C / C++ - Teil 2

SS 2003

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

1.) Beispiel einer Entwurfsspezifikation

2.) Felder und Strukturen; typedef .

- **2.1 Mehrdimensionale Felder**
- **2.2 Zeichenvektoren, Funtionen gets(), puts(), cin.getline()**
- **2.3 Vektoren von Strukturen**
- **2.4 Schlüsselwort typedef**
- **2.5 Unions**
- **2.6 Bitfelder**

3.) Zeiger

- **3.1 Adressierung von Daten über Zeiger**
- **3.2 Zeiger und Vektoren, Adressarithmetik**
- **3.3 Char-Zeiger, Zeichenvektoren**
- **3.4 Mehrdimensionale Vektoren, Vektoren von Zeigern, Zeiger auf Zeiger**
- **3.5 Dynamische Speicherverwaltung**
- **3.6 Zeiger auf Strukturen, einfache Verkettung**
- **3.7 Allgemeine verkettete Listen**

4.) Funktionen

- **4.1 Funktionsdefinition; Werteparameter,Ergebnisparameter,Referenzparameter**
- **4.2 Funktionsprototyp; Deklaration versus Definition von Funktionen**
- **4.3 Speicherklassen auto, static, register; Gültigkeitsbereiche für Namen**
- **4.4 Felder, Strukturen und Zeiger als Parameter**
- **4.5 Zeiger auf Funktionen, Funktionsnamen als Parameter**
- **4.6 Inline Funktionen, Makros**

5.) Strukturierung von nicht strukturierten Programmablaufplänen

6.) Präcompiler

7.) Abstrakte Datentypen und Modularisierung

- **7.1 Abstrakte Datentyypen**
- **7.2 Modularität innerhalb einer einzigen C++ Datei**
- **7.3 Mehrere Dateimodule innerhalb eines Projekts**
- **7.4 Visual C++ : Verwalten von Projekten und Projektdateien**

8.) Rekursion

1.) Beispiel einer Entwurfsspezifikation für eine Programmieraufgabe

Die Entwurfsspezifikation umfasst verschiedene Dokumentationen, welche, ausgehend von einer Aufgabenstellung, nacheinander erstellt werden. Die grundsätzliche Ausgangsbasis dabei ist ein konzeptionelles erstes Verständnis für den Lösungsalgorithmus.

Die einzelnen Dokumentationen sind :

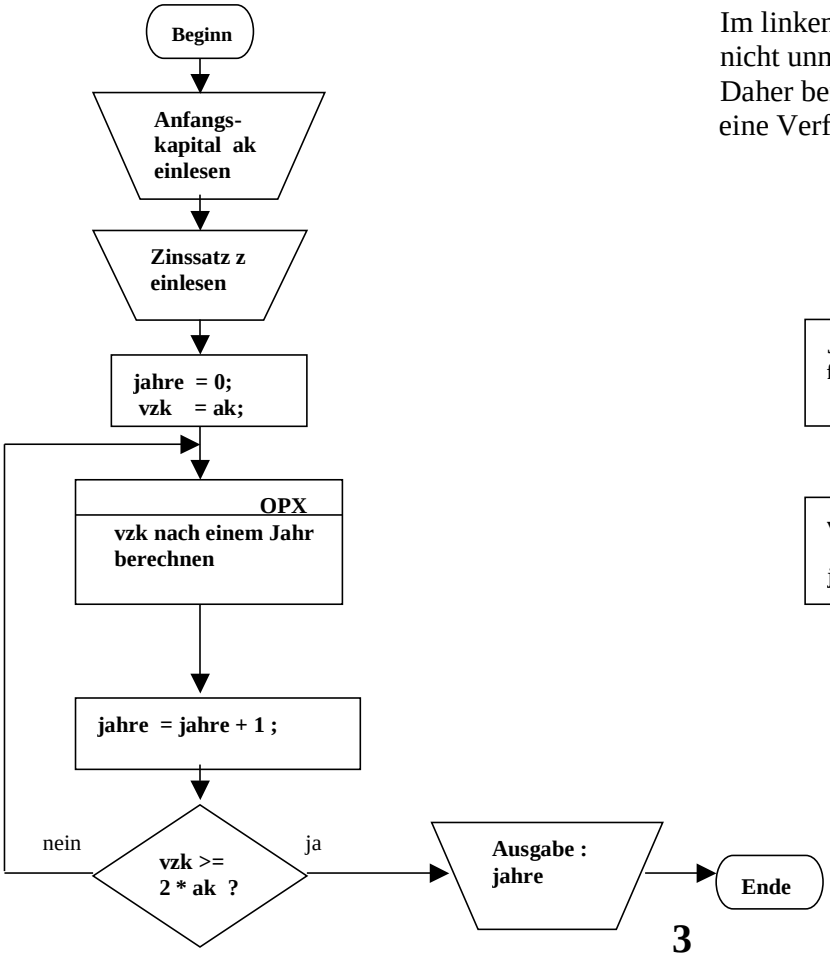
- Beschreibung des Algorithmus in Worten
- Darstellung des Algorithmus durch einen (zunächst groben) PAP / Struktogramm
- Beschreibung der relevanten Datentypen und Datenobjekte
- Darstellung der strukturierten Datentypen durch Datenhierarchien (DH), soweit anwendbar
- Wenn nötig, Verfeinerung des groben PAP's, bzw. einzelner Bausteine davon.
- Darstellung des Algorithmus (PAP) samt seiner E/A - Daten (PN).

Die Behandlung der folgenden Aufgabe dient als einführendes Beispiel :

Aufgabe :
Ein Kapital wird am Ende jeden Jahres mit einem Zinssatz z Prozent verzinst, und der Zins zum Kapital hinzuaddiert. Nach wieviel Jahren ist das Anfangskapital mindestens verdoppelt ?

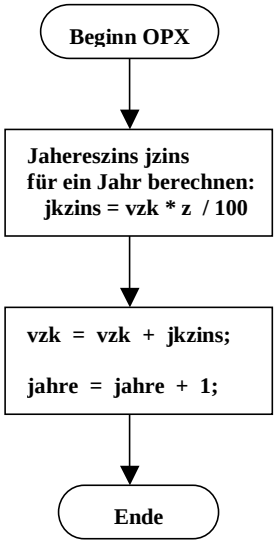
Algorithmus :
Zunächst wird von der Konsole der Wert des Anfangskapitals ak sowie der jährliche Zinssatz z eingelesen. Anschließend wird in einer Programmschleife für jeweils ein weiteres Jahr das verzinste Kapital vzk berechnet und mit dem Anfangskapital ak verglichen. Ist bei dem Vergleich vzk >= ak, so wird Schleifenwiederholung beendet, und schließlich wird die Anzahl der ermittelten Jahre ausgegeben.

Grober PAP



Verfeinerung für OPX :

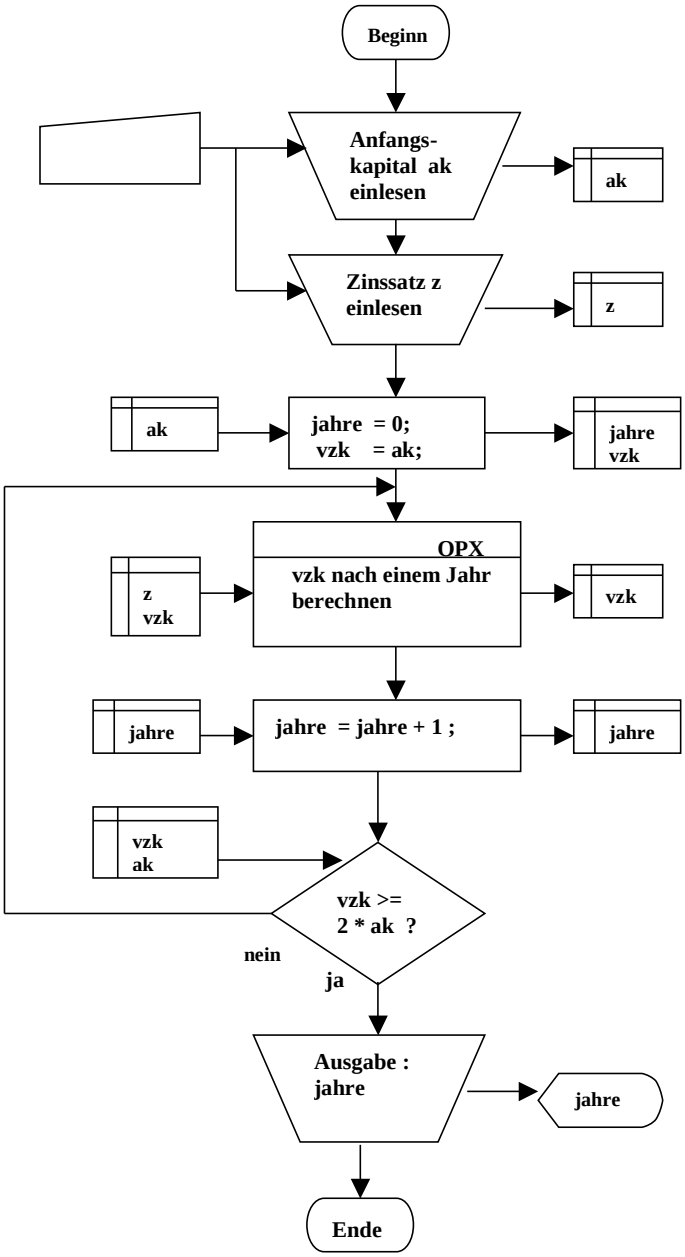
Im linken groben PAP kann die Operation OPX nicht unmittelbar programmiert werden. Daher benötigt die Dartsellung von OPX eine Verfeinerung :



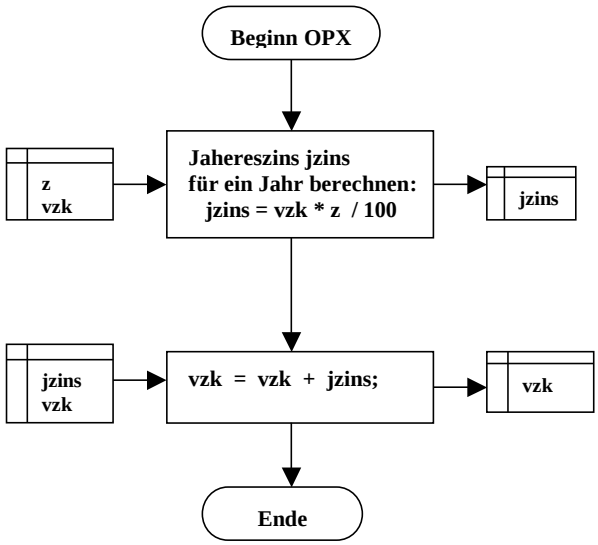
Relevante Datenobjekte :	Typ	Name	Bemerkungen
	float	aka	Anfangskapital
	float	vzk	Verzinstes Kapital
	float	z	Jahreszins
	float	jkzins	Kapitalzins ffür das laufende Jahr
	int	jahre	Anzahl der vergangenen Jahre

Bemerkungen zum PAP : Wann immer in einem PAP eine Operation zu mächtig ist, muß sie verfeinert werden, wie oben die Operation OPX. Das Ziel solcher Verfeinerung ist, jeder Operation des verfeinerten Alogorithmus eine Programmanweisung zuordnen zu können.
Das oben gezeigte PAP-Beispiel zeigt allgemein den Kontrollfluß eines Algorithmus. Im nachfolgend gezeigten **Programmnetz** (PN) wird zusätzlich noch der Datenfluß für den Algorithmus gezeigt : für jede gezeigte Operation sind auch die Ein-/Ausgabedaten mit eingetragen.

