertsteuer "  << setw(6)
          << Summe * ( 1 + 0.01 * Msteu )  << " DM";

    cout <<"\n\nWeiter -> "; cin.ignore(80,10); cin.get();


    return 0;     // Rücksprung nach System
}
```

BA Stuttgart , Elektrotechnik / Informatik WS 2002 / 2003 1. HJ

M. Baum

Übungen zu C / C++

Für jede bearbeitete Übung wird erwartet :

- *** Entwurf und Darstellung des Programms mit einem Programmablaufplan oder Struktogramm .**
- *** Konsistente optische Programm - Strukturierung .**

Verlangt werden für das erste Halbjahr :

- *** Übungen im Umfang von mindestens 30 Punkten . (Jede Übung hat eine angegebene Punktezahl)**
- *** insbesondere die Übungen 5, 12, 14, 16 .**

1.) Eine Funktion y = f(x) ist in den folgenden Bereichen definiert :

x <= -1 : y = -5	"negativ konstant "
-1 < x <= 1 : y = 0	"null "
1 < x <= 8 : y = x	"linear "
x > 8 : y = 8	" positiv konstant "

Man lese einen reellen Wert für x ein und gebe den entsprechenden Funktionswert mit einer Meldung über den Verlauf (z.B. "linear") aus.

(3 Punkte)

Übungen zu C / C++

2.) Erstellen Sie ein C-Programm, welches die folgenden Operationen nacheinander ausführt :

- Das Programm liest zunächst von der Tastatur eine ganze Zahl ein.

- Anschließend erzeugt das Programm eine ganze Zufallszahl, welche größer oder gleich Null und kleiner als die angegebene Zahl ist.

- Der Benutzer hat nun nacheinander drei Möglichkeiten, die Zufallszahl zu erraten und am Bildschirm einzugeben. Nach jeder Eingabe antwortet das Programm mit der prozentualen Abweichung nach unten oder nach oben.

(3 Punkte)

Anleitung zu der Aufgabe 2 :

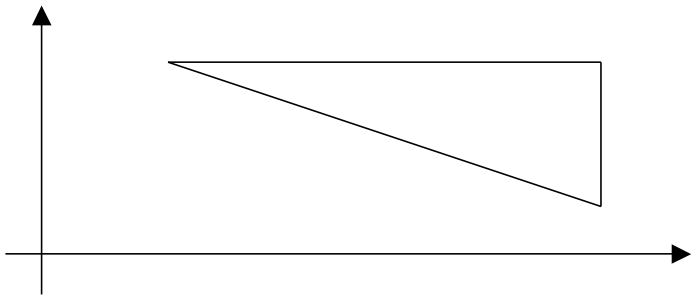
Funktionen :

void randomize() initialisiert den Zufallszahlen - Generator

int random (int num) liefert eine ganzzahlige Zufallszahl im

Übungen zu C / C++

3.) Vorgegeben (Eingabe): drei Wertepaare als Koordinatenwerte von drei Punkten. Ein Programm soll ermitteln, ob die drei Punkte ein rechtwinkliges Dreieck bilden, so, wie in der folgenden Figur gezeigt wird: die beiden Katheten des Dreiecks sollen achsenparallel sein, der rechte Winkel soll nach links bzw. nach unten offen sein.



(2 Punkte)

Übungen zu C / C++

4.) Vorgegeben (Eingabe): Drei Wertepaare (reell)=Koordinaten von drei Punkten A, B, C.
Ein C- Programm soll ermitteln :

a) Ob diese drei Punkte auf einer gemeinsame Linie liegen, oder ob sie ein ein echtes Dreieck bilden. Dabei sind auch senkrechte Strecken zu berücksichtigen!