Grober PAP mit E/A-Daten (PN)



Verfeinerung für OPX mit E/A-Daten (PN) :



2.) Felder und Strukturen; typedef .

2.1 Mehrdimensionale Felder

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 initialisierten 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 .

Beispiel für zeilenweise Eingabe und Ausgabe :

```
#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]   << "     ";
             }
}
```

Ausgabe am Bildschirm :

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

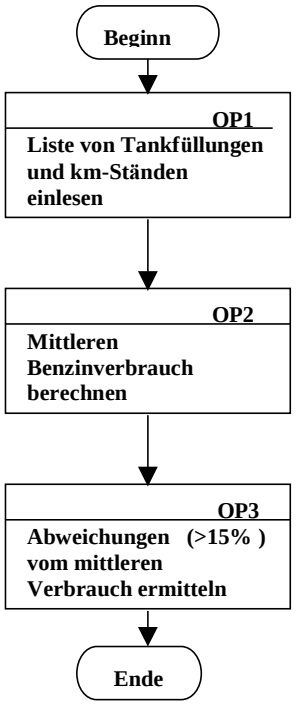
Beispiel :

Ein PKW-Fahrer hat in seinem Notizblock 30 Angaben über die vergangenen Tankmengen und den jeweiligen km-Stand. Dabei hat er bei jedem Tanken voll aufgefüllt.
Ein Programm soll es nun ermöglichen, diese Daten aus dem Notizblock an der Konsole einzugeben. Weiterhin soll das Programm den mittleren Benzinverbrauch errechnen, und anschließend noch auflisten, nach welcher neuen Tankfüllung der Benzinverbrauch seit dem letzten Tanken vom Mittelwert um mehr als 15 % abwich.

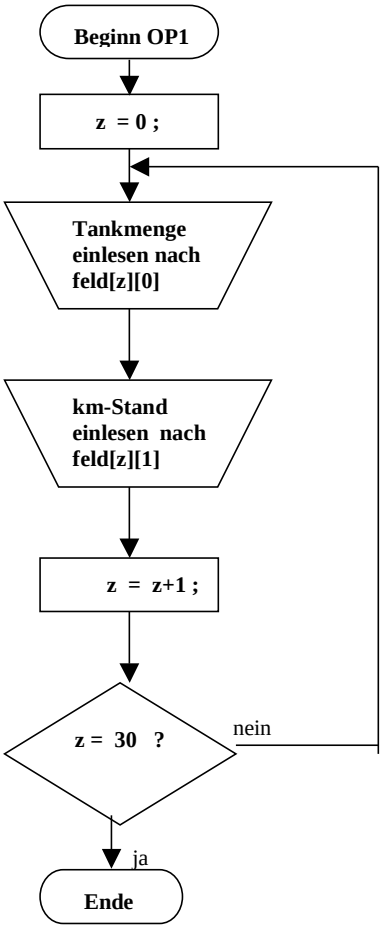
Algorithmus :

Die Daten werden in einen zweidimensionalen Array mit z = 30 Zeilen und s = 2 Spalten eingelesen. Jede Zeile enthält in der ersten Spalte die entsprechende Tankmenge, und in die zweite Spalte den km-Stand dazu.
Die Summe dieser Tankmengen (ohne die allererste), dividiert durch die Differenz vom letzten und erstem km-Stand, ergibt den mittleren Benzinverbrauch. Der individuelle Benzinverbrauch ergibt sich als Quotient aus einer Tankmenge dividiert durch die seit dem letzten Tanken gefahrenen km-Zahl.

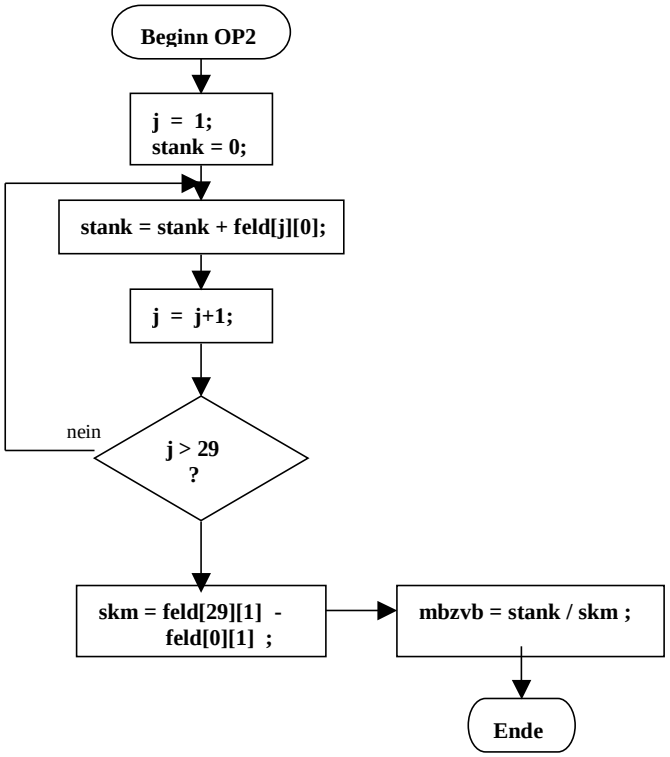
Grober PAP :



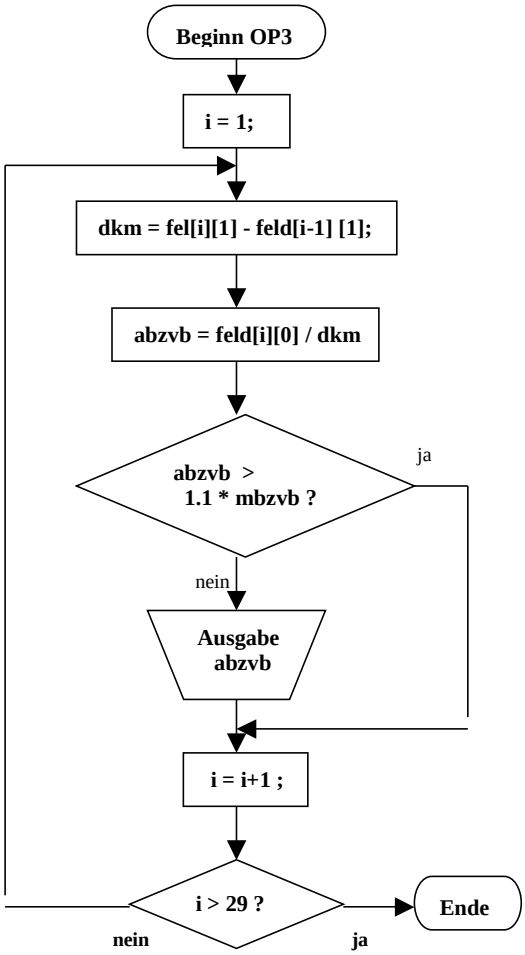
Verfeinerung für OP1 :



Verfeinerung für OP2:



Verfeinerung von OP3 :



2.2 Zeichenvektoren, die Funktionen gets() und puts()

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 z.B. \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 hier 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 Eingabe von Strg+Z (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.

<u>Beispiel für gets und puts :</u>

```
#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.
}
```

<u>Noch ein Beispiel für gets, mit atof (atoi) :</u>

```
#include <stdio.h>
main()
  {  double km;  char string[128];
     printf(“Wie viele km  ? “) ;   gets(string);   // gets liest hier eine  Ziffernfolge als String ein !
     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 Funktionen (#include <string.h>) durchführen.

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

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 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 Einlesen von Text in einen eindimensionalen Vektor:

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

#define  N  81                          // Länge der Textzeile,  Feldgröße
#define  ENDE  "fertig"                 // Endemarke  für Eingabe: ENDE dient als Makroname

main()
{ const char   aus[] = "Ausgabe  => ";  // Textkonstanten ein und   aus
  char  text[N];                        //  Zeichenvektor  text  für maximal N-1 eingegebene Zeichen

  cout  << "\nLeseschleife:  Eingabe von   "   <<  ENDE     "beendet die Eingabe ";

  while ( cout <<  "\nEingabe-> ",
          cin.getline(text, N,  '\n'),
          strcmp(text,  ENDE)       )
                cout <<aus   <<text;            // Vergleichsfunktion strcmp ergibt 0 bei Gleichheit !

   cout  << "\n\nLeseschleife Ende mit Strg und Z";

   while ( cout << "\nEingabe -> ",
           cin.getline(text, N,  '\n'),
           !cin.eof()                    )      //    cin.eof()  liefert   !0 bei Eingabe  von  Strg+Z.
        cout << aus   << text;

  cin.clear;                                    // cin.clear löscht Endbedingung  eof
}
```

Gelegentlich will man **Textfelder** vereinbaren, etwa der folgenden Form :

```
char textfeld[20][81];
```

In der ersten Dimension liegen hier die Textzeilen, wobei jede Textzeile bis zu 80 Zeichen lang sein kann: die letzte Stelle in einer Textzeile muß für den Text - Abschluß (bei maximaler Textlänge = 80) reserviert werden.
Beim Einlesen einer einzelnen Textzeile i können wir schreiben :

```
gets(textfeld[i] );
```

Hier entspricht die Formulierung *textfeld[i]* einer Referenz zu einem eindimensionalen Textvektor.

Im folgenden Beispiel können mehrere Namen nacheinander in eine Text-Tabelle eingegeben werden. Das Programm sortiert anschließend die eingegebenen Namen, und gibt schließlich die sortierte Namensliste am Bildschirm aus :

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

#define   Z  20                    // maximale Amzahl Z  der Textzeilen
#define   N  81                    // N-1 ist die maximale Textlänge, in jeder Zeile

void main()
{  char  tabelle[Z][N] ;    //  tabelle enthält  Z  Zeichenvektoren der maximalen Länge N
   char temp[N];            //   temporäres Feld beim Sortieren

                            //  ------------   Zunächst die Eingabe der Namen  :  --------------------
   for ( int i = 0; i<Z; i++)
    { puts("Bitte, einen Namen eingeben, Ende mit Strg+Z ");
      if ( gets( tabelle[i] ) == NULL) break;                    //  Text in Zeile i einlesen
    }

                            // ------------------------- Nun wird sortiert :   Bubble Sort   -----------
  int ober = i-1 ,          //  Index vom letzten eingelesenem Namen
      u ;                   //   Laufvariable u

  for ( int ober = i-1; ober > 0;  ober--)
     for ( u=0; u<ober;  u++)
        if (strcmp(tabelle[u], tabelle[u+1] )  > 0 )
          {  strcpy( temp, tabelle[u+1] );              //  Vertauschen der Elemente
             strcpy( tabelle[u+1], tabelle[u] );
             strcpy( tabelle[u], temp );
          }

                            //------------ Nun folgt die sortierte Ausgabe :    --------------------------

   for ( int j = 0; j<i; j++)  printf("\n  Zeile = %d, Text = %s ", j , tabelle[j]  ) ;

   return;
}
```

2.3 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] ;
```

Mit solchen Deklarationen und Definitionen erhält man eine gute Übersicht bzw. 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.
```

2.4 typedef struct

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

```
typedef float Gleikomma ;
```

der Name Gleitkomma synonym zu float.

Die Typbezeichnung Gleitkomma kann nun bei nachfolgenden Deklarationen verwendet werden :

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

Ein Vergleich :

```
#define   Text              Ersatztext    //   kein Semikolon !
typedef   Standard-Typ   neuer-Typname  ;
```

Für die mittels typedef definierten Namen gelten die normalen Regeln bezüglich Gültigkeitsbereich.

Ein Beispiel :

```
typedef  int BOOL;

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

Eine typedef - Vereinbarung konstruiert keinesfalls einen neuen Datentyp, sonder es wird lediglich ein zusätzlicher Name für einen existenten Typ eingeführt.

```
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 .
Im obigen Strukturbeispiel wird der Strukturtyp 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
```

Nachfolgend werden verschiedene Möglichkeiten zur Deklaration von Vektoren von Strukturen an einem einfachen Beispiel vergleichend betrachtet.

Für die Koordinaten eines Punktes in der Ebene können wir eine Struktur deklarieren :

```
struct Punkt  {  float  x ;
                 float  y ;
              };
```

Um einen Vektor von solchen Punkt - Strukturen zu deklarieren, bieten sich die folgenden Möglichkeiten an :

Möglichkeit 1 : Deklaration : `Punkt  punkteliste[3];`

Als Referenz für x in der zweiten Struktur gilt : **punkteliste[2].x**

Nachteil : Es fehlt ein expliziter Typ für den Gesamtvektor punkteliste[3], d. h. , der Gesamtvektor kann nicht als Parameter in einer Funktion verwendet werden.

Möglichkeit 2 : Deklaration :
```
typedef  Punkt  Punkteliste[3];
Punkteliste  punkteliste;
```

Als Referenz für x in der zweiten Struktur gilt : **punkteliste[2].x**

Vorteil : Punkteliste ist der Datentyp für den Gesamtvektor punkteliste. Der Gesamtvektor kann also als Funktionsparameter verwendet werden.

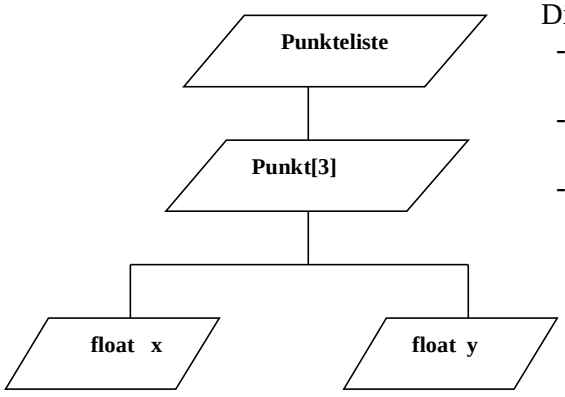
Möglichkeit 3 : Deklaration :
```
struct Punkteliste  { Punkt pkt[3];  };
Punkteliste  punkteliste;
```

Als Referenz für x in der zweiten Struktur gilt : **punkteliste.pkt[2].x**

Vorteil : Auch hier ist Punkteliste der Datentyp für den Gesamtvektor punkteliste. Auch hier dient der Gesamtvektor gegebenfalls als Funktionsparameter.

Vorteil gegenüber Möglichkeit 2 : Die Referenz - Notation spiegelt hier explizit den Aufbau des Vektors wieder : Element x vom 2. pkt in punkteliste .

Für die eben dargestellten Möglichkeiten 2 und 3 kann die hierarchische Relation der Datentypen gemäß DIN 66001 als ein **Datenhierarchie - Diagramm (DH)** dargestellt werden :



Diese Darstellung ist wie folgt zu lesen :
- Typ Punkteliste besteht aus drei Elementen vom Typ Punkt
- Typ Punkt besteht aus zwei Komponenten vom Typ float
- Die beiden Komponeneten heißen x bzw. y

2.5 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.

2.6 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, wobei 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.
Zu beachten ist, daß ein Programm mit Bitfeldern unter Umständen nicht mehr portabel ist !

3.) Zeiger

3.1 Adressierung von Daten über Zeiger

Ein Zeiger ist eine Variable, welche die Adresse einer Speicherstelle enthält.
Zeiger werden in C echt häufig dazu benützt,

- -- um Strukturen bzw. Struturelemente zu adressieren,
- -- um verkettete Listen programmtechnisch zu verarbeiten,
- -- um Datenbereiche im sog. dynamischen Speicher zu verwalten.

Wie später noch gezeigt wird, besteht auch eine besondere Verwandtschaft zwischen Zeigern und Vektoren und ebenso zwischen Zeigern und referieten Zeichenketten. Diese Zeigertechnik eralubt in C einen einfachen Umgang mit Strings.

Bei einer Variablen unterscheiden wir grundsätzlich zwischen dem Inhalt eines Speicherplatzes, nämlich dem Wert der Variablen, und der Adresse des Speicherplatzes, d.h. der Speicheradresse dieser Variablen.

Die Speicheradresse der Variablen y ergibt sich aus &y, dabei ist & ein Adress-Operator.

Um solche Adresswerte als spezielle Datentypen zu behandeln, stellt die Programmiersprache C einen **besonderen vordefinierten Datentyp Zeiger (engl. pointer)** zur Verfügung : **dazu wurde in C kein eigenes Schlüsselwort für Zeiger reserviert**, **sondern man erweiterte die Bedeutung des Zeichens '*' .**

```
int  x,y  = 20;
int   *  ip;                  //    ip ist eine Zeigervariable vom Typ int.

ip   =  &y;                   //   nun  beinhaltet  ip die Adresse von y,
                              //   à  Wir sagen :  ip zeigt  auf  y
```

Der Stern (*) ist hier nicht Bestandteil des Namens und kann auch durch Leerzeichen von ihm getrennt werden. Wir können auch formulieren :

```
int* iq;              //  Zeiger iq
int  *iz;             //  Zeiger iz
```

Ausgehend von einem solchen Adresswert kann man auf den Inhalt der adressierten Speicheradresse zugreifen :

```
x  =  *ip ;    // gleichbedeutend zu  x  =  y,      *ip ist hier also identisch zu  y
```

Den Ausdruck *ip (allgemein *Zeigerbezeichner) bezeichnet man als **Indirektion** oder **Dereferenzierung;** das Zeichen * ist hier ein **'indirection– Operator ' :**

nimm nicht den Zeiger (ip), sondern die durch den Zeiger referierten Daten.

Die Formulierung *ip darf überall stehen, wo in unserem Beispiel y stehen würde :

```
*ip  =  *ip  +  1 ;          //  gleichbedeutend zu  y  =  y  + 1
```

In Ausdrücken kennzeichnet der indirection-Operator * vor dem Bezeichner, daß die durch den Zeiger adressierten Daten als Operand verwendet werden sollen. Dabei ist auf den Unterschied zur Multiplikation zu achten :

die Anweisung `x = 3**ip;` ist hier gleichbedeutend zu `x = 3 * y;`

Ist iq ein weiterer Zeiger auf int, dann gilt auch : iq = ip ; Hier wird der Adressinhalt von ip nach iq kopiert ; ip und iq zeigen anschließend auf die gleiche Stelle im Hauptspeicher. Adressen können also gespeichert werden wie Zahlen oder Zeichen .

Die unären Operatoren * und & haben höheren Vorrang als arithmetische Operatoren.

```
*ip  += 5;                  //  gleichbedeutend zu  y  +=  5;

++*ip ;  (*ip)++;           // gleichbedeutend zu  ++y ;  y++ ;
```

In *ip++ wird zunächst der Zeiger ip inkrementiert, und nicht das von ip referierte Objekt !
In (*ip)++ sind die Klammern nötig, da unäre Operatoren wie ++ und * von rechts nach links zusammengefaßt werden .
Anstatt i = 10 /*p_int; schreiben wir besser i = 10 / (*p_int).

Namen für Zeigervariablen sollten von ihrer Bezeichnung her eindeutig auf Zeiger-Typen hinweisen .

Eine Variablenvereinbarung kann sowohl Daten als auch Zeiger auf Daten definieren :

```
int  x, y=20, * iq = &y ,  z;      //  hier sind x, y und z int-Variablen, iq ist eine Zeigervariable
```

Ein definierter Zeiger darf immer nur auf einen bestimmten Datentyp zeigen, entsprechend seiner Deklaration, mit Ausnahme von 'Zeiger auf void ', doch darüber später mehr !

Alle Zeigervariablen belegen 4 Bytes, unabhängig vom referierten Datentyp.

Zeigervariablen sind Adressvariablen; nur mit static oder global deklarierte Zeiger haben den Anfangswert NULL und werden im statischen Speicherbereich angelegt. Wie oben gezeigt, können Zeigervariablen bei ihrer Deklaration initialisiert werden.

Initialisierung von Zeigervariablen bei der Deklaration :

```
int    x = 0,  *  ip     =  &x;;
char*    cp    =  NULL;
```

Hier wurde ip mit dem Adresswert von x bei der Definition initialisiert. In stdio ist NULL als 'Nullpointer' mit dem Wert 0 definiert. Die Zeigervariable cp ist hier mit diesem Anfangswert initialisiert.

Ein weiteres Beispiel :

```
int  x,  *ip =  &x,  *iq = ip;
```

Hier wird iq mit dem Zeigerwert von ip initialisiert; dies könnte auch entsprechend durch eine spätere Wertzuweisung (iq = ip ;) geschehen.

Die folgenden if-Anweisungen sind gleichwertig :

```
if( ip != NULL)..    und        if(ip)...
```

Konstante Zeiger, deren Inhalt nicht mehr verändert werden kann, erhalten zwischen dem Operator * und ihrem Bezeichner **das Modifizierkennwort const :**

```
float * const  fp = "abcd";       //  fp zeigt immer auf den  gespeicherten Text   "abcd".

        fp  = "abend";     //  Fehler :  fp ist als Zeigerkonstante deklariert.
```

Die durch solche konstanten Zeiger adressierten Daten können variabel sein :

```
int x;
float   * const  fq = &x;    //  fq zeigt immer nur auf die Variable x;
```

Eine Zeiger - Deklaration mit vorangestelltem const macht das adressierte Objekt, nicht den Zeiger konstant :

```
const float * zeiger = "abcd" ;
zeiger = "morgen";    //   Die Zeigervariable zeiger kann ver ändert werden.
```

Wird bei einer Funktionsdeklaration in der formalen Parameterliste ein Pointer-Argument vom Typ const * float spezifiziert, ist damit sichergestellt, daß die Funktion das referierte Objekt nicht ändert.

Um beide, Zeiger und Objekt als konstant zu deklarieren, muß die Deklaration zwei const enthalten :

```
const float  *  const fr  = "gestern";  //  fr ist eine Zeigerkonstante,
                                        //  "gestern"   wird als eine Zeichenkonstante behandelt.
```

Es gibt globale, statische und lokale ('automatische ') Zeigervariablen. Besonders bei lokalen Speichervariablen ist auf die nötige Initialisierung bei der Definition bzw. durch explizite Wert - Zuweisung zu achten . Auf dynamische Variablen geht ein späteres Kapitel ein.

Zeigerfehler sind recht schwierig zu lokalisieren ! Zeigervariablen sollten bei der Deklaration oder unmittelbar danach initialisiert werden.