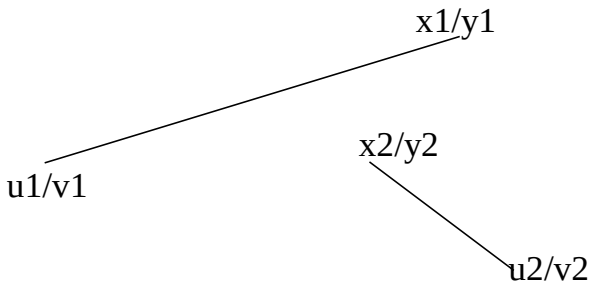
b) Ob das Dreieck gegebenenfalls rechtwinklig ist.

c) Ob das Dreieck gegebenenfalls gleichschenklig ist. **(3 Punkte)**

BA Stuttgart , Elektrotechnik / Informatik WS 2002 / 2003 1. HJ

Übungen zu C / C++

5.) Vorgegeben (Eingabe) : Zwei Punktepaare =>> Endpunkte von zwei Strecken.



Ein C - Programm soll die Koordinatenwerte der Punkte einlesen und berechnen :

a) ob sich die entsprechenden Geraden schneiden. Dabei sind auch senkrechte, waagrechte, sowie auch parallele Strecken zu berücksichtigen.

b) ob der Schnittpunkt (x/y) gegebenenfalls innerhalb einer Strecke liegt.

(5 Punkte)

BA Stuttgart , Elektrotechnik / Informatik WS 2002 / 2003 1. HJ

M. Baum

Übungen zu C / C++

6.) Schreiben Sie ein Programm, welches die Umrechnung von DM in Euro und umgekehrt am Bildschirm tabellarisch anzeigt. Die Umrechnungsrichtung soll über die Tastatur vorgegeben werden können. (1 Euro = 1,95583 DM)

(1 Punkt)

Übungen zu C / C++

7.) Es ist die Summe der Zahlen von 1 bis n (1 + 2 + 3 + 4 + ..) zu berechnen und mit x aus

$$x = \frac{n * (n + 1)}{2}$$

zu vergleichen. **(1 Punkt)**

BA Stuttgart , Elektrotechnik / Informatik WS 2002 / 2003 1. HJ

M. Baum

Übungen zu C / C++

8.) Für n reelle Meßwerte sind die Summe und der arithmetische Mittelwert berechnen und anzuzeigen. Die Anzahl n der Meßwerte ist zu lesen und zur Steuerung einer Programmschleife zu verwenden, in der die Meßwerte gelesen und für die Ausgabe summiert werden. Schleifenabbruch mit Endebedingung !

(2 Punkte)

Übungen zu C / C++

9.) Ein Programm soll den Wert der folgenden Reihenentwicklung näherungsweise berechnen :

$$S = \frac{1}{1*3} + \frac{1}{3*5} + \frac{1}{5*7} + \ldots + \frac{1}{(2n-1)*(2n+1)} + \ldots$$

(2 Punkte)

Übungen zu C / C++

10.) Ein Programm berechne die reellen Nullstellen eines quadratischen Polynoms :

$$y = a_2x^2 + a_1x + a_0$$

durch iterative Halbierung eines x – Intervalls.
Die Werte der Koeffizienten a_2 , a_1, a_0 sowie der Anfangswert und der Endwert des x–Intervalls sollen von der Konsole eingelesen werden.
Die eingelesenen Koeffizienten sind in einem programminternen Feld zu speichern (Erweiterungsmöglichkeit für allgemeine Polynome n-ten Grades).

(2 Punkte)

Übungen zu C / C++

11.) Bei der Addition zweier Matrizen a und b entsteht eine gewichtete Summenmatrix durch die folgende gewichtete Addition der Elemente :

```
c[i][j]  =  ka  *  a[i][j]  +  kb  *  b[i][j]
```

Man lese die Matrizen a und b zeilenweise ein. Die Matrizengröße sowie die Konstanten ka und kb sollen durch Symbolkonstanten festgelegt werden. Die Ausgabe für die Summenmatrix c erfolge wiederum zeilenweise.