Weitere Regeln im Umgang mit Zeigern sind :

- Mit Hilfe des cast-Operators können Werte einer Zeigervariablen einer Zeigervariable eines anderen Datentyps zugewiesen werden :

```
int  i,    *ip =  &i ;
float  *fp  = (float *) ip ;     // fp erwartet in &i einen float – Wert  !
```

- Zeigervariablen vom gleichen Typ dürfen mit Hilfe der Vergkleichsoperatoren verglichen werden.

- Zeigerwerte, also Adressen, können nicht wie Daten von der Konsole gelesen werden.

- Zeigervariablen dürfen nicht mit Zahlenkonstanten besetzt werden

- Der Inhalt eines Zeigers läßt sich mit printf (Format %p) bzw. mit cout hexadezimal ausgeben :

```
int i,   *ip = &i;
cout << "Adresswert : "  << ip ;  //  Adresse hexadezimal !
```

Beim Aufruf der Eingabefunktion scanf werden Adressen als Parameter erwartet, die zur Übergabe der einzulesenden Werte erforderlich sind:

```
 double  wert= 5.648,  *pw = &wert,  z[100] ;

scanf("%lg", &wert) ;    // direkte Adressierung
scanf("%lg", pw);        // indirekte Adressierung über den Zeiger pw
scanf("%lg ", z );       // z[0] wird eingelesen, da z den Adresswert von z[0] beinhaltet.
```

3.2 Zeiger und Vektoren, Adressarithmetik

In C besteht eine besonders enge Beziehung zwischen Zeigern und Vektoren.
Jede Operation mit Vektorindizes kann auch mit Zeigern formuliert werden, wie im folgenden gezeigt wird.

```
int   a[20] ,  *pa ;   // Der Vektorname  a ist hier eine Adresskonstante, es zeigt auf a[0] .
pa  =  &a[0] ;       //   pa enthält anschließend die Adresse von  a[0];  auch so:  pa=a;
                     //   pa+1 zeigt nun auf das nachfolgende Element, a[1],
                     //    pa+5 auf das 5.  Element hinter  a[0],     a[5] .
```

Da nun pa auf a[0] zeigt, ist die Adresse von a[i] gleich pa+i,
und *(pa+i) ist der Wert von a[i] (auch pa[i]).
Nach der obigen Zuweisung 'pa = a ;' bedeuten a+i und pa+i den gleichen Adresswert, nämlich die Adresse von a[i] bzw. von pa[i].
Formulieren wir pa = &a[8]; , dann können wir auch entsprechend pa-5 bilden.

Der Name eines Vektors ist synonym für die Adresse des Anfangselements, d.h. ein Vektorname wird wie ein Zeiger behandelt, er ist eine Adresskonstante.

Die Zeigerarithmetik arbeitet in der Länge des adressierten Datentyps :
Addieren eines int-Wertes i zu einer Zeigervariablen verändert den Adresswert in der Zeigervariable um die i-fache Länge des entsprechenden Datentyps.

Diese Adress-Arithmetik ist in C konsistent für Zeiger und Vektoren integriert.
Die Ausdrücke p+i, i+p, p-i sind Pointer–Werte, dagegen ist p - p vom Typ integer.
Zu beachten ist noch, daß für die Zeigerkonstante a **die Formulierung a++ nicht erlaubt ist.**
Ist dagegen pa ein Zeiger auf ein Vektorelement, dann ändert pa++ die Zeigervariable pa so, daß pa auf das nächste Element zeigt.

Der Bezeichner einer Struktur wird bei der Adressierung durch Zeiger wie eine normale Variable behandelt!

Weitere Regeln für Zeiger sind :

-- Addition und Subtraktion einer ganzen Zahl zu einem Zeiger ist möglich
-- Zwei Zeiger dürfen nicht additiv verknüpft werden.

Zu beachten ist der Unterschied zwischen

```
y = *zeig+1;        und     y = *(zeig+1);
```

Im ersten Fall wird der Inhalt der von zeig adressierten Speicherstelle um 1 erhöht und das Ergebnis in y abgespeichert.

Im zweiten Fall wird der Adresswert von zeig um die Datentyp-Länge erhöht und dann der Inhalt der neu adressierten Stelle in y abgespeichert.

Ein Beispiel : Vektorelemente können über Zeiger angesprochen werden.

```
float *fpointer,   xfeld[40];
fpointer = xfeld ;                  // in fpointer wird Startadresse von Vektor xfeld[40] gespeichert.

for (int i=0; i<= 39; i++)          // Alle Vektorelemente werden mit 1.0 initialisiert.
      *(fpointer + i ) = 1.0;       // auch feld[i], oder *(xfeld+i), oder auch fpointer[i] )
```

Noch ein Beispiel : Vektorelemente einlesen, mit einem Skalar multiplizieren, und wieder ausgeben.

```
#include <iostream.h>

void main()
{     float feld[10], skalar ;
      int iz = 0;                               // Laufvariable iz

      cout << "\nBitte, Skalar eingeben : " ;  cin >> skalar ;
      while ( iz < 10 )
            {   cout << "\n Bitte, Feldelement "  << iz  ;
                cin >>  * (feld + iz ) ;
                *(feld+iz) = *(feld+iz) * skalar ;      // auch so : feld[iz] = feld[iz] * skalar;
                iz++;
             }

      cout << "\n Feldkomponenten : " ;
      while ( iz > 0 )                          // Ausgabeschleife beginnt mit letztem iz-Wert .
             {   cout << * ( feld + 10 - iz )   << "      " ;
                 iz--;
              }
       return ;
}
```

Die Formulierung

y = *zeig1 * * zeig2;

bedeutet die Multiplikation der durch zeig1 und zeig2 adressierten Werte.

Der *-Operator kann nur auf Zeiger angewandt werden. Die Formulierung *512 ist also nicht erlaubt. Allerdings kann man sich hier mit dem cast-Operator helfen :

***(int *)512**

Mit dieser Formulierung erreicht man den Inhalt der Adresse 512.

Das folgende Beispiel zeigt eine Anwendung dafür: es besorgt die Ausgabe der Inhalte der ersten 48 Bytes des Hauptspeichers.

```
main()
{    unsigned char  inhalt,i;

     printf ("\n\n");

     for (i= 0; i < 48; i++)
         {  if  ( (i  %  8 ) == 0 )  printf("\n");
             inhalt = *(char * ) i;              //  Inhalt der Speicherstelle i
             printf("  %2x", inhalt );
         }

      return 0;
 }
```

3.3 Char-Zeiger, Zeichenvektoren

Zeichenvektoren wurden schon kurz behandelt. Hier sei nochmals darauf hingewiesen, daß auf eine konstante Zeichenkette (String) über Zeiger zugegriffen wird :

```
printf("Heute ");
```

Die Zeichenkette "Heute " wird vom Compiler als Textkonstante abgespeichert, mit zusätzlichem '\0' am Schluß, und die Funktion printf erhält als Argument einen Zeigerwert auf den Anfang dieser Zeichenkette.

Steht irgendwo eine Stringkonstante im Programm, so wird zu ihrer Referenz immer ihre Anfangsadresse genommen.

Für die Initialisierung eines deklarierten Zeichenvektors gilt :

```
char  t[]  =  "Rechner" ;

char  *p   = "Rechner ";
```

Bei der Initialisierung für das Feld t[] wird ein Feld genügender Größe für den String einschließlich dem Stringendezeichen angelegt.

Bei der Initialisierung für den char-Zeiger p wird zunächst für die Textkonstante "Rechner " einschließlich dem Stringendezeichen ein Speicherplatz angelegt, und dann wird in p die Anfangsadresse dieses Strings abgespeichert.
Für die Wertzuweisung gilt :

```
p  =  "Montag " ;
```

Wie bei der Initialisierung wird hier in p die Anfangsadresse der Textkonstante "Montag " abgespeichert. Dabei wird die Zeichenkette nicht kopiert, an einer solchen Operation sind nur Zeiger beteiligt.

Dagegen ist (bei obiger Deklaration für t) die Formulierung

```
t  = "Montag ";
```

falsch, weil t eine Zeigerkonstante ist und immer die Anfangsadresse des definierten Feldes beinhaltet. Die Zeigerkonstante t kann nicht verändert werden !

C/C++ hat keine Operatoren, welche eine Zeichenkette als Einheit behandeln.
Für die Operationen mit Zeichenketten ist entweder ein Zugriff auf einzelne Zeichen dieser Strings nötig, oder die Benutzung von vordefinierten Bibliotheksfunktionen (#include <string.h>).

Beispiel für die Operation Kopieren, mit Zugriff auf einzelne Zeichen :

```
char  ziel[100],  quelle[100] = "Ein Programm";
int  i=0;

while (quelle[i]   !=  '\0' )
    {   ziel[i]   =  quelle[i] ;          //  Kopieren durch Zugriff auf einzelne Zeichen
        i++;
    }
ziel[i]  =  '\0' ;                        // Das Stringende muß extra kopiert werden.
```

Das gleiche Beispiel, mit Benutzung einer vordefinierten Funktion :

```
char ziel[100],  quelle[100] = "Ein Programm";

strcpy(ziel, quelle);           // Das Stringende wird mit kopiert
```

Die folgenden vier Programmbeispiele zeigen mögliche Variationen unseres obigen Kopierprogramms.

a)
```
char  ziel[100],  quelle[100] = "Ein Programm";

int i= 0 ;
while (   (ziel[i]  = quelle[i] )   !=  '\0'  )
        i++;
```

b)
```
char ziel[100],  quelle[100] = "Ein Programm";
char * z = ziel,    *q = quelle ;             // Hier werden nur Zeigerwerte bewegt

while ( (*z  =   *q)    !=   '\0'  )
     {  z++;  q++;    }
```

c)
```
char ziel[100], quelle[100] = "Ein Programm";
char * z=ziel,  *q = quelle ;

while ( ( *z++  = *q++)  !=  '\0'  );  //  Die Inkrementierung  von  z und q wurde hier in die
                                       //  Bedingung  verlegt.
```

d)
```
char ziel[100], quelle[100] = "Ein Programm";
char  *z=ziel,  *q = quelle ;

while  ( *z++  =  *q++ );  //  Ein Vergleich mit '\0' ist unnötig, da es nur darum geht, ob der
                           //  Ausdruck Null ist.
```

Die nachfolgende Zusammenstellung zeigt Funktionen für Operationen mit Zeichenketten :

Aufgabe	**Funktionsaufruf**	**Ergebnis der Funktion**
kopiere Strings: Quelle nach Ziel	strcpy(Ziel, Quelle)	Zeiger auf Anfang
hänge Quelle an Ziel an	strcat(Ziel, Quelle)	Zeiger auf Anfang
vergleiche Strings alphabetisch	strcmp(Strng1, Strng2)	< 0 0 >0
Länge eines Strings	strlen(Strng)	Anzahl der Zeichen, ohne '\0'

Beispiele für mögliche Probleme mit Zeigern :

1.
```
char  *ziel.
strcpy  (ziel,  "Irgendwo");
```

Hier ist die Zeigervariable ziel noch nicht initialisiert, und damit steht auch noch kein Speicherplatz zum Kopieren zur Verfügung.

2.
```
char * ziel = "J";
strcpy(ziel,"Überschreiben";
```

Hier enthält ziel zwar eine aktuelle Speicheradresse, aber es steht dort nur 1 Byte zur Verfügung. strcpy überschreibt hier die nachfolgenden Speicherplätze !

3.4 Mehrdimensionale Vektoren, Vektoren von Zeigern, Zeiger auf Zeiger

Wir gehen aus von einer zweidimensionalen Matrix:

```
float  zweidim[3][4] ;
```

Mit zweidim[i][j] spricht man den Inhalt des Matrixelements in der (i+1)-ten Zeile
und (j+1)-ten Spalte an.

Die Adressorganisation einer solchen Matrix ist zweistufig aufgebaut :

Die Formulierung zweidim[i] kennzeichnet die Adresse der (i+1)-ten Zeile,
ist also eine Zeigerkonstante 1. Stufe, vom Typ float*.

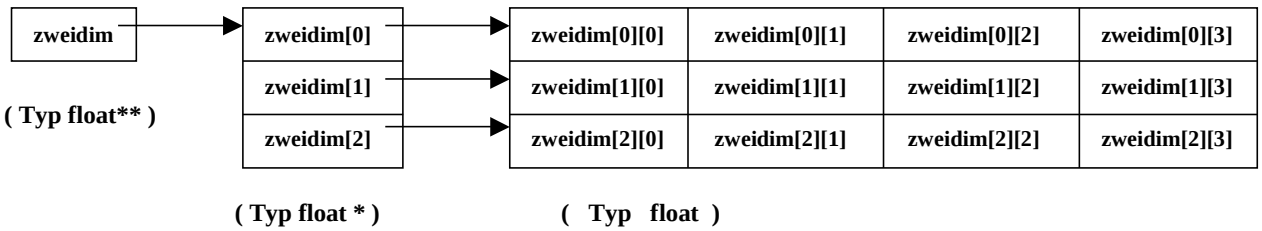
zweidim[0] zeigt auf den Beginn der ersten Zeile, zweidim[1] zeigt auf den Beginn der zweiten Zeile.

Die Folge der Adresskonstanten zweidim[0] zweidim[1] zweidim[2] wird im Speicher als ein eindimensionaler Vektor von Zeigern (Zeigerfeld) gehalten.

Der Name zweidim selbst ist auch eine Adresskonstante: dieser sogenannte zweistufige Zeiger zeigt auf den Beginn des Zeigerfeldes, also auf zweidim[0].

Der Matrixname zweidim ist vom Typ float **, wir sprechen auch vom Typ *Zeiger auf Zeiger :*
er zeigt auf einen Zeiger vom Typ float * .

Die folgende Figur illustriert diese Zusammenhänge :



Beispielsweise ist **zweidim[2] + 3** ein Zeigerwert und zeigt auf das 4. Element der 3. Zeile. Der Inhalt dieser Speicherstelle wird mit ***(zweidim[2] +3)** oder auch mit **zweidim[2][3]** angesprochen.

zweidim ist ein zweistufiger Zeiger vom Typ einer Matrixzeile : zweidim+1 zeigt auf zweidim[1] und damit mittelbar auf die zweite Zeile.

Wir können noch formulieren :

```
zweidim[i][0]      ist äquivalent zu      *zweidim[i]
zweidim[i][j]      ist äquivalent zu      *(zweidim[i]+j)
zweidim[0][0]      ist äquivalent zu      ** zweidim
zweidim[i][j]      ist äquivalent zu      *(* (zweidim+i)  +j)
zweidim[i][j]      ist äquivalent zu      *(zweidim+i)[j]
```

Daher können wir zweidim[5][7] auch mit *(*(zweidim+5) +7) ansprechen.

Die einstufigen Zeigerwerte (Typ *) können wir in einer einfachen Zeigervariablen speichern :

```
float    * zeiger;
zeiger =  zweidim[3];    // Adresse der 4.ten Zeile
```

Wir können auch einen Vektor aus einstufigen Zeigervariablen bilden :

```
float  * zeiger[10];      // Zeigervektor
for  ( int i=0; i < 10; i++)    zeiger[i]  = zweidim[i];
```

Jetzt wird sowohl mit zweidim[i][j] und zeiger[i][j] das gleiche Element der Matrix angesprochen.

Bei der Zeigervariablen entsprechend dem zweistufigen Zeigertyp zweidim brauchen wir noch eine Spaltenangabe :

```
float ( *zeig)[20] ;          // zweistufiger Zeiger, Typ **
```

Hier ist nun zeig eine zweistufige Zeigervariable mit der Zusatzangabe einer festen Spaltenanzahl.

Die extra Klammern sind nötig, um von float *zeig[20] zu unterscheiden, das ein 20-stelliges Feld von einstufigen Zeigern bedeutet. Solche zweistufigen Zeigervariablen mit Angabe der Spaltenlänge zeigen auf zweidimensionale Matrizen und eignen sich speziell zur Übergabe von Matrizen an Funktionen.

Durch die Anweisungen

```
zeig   =  zweidim ;
zeig++;
```

erhält zeig zunächst den Adresswert der Zeigerkonstanten zweidim, anschließend erhält zeig die Startadresse der zweiten Zeile dieser Matrix.

Jeder * oder jede eckige Klammer bedeutet eine Stufe für den Zeigertyp.

Allgemeine Vorteile von Zeigern :

-- einfacher Umgang mit Strings
-- schnelle Vektorrechnung und Matrizenrechnung
-- Parameterübergabe bei Funktionen
-- notwendig bei dynamischer Speicherzuweisung

Die Adressierung der Zeilen einer Texttabelle kann eindimensional erfolgen, wie im folgenden Beispiel gezeigt wird :

```
#define  NZ  5          // Zeilen
#define  NS 11         //  Spalten + 1
main()
 {  int i;
    char text[NZ][NS]  =  {"eins","zwei","drei","vier","fünf" } ;    // auch   char text[][NS]  wäre richtig !

    for ( i = 0; i < NZ; i++)   cout << text[i] ;      //   zeilenweise Ausgabe

    return 0;
 }
```

Diese Tabelle ist ein fester zusammenhängender Array mit 11 Spalten : wir können in jeder Zeile einen Text der maximalen Länge 10 Zeichen (plus Textende) abspeichern. Die bei der Initialisierung angegebenen Textkonstanten werden in der Tabelle zeilenweise abgespeichert, jeweils mit einem zusätzlichen Textende-Zeichen. Eine Textkonstante mit mehr als 10 Zeichen wird hier vom Compiler als fehlerhaft angezeigt.

Bei der Ausgabe mit cout wird jeweils der Text einer Zeile ausgegeben.

Vergleichen wir damit einen deklarierten Zeigervektor :

Da Zeiger selbst Variablen sind, können sie genau wie andereVariablen in Vektoren zusammengefaßt werden :

```
char   *  zeilenzeig[5];
```

Hier wird zeilenzeig auch als Zeigervektor bzw. als Vektor von Zeigern bezeichnet. Jedes Element von zeilenzeig ist ein Zeiger auf einen char-Wert.

zeilenzeig[i] zeigt hier auf den Beginn der (i+1).ten gespeicherten Zeichenkette, und *zeilenzeig[i] ist das erste Zeichen dieser (i+1).ten gespeicherten Zeichenkette.

Die einzelnen Zeichenketten können irgendwo im Speicher liegen und ganz verschiedene Längen haben, wie sich aus den folgenden beispielhaften Zuweisungen ergibt :

```
zeilenzeig[0]  = " erste Zeile";
zeilenzeig[1] = "ein Programm";
```

Das folgende Beispiel zeigt eine Initialisierung eines solchen Zeigervektors :

```
 char  *  monat [] = { "Kein Monat",
                       "Januar",
                       "Februar",
                       "M ärz",
                       "April"  } ;
```

Die Anzahl der Elemente ist hier : sizeof(monat) / sizeof(monat[0])

Diese Initialisierung ist eine Liste von Zeichenketten. Jede einzelne Zeichenkette wird irgendwo im Speicher plaziert, und in monat[i] wird die Anfangsadresse der i-ten Zeichenkette abgespeichert. Der Compiler zählt die Zeichenketten und dimensioniert den Zeigervektor entsprechend.

Nochmals ein Gegenbeispiel :

```
char amonat[][15] = {"Kein Monat", "Januar", "Februar", "März", "April" };
```

Hier ist amonat ein fester zusammenhängender Array, mit einer Zeilenlänge von 15 Bytes. Die Initialisierung bestimmt hier die Anzahl der Zeilen des Arrays.

Der wichtigste Vorteil des obigen Zeigervektors sind die möglichen verschieden langen Zeilen.

Beispiel für Zeichenketten :

```
#include   <stdio.h>

void main()
{
   char  * artikel [] =    {   "Schraube" , // Jeder String hat am Ende ein Stringendezeichen \x00
                               "Mutter" ,
                               "Scheibe",
                               "Werkzeug"
                           } ;

   printf("\n%s",  &artikel[2][0] ) ;  //Ausgabe von  Scheibe (beachte die Formatangabe  %s ! )
   printf("\n%s",  &artikel[2][3] );                    // Ausgabe von eibe
   printf("\n%c", artikel[2][3] );      // Ausgabe von e   ( beachte die Formatangabe %c !)

   return;
}
```

Ein Zeiger kann nicht nur auf Daten, sondern auch auf andere Zeiger zeigen; der Zeigeroparator läßt sich mehrfach anwenden :

```
double    x= 1.23,  *px = &x ,  **ppx ;    // ppx  zeigt auf Zeiger auf double.
ppx  =  &px;
```

Nach dieser Vereinbarung weist **px auf die durch px adressierte Daten (x) hin.

```
cout << "\n" << **ppx ;   // Ausgabe von x-Wert.
```

3.5 Dynamische Speicherverwaltung

Lokale, also innerhalb von main() definierte Variablen liegen **"automatisch"** in einem besonderen Speicherbereich, **dem Stapel (stack, Keller)**. Der Inhalt dieser Variablen ist zufällig, sofern sie nicht bei der Deklaration initialisiert werden.

Globale, also außerhalb von main() definierte Variablen liegen in einem besonderen **statischen** Speicherbereich und werden mit dem Anfangswert Null belegt.

Oft weiß der Programmierer nicht, wie groß man die Felder dimensionieren soll, da dies vom Anwendungsfall abhängen kann.

In C / C++ gibt es einige Funktionen bzw. einen Operator new, womit man sich nach dem Programmstart dynamisch Speicherplatz reservieren kann.

Der Zusatzspeicher Heap (Halde oder Haufen) umfasst den Teil des **dynamischen Arbeitsspeichers,** welcher nicht vom Betriebssystem oder vom laufenden Programm belegt wird.

Die Vergabe von solchem **dynamischen Speicher** erfolgt durch das Betriebssystem über die in <stdio.h> vordefinierten Funktionen **malloc , calloc und free, bzw. in C++ über die Operatoren new und delete.**

In C kann mit malloc und calloc auf dem Heap ein zusammenhängender Speicherbereich angefordert werden, der durch die Funktion free wieder freigegeben werden kann.

Bei der Vereinbarung lassen sich Zeiger wie Datenspeicherstellen mit Anfangswerten besetzen; entweder mit Adressen bereits definierter Bezeichner (Adressoperator &) oder mit dynamischen Adressen (Operator new) oder mit bereits definierten anderen Zeigern.

Die Funktion malloc übergibt die dynamischen Speicherstellen mit zufälligen Inhalten, die Funktion calloc mit 0 vorbesetzt.

Beide Funktionen liefern eine Zeiger auf den Beginn des neuen dynamischen Speicherbereichs zurück, mit dem Zeigerdatentyp void. Über diesen Zeiger kann dann auf die Daten im dynamischen Speicherbereich zugegriffen werden.