(2 Punkte)

Übungen zu C / C++

12.) Ein Programm soll für die in einer Klausur erreichte Punktezahl die zugehörige gerundete Note berechnen. Die Punktezahl ist von der Konsole einzulesen.
Für die Notenzuordnung gilt : Note 1 entspricht 100 Punkten, Note 5 entspricht 20 Punkten, der Gesamtverlauf ist streng linear.
Mögliche „gerundete“ Noten sind :

1,0 1,3 1,7 2,0 2,3 2,7 3,0 3,3 3,7 4,0 4,7 5,0

(3 Punkte)

Übungen zu C / C++

13.) Eine Struktur beinhalte als Elemente die Abmessungen einer Kreisscheibe, wie Radius, Höhe, sowie weitere Angaben dieser Kreisscheibe, wie Kennzeichnung (max 20 Stellen lang) und Volumen.
Das zu entwerfende Programm soll ein Feld für max. 50 solchen Strukturen beinhalten.
Im ersten Teil des Programms werden für einzelne Feldelemente die Daten für Radius, Höhe und Kennzeichnung eingelesen ; der jeweils entsprechende Volumen - Wert wird dabei per Programm unmittelbar berechnet. Die entsprechende Leseschleife wird durch eine Abbruchbedingung beendet. Im zweiten Teil des Programms wird die im Feld gesammelte Information in einer geeigneten Tabelle am Bildschirm ausgegeben.

(3 Punkte)

Übungen zu C / C++

14.) Erstellen Sie ein Programm, welches einen Ausdruck der folgenden Form von der Tastatur als Zeichenfolge einliest, die Zahlenwerte identifiziert, und den Wert des Ausdrucks berechnet und dann ausgibt :

Beispiele :

Eingabe : 12+15	Ausgabe: 12+15 = 27.00
123.5+26,3	123.5+26.3 = 149.80
-25+12	-25+12 = -13.00
15.8-17.27	15.8-17.27 = -1.47
-15.8-17.27	-15.8-17.27 = -33.07

- Der erste Operand darf mit einem Minus-Zeichen beginnen.
- Beide Operanden können ganze Zahlen oder Dezimalzahlen sein.
- Das Resultat soll eine Dezimalzahl mit zwei Stellen nach dem Dezimalpunkt sein.
- Wenn beide Operanden ganze Zahlen sind, soll auch das Ergebnis als ganze Zahl ausgegeben werden.

Hinweis : Die Umwandlungsroutinen atoi und atof sind in stdlib.h deklariert.

(4 Punkte)

Übungen zu C / C++

15.) Schreiben Sie Zeigerversionen zu den folgenden Programmen, zusammen mit entsprechenden Testprogrammen (main()):

a)
```
void Anhang(char s[], char t[])
{ int  i=0,  j=0;
  while( s[i]  !=  '\0')  i++;
  while (s[i++] = t[j++]  !=  '\0') ;
}
```

b)
```
int Svergl(char * s,  char * t)
{  int i;
  for (i=0; s[i] == t[i]; i++)
      if (s[i] == '\0')  return 0;
  return s[i]  -  t[i] ;
}
```

(1 Punkt)

Übungen zu C / C++

16.) Funktion für Skalarprodukt

Ein Programm ist zu entwickeln, in dem eine Funktion das Skalarprodukt von Zwei Vektoren berechnet. Die Zahl der Elemente (Datentyp int) wird an die Werteparameter übergeben. Der errechnete Wert des Skalarprodukts wird mittels eines Referenzparameters zurückgegeben. Im Hauptprogramm wird eine weitere Funktion zur Eingabe eines Vektors aufgerufen.
Diese Funktion gibt den eingelesenen Vektor mittels eines Zeigerparameters (Variablenparameter) zurück. Das geeignete Festlegen von weiteren Funktionsparametern , soweit nötig, ist Teil der Aufgabe.

(3 Punkte)