Zeigerbezeichner = (Datentyp *) malloc(Anzahl_der_Bytes) ;
Zeigerbezeichner = (Datentyp *) **malloc (Anzahl * Datenlänge)** ;
Zeigerbezeichner = (Datentyp *) **calloc (Anzahl , Datenlänge)**;

free (Zeigerbezeichner); ==>>> gibt den durch Zeigerbezeichner referierten dynamischen Speicher wieder frei.

Für die Ermittlung der Datenlänge kann der Operator **sizeof(Datentyp)** verwendet werden.

Falls nicht genügend Speicher im Heap vorhanden ist, findet durch malloc oder calloc keine Speicherreservierung statt und die Funktionen liefern NULL zurück.

Ein Beispiel mit malloc :

```
#include  <iostream.h>
#include  <stdio.h>                        // für malloc
main()
{   double * dzeig ;
    dzeig = (double *) malloc(800) ;     // Platzreservieung für 100 double - Werte
    if  (dzeig  ==  NULL)
         {  cout << "malloc-Fehler";
            return;
          }
   ..............
}
```

Und noch ein Beispiel , mit calloc :

```
#include <iostream.h>
#include <stdlib.h>       //  für calloc
main()
{    int  * izeg,  i,  n=4;

    izeig  =  (int *) calloc(n, sizeof(int)  );   // Platzreservierung für  n=4  int-Werte.
    for (i=1; i<=n;  i++)
       {
          *izeig  =  i ;                  //  Wertzuweisung  an Daten
          cout  << "\n"  << izeig ;   //  Kontrollausgabe
          izeig++;                       //   izeig zeigt nun auf den nächsten int - Wert.
       }
  cin.get();
   return 0;
}
```

Mehrere aufeinanderfolgende Operationen mit malloc, calloc liefern getrennt liegende Speicherbereiche, welche nicht zusammen mit der Zeigerarithmetik angesprochen werden können.

In C++ liefert der Operator new einen dem Datentyp entsprechenden Speicherbereich auf dem Heap :

```
Zeigerbezeichner  =  new Datentyp;   //  Speicherbereich für einen Datentyp-Wert
Zeigerbezeichner  =  new Datentyp[20 ];    // Speicherbereich für 20 Werte.
```

// Zu beachten sind hier die eckigen Klammern !!

```
delete Zeigerbezeichner;     /* Von Zeigerbezeichner referierter Bereich wird   freigegeben.   */
```

Wir vergleichen :

```
char * str  =  malloc(20) ;
char * str  =  new char[20];
```

malloc ist eine Funktion, new ist ein Operator ! Die Funktion malloc bekommt als Parameter die Anzahl der zuzuweisenden Zeichen, der Operator new hingegen erhält als Operanden einen echten Datentyp.

Dies kann irgendein gültiger Datentyp sein, auch eine Struktur :

```
struct Person  { char * name ;  int alter;  };

Person * p  = (Person *) malloc(sizeof(Person) );          //   C  _  Stil
Person * p  =  new Person ;                                      //   C++ Stil
```

Im obigen Fall weist new eine Struktur vom Typ Person zu und gibt einen Zeigerwert darauf zurück. Im Gegensatz zu malloc benötigt new keine Typumwandlung.

Der Zeigertyp, welcher von new zurückgegeben wird, ist immer void* und damit kompatibel mit jedem beliebigen Zeigertyp.

Das Arbeiten mit dynamischen Speicherbereichen ist **in C++ mit weniger Typumwandlungen** verbunden : der Compiler übernimmt hier einem Großteil der Arbeit.
Zur Freigabe eines Speicherbereichs kann in C die Funktion free, in C++ der Operator delete verwendet werden :

```
free( str);              //  C - Stil
delete  p;               //  C++ - Stil.
```

Die folgenden zwei Beispiele zeigen spezielle Verwendungen von dynamischem Speicher für zweidimensionale Felder.

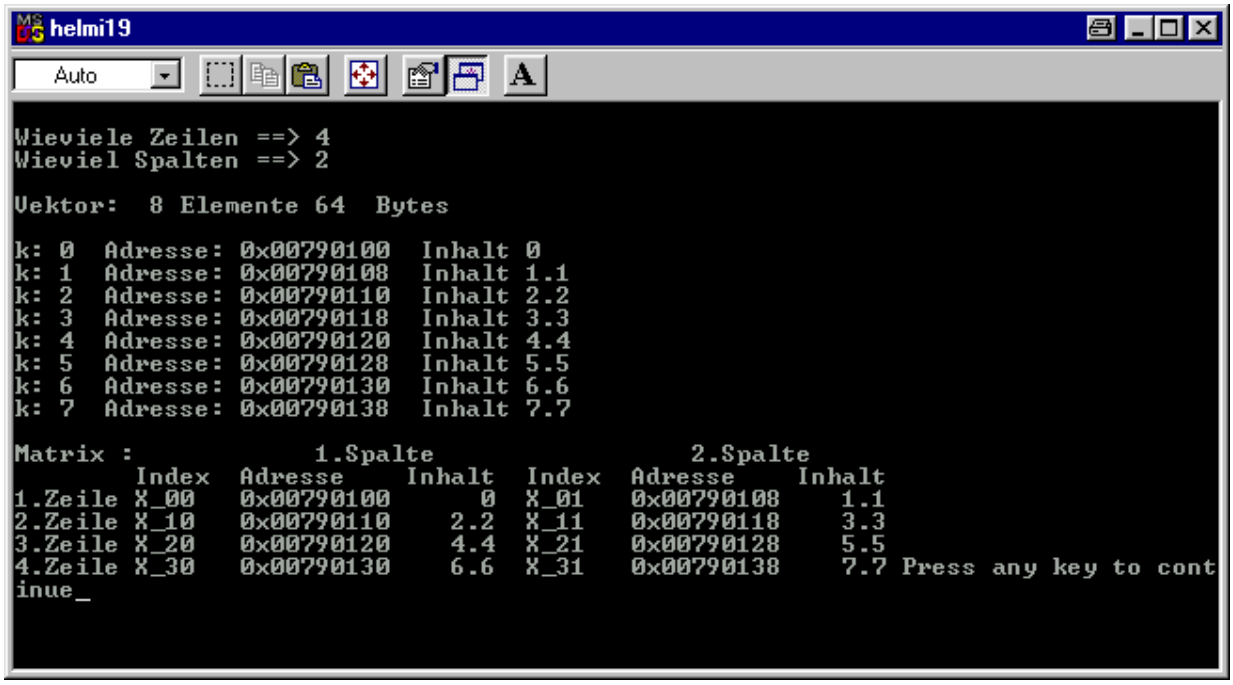
Beispiel 1 für Zeilen- und Spaltenadressierung im dynamischen Speicher

In diesem ersten Beispiel wird zunächst ein zusammenhängender dynamischer Speicher für die gesamte Anzahl der Elemente eines zweidimensionalen Feldes angelegt.
Die Ausgabe der Adressen dieser einzelnen Feldelemente zusammen mit ihrem Inhalt geschieht zunächst in der aufsteigenden Reihenfolge der Adressen, anschließend gruppiert nach einzelne Feldzeilen.

Dabei werden die einzelnen Feldelemente unter Verwendung der Zeigerarithmetik referiiert.

```
#include <iostream.h>              // System I/O-Funktionen
#include <iomanip.h>
#include <stdlib.h>                // für calloc

main()
{
 int  i, j, nz, ns, k;       // Zähler und variable Feldgrößen
 double * xz,  * add;        // Zeiger  xz und add

 cout <<"\nWieviele Zeilen ==> ";  cin >> nz;   // Gewünschte  Zeilenzahl  nz einlesen
 cout <<"Wieviel Spalten ==> "; cin >> ns;      //  Gewünschte Spaltenzahl ns einlesen
 cin.ignore(80,10);
 xz = (double *) malloc(nz*ns*sizeof(double));    //   nz * ns * 8  Bytes
                                                  //  Lineare Adressierung von 0 bis  nz*ns
                                                  //   in C++ :     xz  = new double[nz*ns] ;

 cout  <<"ßnVektor:  "  <<nz*ns  <<" Elemente "  <<nz*ns*sizeof(*xz)   <<"  Bytes";

       // ----------   Ausgabe : einzelne Elemente zeileweise  ------------------------
 for(k=0; k < nz*ns; k++)
    *(xz+k) =  1.1*k,                                                    // oder  xz[k]  = 1.1*k,  in C
      cout <<"\nk: " <<k  <<"  Adresse: "<< (xz+k) <<"  Inhalt " <<xz[k] ;

   // ------------   Ausgabe als Matrix mit Zeilen- und Spaltenadressrechnung  ------------
 cout <<"\n\nMatrix :  ";
 for (j=0; j<ns; j++) cout <<"          "  << j+1 <<".Spalte       ";
 cout <<"\n      ";
 for (j=0; j<ns; j++) cout <<"  Index  Adresse  Inhalt ";

 for (i=0; i<nz; i++)                        // ********** Zeilenschleife für Zeiger
  {
   cout <<"\n"  <<i+1  <<".Zeile";

   for(j=0; j<ns; j++)                       // **********  Spaltenschleife für Zeiger
         {
          add  = (xz + i*ns + j );
          cout <<" X_" <<i <<j <<"   " <<add << setw(7) << *add  <<" ";
         }
  }

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

Hier das Ausgabefenster zum Beispiel 1 :



Beispiel 2 für Zeilen- und Spaltenadressierung im dynamischen Speicher

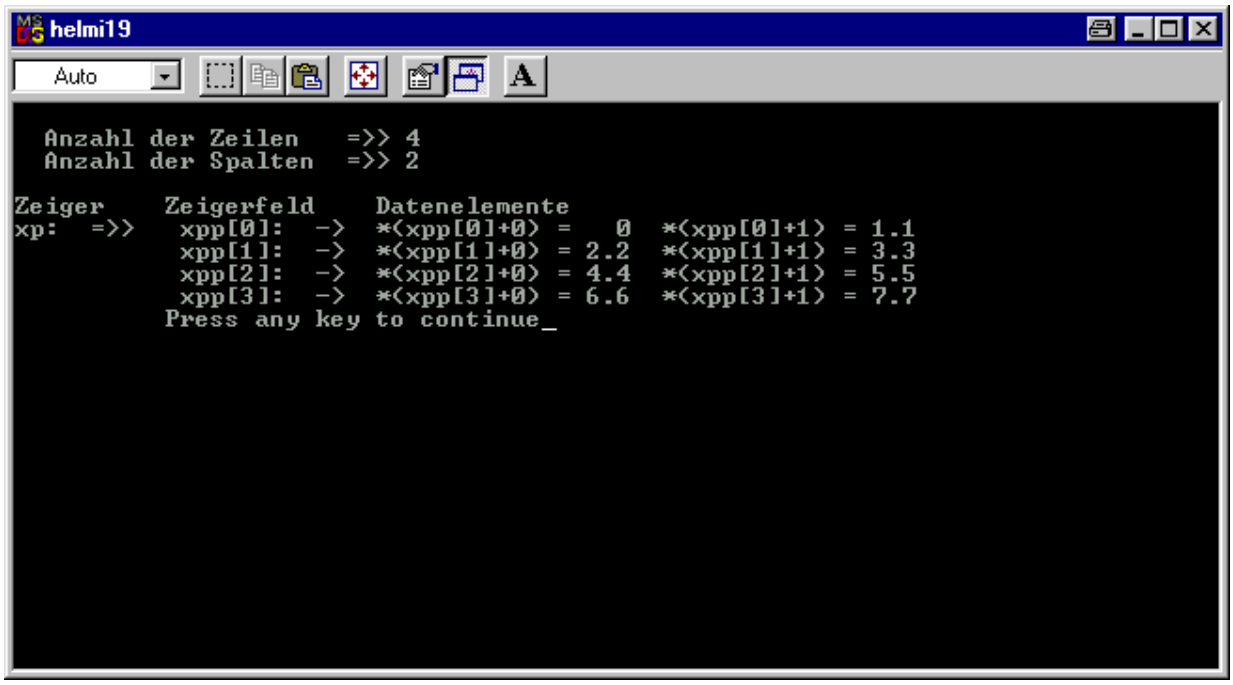
In diesem Beispiel wird im dynamischen Speicher zunächst ein Adressvektor, addressiert von xpp, angelegt.
Jedes Element dieses Adressvektors erhält anschließend die Adresse eines dynamischen Feldes aus ns Speicherstellen. Daraufhin erhält jedes dynamische Feldelement zeilenweise einen Speicherwert.

Schließlich werden die Elementinhalte zeilenweise ausgegeben, mit Angabe der jeweiligen Index Referenz.

Im Programm wird auch hier die Zeigerarithmetik verwendet.

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


main()
{  int i,  j,  k=0,  nz,  ns;    // Zähler und variable Feldgrößen
   double **xpp;                 // Zeiger auf Feld von Zeigern

   cout <<"\n  Anzahl der Zeilen  =>> "; cin >> nz;      //  Gewünschte Zeilenzahö nz
   cout <<"Anzahl der Spalten   =>> ";  cin >> ns;       //   Gewünschte Spaltenzahl ns

   // -----------------  Nun wird ein Feld aus nz Zeigern aufgebaut, für jede Zeile ein Zeiger.-----------
   xpp = (double**) calloc(nz, sizeof(double*));
              // in C++ :    xpp  = new double*[nz] ;

   //  -------------------  Für jeden Zeilenzeiger werden ns Speicherstellen angefordert. ------------------
   for(i=0; i<nz; i++)  xpp[i] = (double*) calloc(ns, sizeof(double));
                 // In C++ :          xpp[i] = new double[ns];

   //  -------------------  Jedes Speicherelement wird aufgefüllt :  ------------------------------
   for(i=0; i < nz;i++)      // Für alle Spalten
     for(j=0; j < ns; j++)  // Für alle Zeilen
              *(xpp[i]+j ) = 1.1*k++;    // Laufende Werte auffüllen
```

```
  //  -------------------  Ausgabe : Feld aus Zeiger auf Zeiger, Zeigerfeld und Daten  ---------
  cout <<"\nZeiger    Zeigerfeld  Datenelemente\nxp:  =>>  ";
  for (i=0; i< nz; i++)
   {
     cout <<" xpp[" << i <<"]:  ->";
     for(j=0; j <ns; j++) cout <<" *(xpp[" << i <<"]+"  <<j
                                    <<")="  << setw(4)  <<*(xpp[i] + j);
     cout <<"\n          ";
   }

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



3.6 Zeiger auf Strukturen

Strukturen lassen sich auch über Zeiger adressieren. Dabei verkürzt **der Operator ->** die Schreibweise auf die adressierten Daten.

```
struct  werte  {    int  std,  min;
                    char einheit[10];
                    double  faktor;
                } ;
werte   * wertez;      //   wertez ist ein Zeiger auf den Strukturtyp werte

(*wertez).std  =  6;          // wie wertez  -> std  = 6;
wertez  -> std  = 6;          //  wie (* wertez).std  = 6;
```

3.7 Allgemeine verkettete Listen

Bei einer **einfach verketteten Liste** besteht ein Listenelement aus einer Struktur mit einem Datenteil und einem Zeigerteil :

```
struct  listenelement  {
                         int daten;                        // Beispiel f ür Datenteil
                         struct listenelement  *  nachfolger;   // Beispiel f ür Zeigerteil
                        };
```

Der Datenteil enthält im allgemeinen Meßwerte oder die Daten eines Bauteils, der Zeigerteil enthält einen Zeiger auf das nachfolgende Listenelement.
In einer struct - Deklaration erlaubt die Programmiersprache C (auch C++ !) ein struct-Element zu deklarieren, welches einen Zeiger auf eben diesen struct-Datentyp darstellt. Damit werden neue Datentypen möglich, welche auf Daten vom eigenen Typ zeigen.

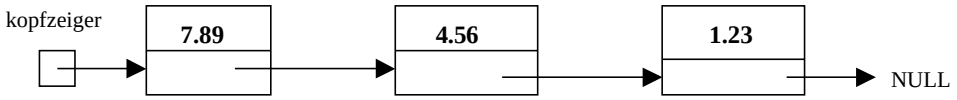
Solche verketteten Listen werden mit malloc, calloc bezw. mit dem Operator new (C++) im Heap angelegt und mit Zeigervariablen verwaltet.

Das folgende Beispiel baut eine verkettete Liste aus einzulesenden Meßwerten auf.

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

void main()                                                  // ----------------  Hauptprogramm ------------------------------------
{  struct  Ltyp  {
                    double daten;                    //    Meßwert
                    struct Ltyp * nachfolger;   // Zeiger auf Nachfolger
                  } ;
    Ltyp  * kopfzeiger,         // kopfzeiger zeigt auf  erstes Element der Liste
          * endezeiger,         // endezeiger zeigt auf das letzte Element der Liste
          * neuzeiger;          //  neuzeiger zeigt auf das jeweils neue Element.

    kopfzeiger  = neuzeiger  = endezeiger  = NULL;
    double wert;               // Einzulesender Wert

     cout << "\nWerte eingeben,  Ende mit  ^Z\n";                    //----------  Einlesen der Werte --------

     while (cout << " -> ", cin >> wert,  !cin.eof()  )
         {
           endezeiger  = neuzeiger ;
           neuzeiger    = (struct Ltyp *) calloc(1, sizeof(struct Ltyp ) );       // in C++ : neuzeiger = new   ltyp;

           neuzeiger -> daten  =  wert;    neuzeiger -> nachfolger  = NULL;    //  Dies wird der neue Listenbeginn
           neuzeiger ->  nachfolger = kopfzeiger ;

           kopfzeiger -> neuzeiger;                                          // kopfzeiger zeigt immer auf den Listenbeginn
         }
     cin.clear();                          // eof-Markierung zurücksetzen

    neuzeiger = kopfzeiger ;              //  --------Vorbereitung zur Ausgabe der einzelnen Listenelemenete    -----------

    cout <<"\n\n Liste der Elemente : ! ";

    if ( neuzeiger == 0 ) cout <<"\n\n  Leere Lite  ";
    while (neuzeiger != NULL )
          { cout <<"   " <<neuzeiger->daten;
            neuzeiger = neuzeiger->nachfolger;
          }
   return ;
 }
```

Beim Start dieses Programms ist die Liste leer, alle Zeiger werden auf NULL gesetzt.
Im allerersten generierten Element erhält das Zeigerelement 'nachfolger' den Wert Null.
In den weiteren Elementen erhält 'nachfolger' den Adresswert des Listenbeginns.
Der Zeiger kopfzeiger zeigt immer auf das neu generierte Listenelement.

Werden bei der Eingabe für dieses Programm nacheinander die Zahlenwerte 1.23, 4.56 und 7.89 nacheinander eingegeben, so erhalten wir den folgenden Zusammenhang :



Die Reihenfolge der Ausgabe ist bei diesem Beispiel umgekehrt zur Reihenfolge der Eingabe .
Dieses Programm eignet sich also **Datensammlung in einem Stack (Last In, First Out) .**

Das folgende Beispiel kann für die **Behandlung einer Warteschlange dienen (First In, First Out) .**

Dieses Programm erzeugt im dynamischen Speicherbereich eine verkettete Liste,

- wobei neue Elemente immer am Listenende angehängt werden,
- und alte Elemente am Listenbeginn weggenommen werden können.
- Auch kann der gesammte Listeninhalt kann jederzeit aufgelistet werden.

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

void main()
{
 struct styp {  double daten;         //  Struktur für verkettete Elemente
                     styp * nach;
             };
  styp * kopfz,  * neuz,  *  letztz,  *  laufz;
  char ant ;                  //  Nur ein Zeichen für Eingabekategorie

  kopfz = neuz = letztz = NULL;  // Initialisierung der Zeigerwerte

  cout <<"\nA = Anhängen,  W = Wegnehmen, L = Liste,  E = Ende  eingeben \n" ;
  do
  {  cout <<"\nBitte Eingabe :   A | W | L | E -> ";
    switch( ant = getche()  )
    {      case 'a':                          //-----------------   Anhängen neuer Listenelemente   ----------------------
           case 'A': letztz = neuz;               // Zeiger auf das Listenende
                     neuz = new styp;             // neues Listenelement im dyn. Speicher.
                     neuz->nach = NULL;           // Neues Element wird Listenende
                     cout <<" Anhängen -> "; cin >> neuz -> daten;   // Daten für das neue Element
                     if ( kopfz == NULL )
                          kopfz = neuz;           // Neues Element als Listenbeginn,
                     else letztz-> nach = neuz;    // oder als Anschluss and das Listenende
                     break;

        case 'w':                          //------------------   Wegnehmen eines ersten Listenlements  ----------------
        case 'W':  if (kopfz == NULL ) cout <<"\n Liste Leer ";
                   else { cout <<"\n\n Erstes Element der Liste : " <<kopfz->daten <<"  weggenommen\n";
                          laufz = kopfz;
                          kopfz = kopfz->nach;
                        }
                   delete laufz;
                   break;
```

```
      case 'l':                          //  ----------------------  Ausgabe der gesamten verketteten Liste  ------------------------
      case 'L':  laufz = kopfz;                          // Listenbeginn
                  if (laufz == NULL) cout <<"\n\n Liste ist leer ";
                  else
                      while ( laufz != NULL) { cout <<"\n   " <<setw(8)  <<laufz-> daten;
                                               laufz = laufz -> nach;
                                             }
                  cout <<endl;
                  break;
    }
  } while ( ant != 'e' );
  return;
}
```

Das nun folgende Beispiel behandelt ein **alphabetisch sortiertes Einfügen von Namen** in eine verkettete Liste

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

void main()
{
  struct Namen  { char namen[30];          // Element der verketteten Liste
                  Namen * next;
                } ;
  char text[30];
  Namen * start, * neuz, * laufz, *hilfe;

  start = neuz = NULL;
  cout <<"\n Bitte, Bezeichnungen eingeben, Ende mit ^Z\n ";
  while ( cout <<" -> ", cin.getline( text ,80,'\n'), !cin.eof() )
  {
    neuz = new Namen;
    strcpy( neuz->namen , text );
    neuz->next = NULL;                          // Vorsorgliche Initialisierung.
    if  (start == NULL)
      start = neuz;            //  Erstes Listenelement
    else
      {                                                 // Liste enthält nun mindestens ein Element
          laufz = hilfe = start;

          while ( ( laufz != NULL ) && ( strcmp(laufz->namen,text) < 0  ) ) // Suche nach nächst höherem Element
              hilfe = laufz, laufz = laufz->next;

          if (laufz == start)                        // Neues Element kommt an den Listenbeginn
              { neuz->next = start;
                start = neuz;
              }
          else   if ( laufz == NULL )                // jetzt zeigt der Zeiger hilfe auf das letzte Element
                     hilfe -> next = neuz;

                 else                                // jetzt müssen wir innerhalb der Liste einfügen
                    {  neuz->next = hilfe;
                       hilfe->next = neuz;
                     }
       }
  }
```

```
                        // -------------------  Ausgabe ---------------------------------
 cout <<"-------AUSSGAABE" ;
 laufz = start;
 if ( laufz == NULL ) cout <<"\n\n keine Namen in der Liste  ";

 while  ( laufz != NULL)  {  cout <<"\n  " << setw (15)  <<laufz->namen ;
                                laufz  = laufz->next;
                             }
 cout <<endl;


}
```

Und nun noch ein letztes Beispiel für verkettete Listen : **Hier kann man beliebige Listenelemente wieder entfernen.** Ein praktische Anwendung wäre, falsche Werte einer Messreihe zu löschen.

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

void main()
{
 struct Messwert {
                    double daten ;
                     Messwert * nach;          // Vorwärts verkettet
                     Messwert * vorg;          // Rückwärts verkettet

                  } *kopfz,  * neuz,  * altz,  * laufz;

 char antwort = 'e';       //  Eingabezeichen
 double wert;              //  zu löschender Wert
 int gefunden;             //  Markierung für gefunden

 kopfz =neuz = altz = NULL;                // Initialisierung

 cout <<"\n M = neuer Messwert  D = Löschen   L = Auflisten  E = Ende  eingeben \n";

 do
 {       // --------  Beginn der großen Schleife -------------
   cout <<"\n Eingabe -> ";
   switch( antwort = cin.get() )
   {
    case 'm':       //  --------------------  Eingabe neuer Messwerte ------------------------------------------
    case 'M':   altz = neuz;                              // altz zeigt immer auf das Listenende
                neuz = new  Messwert;          // ----Neues Listenelement -----
                cout <<"\n\n Neuen Messwert eingeben : -> ";
                cin >> neuz-> daten; cin.ignore(80,10);
                neuz->nach = NULL;
                neuz->vorg = NULL;

                if ( kopfz == NULL)  kopfz = neuz;  // Beim ersten Element
                else { altz->nach = neuz;           // ... später !
                       neuz->vorg = altz;
                     }
               break;

     case 'l':                //  --------------------  Auflisten der Messreihe -----------------------------
    case 'L':   cout <<"\n\n Liste :   \n";       //---- Auflisten aller Elemente
                laufz = kopfz;
                if (laufz == NULL ) cout <<"Liste ist leer ";
                while (laufz != NULL )          //  Ausgabe der einzelnen Elemente
                   { cout <<"\n " <<laufz -> daten ;  laufz = laufz->nach;  }
                cout <<"\n\n";
                break;
```

```
    case 'd':               //  ---------------------  Löschen eines Messwertes in der verketteten Liste von Werten. ----------------
    case 'D':   cout <<"\n\n Zu löschenden Wert eingeben : ->"; cin >> wert; cin.ignore(80,10);
                laufz = kopfz; gefunden = 0;

                while (laufz != NULL )
                   { if ( laufz->daten == wert)
                        {
                          if  ( laufz->vorg == NULL )
                             { if ( laufz->nach == NULL )  kopfz = NULL;      // Einziges Element
                               else { kopfz = laufz->nach;                    // Erstes Element in Liste
                                      kopfz->vorg = NULL;
                                    }
                             }

                          else  if (laufz->nach == NULL )                     // Letztes Element in Liste
                                   {  (laufz->vorg)->nach = NULL ;
                                      neuz = laufz->vorg;                     // Für erneutes Anhängen
                                    }
                               else  {                                        // Element innerhalb der Liste
                                       laufz ->nach ->vorg = laufz->vorg;
                                       laufz ->vorg ->nach = laufz->nach;
                                     }

                          delete laufz;                                       // Dynamischen Speicher freigeben
                          break;
                                    }

                     else {laufz = laufz->nach ;        // Weiter zum nächsten Element

                    }
              }
        }
  } while ( antwort != 'e' && antwort != 'E' );

  return;
}
```

4.) Funktionen

4.1 Funktionsdefinition; Werteparameter,Ergebnisparameter,Referenzparameter

Bei größeren Programmen ist es zweckmäßig, Teilprobleme auf Unterprogramme zu verlagern. Der Grund kann darin liegen, daß die gleiche Verarbeitung sich an verschiedenen Programmstellen wiederholt : an jeder dieser Stellen genügt dann ein entsprechender Unterprogrammaufruf.

Ein weiterer Grund kann sein, daß durch die Zusammenfassung eines bestimmten Programmteils in einem Unterprogramm das Hauptprogramm (main()) wesentlich kürzer und damit übersichtlicher wird.

In C ist ein solches Unterprogramm **eine Funktion**, andere Sprachen (wie z.B. Pascal) unterscheiden zwischen Funktionen und Prozeduren.

Ganz allgemein besteht ein C-Programm aus globalen Deklarationen und Funktionen, wobei zumindest die Funktion main() vorhanden sein muß: jedes C-Programm beginnt mit dieser Funktion main() .

Soweit es die Übersicht nahelegt, sollten größere Funktionen in kleinere Funktionen zerstückelt werden. Diese Maßnahme erleichtert gegebenenfalls die Beschränkung auf nur eine return Anweisung innerhalb einer Funktion.

Man unterscheidet **interne Funktionen,** welche zusammen mit der Hauptfunktion main() in einer Programmdatei stehen und die zusammen mit main() übersetzt werden, und externe Funktionen, die in einer besonderen Programmdatei enthalten sind und separat übersetzt werden.

Die Verbindung der Hauptfunktion main() mit den aufgerufenen Funktionen übernimmt der Linker; darüber später mehr.

Im folgenden befassen wir uns zunächst nur mit internen Funktionen. Wir unterscheiden dabei in drei Unterabschnitten mit den verschieden Kategorien von Parametern.

4.1.1 Funktionen mit Werteparametern

Das nachfolgende einführende Beispiel zeigt eine solche **benutzerdefinierte Funktion,** welche vom Hauptprogramm aus mehrmals aufgerufen wird.

```
#include  <iostream.h>
int summe( int x,  int y)    // Funktion summe liefert Zahlenwert vom Typ int.
{   int z;
    z  =  x + y;
    return z;
}
main()
{   int a= 5,  b = 6,  c;
   c  =  summe(a,b);
   cout << "\nSumme von "  << a  <<'+'  << b   << " ist : " << c ;
   c  = summe(12 , 18);
   cout << "\nSumme von 12+18 ist : "    <<  c;
   return 0;
}
```

In diesem Beispiel wird die Funktion summe zweimal in main() aufgerufen, zunächst mit den Parameterwerten a und b, dann mit den Zahlenkonstanten 12 und 18.

Die Funktionsdeklaration selbst beginnt mit dem Datentyp für den Rückgabewert (int), gefolgt vom Funktionsname (summe) und der Paramerterliste. Der Funktionsname wird nach den üblichen Regeln für Variablennamen gebildet.

Die Funktion summe übernimmt zwei int-Werte als Parameter. Die Parameterliste in der Funktionsdeklaration wird auch als **formale Parameterliste** bezeichnet, entsprechend sind hier x und y die **formalen Parameter** der Funktion, in der Parameterliste durch Kommata getrennt, wobei den Parameternamen noch die Typangaben vorangestellt werden.

Die im Beispiel beim Aufruf verwendeten Parameter a und b, bzw. 12 und 18 bezeichnet man als **aktuelle Parameter.** Die aktuellen Parameter müssen immer in der Anzahl, der Reihenfolge und im Typ den formalen Parametern entsprechen.

Die formalen Parameternamen sind sozusagen **Platzhalter** für die aktuellen Parameter.

Bei der obigen Form von Parameterübergabe spricht man auch von <u>Werteparameter</u>
('call by value ').

Beim Funktionsaufruf kann hier als aktueller Parameter ein beliebiger Ausdruck stehen : der Ausdruck wird ausgewertet, **sein Wert wird nötigenfalls in das Datenformat entsprechend dem formalen Parameter umgewandelt und auf dem Stapel abgesetzt.**

Aktuelle ganzzahlige Werte für reelle formale Parameter werden dabei reell übergeben; aktuelle reelle Werte für ganzzahlige formale Parameter werden unter Verlust der Nachkommastellen ganzzahlig gemacht.

Bei jedem Funktionsaufruf werden hier die formalen Parameter durch die aktuellen Parameter in folgender Weise ersetzt:

Auf dem Stapel (stack) werden beim Funktionsaufruf Kopien der aktuellen Parameterwerte gespeichert, zusätzlich zur Rückkehradresse. Diese Kopien dienen dann bei der Funktionsausführung als Werte für die deklarierten formalen Parameter.

Für die formalen Parameter einer Funktion werden also im übersetzten Programm keine permanenten Speicherplätze angeordnet.
Jede Änderung der Parameterwerte innerhalb der Funktion bewirkt lediglich eine Änderung der auf dem Stapel gespeicherten Kopien, die Originale beim Funktionsaufruf mit solchen Werteparametern bleiben unverändert.

Nach Ende der Funktionsausführung sind die Kopien nicht mehr vorhanden : bei jedem nachfolgenden Funktionsaufruf wird wieder eine neue Liste von kopierten Parameterwerten im Stapel aufbereitet.

Die Anweisungen der Funktion werden wie bei main() in geschweifte Klammern { } gesetzt.

In den Anweisungen einer Funktion können sowohl vordefinierte als auch benutzerdefinierte Funktionen aufgerufen werden.

Da vor dem Funktionsnamen summe der Typ int angegeben wurde, muß auch der Rückgabewert (in return) von diesem Typ sein. Die return-Anweisung gibt das Funktionsergebnis an das aufrufende Programm zurück. Mit der Rückkehr aus einer Funktion kann genau ein Wert eines Datentyps zurückgegeben werden.

An die return Anweisung darf mit oder ohne Klammer ein allgemeiner Ausdruck angehängt werden.
Eine return Anweisung ohne Rückgabewert gibt einen undefinierten Funktionswert zurück; dies ist bei einem Funktionstyp void sinnvoll.

Innerhalb einer Funktion kann mehrmals eine return-Anweisung vorkommen.
Als guter Programmierstil hat es sich bewährt, innerhalb einer Funktion möglichst nur eine einzige return-Anweisung zu verwenden.

Damit kann auch sichergestellt werden, daß eine Funktion nicht unbeabsichtigt ohne ausdrückliche return - Anweisung beendet wird.

Falls innerhalb einer Funktion keine return-Anweisung vorhanden ist, werden alle Anweisungen im Funktionsblock durchlaufen und danach wird zurückgesprungen, ohne einen Wert zurückzugeben.

Bei der Funktionsdeklaration kann man den Rückgabetyp weglassen : dann ist die Funktion vom Typ int ; dies gilt insbesondere auch für die Funktion main().

Funktionen mit dem Rückgabewert void benötigen keine return - Anweisung, bzw. nur return ohne Wert.

Beim Funktionsaufruf können wir noch unterscheiden :

1. `c = c + 5 *summe(12,18)`
   Hier wird das Funktionsergebnis innerhalb eines Ausdrucks verwendet, oder bei einer Wertzuweisung.

2. `cout << summe(12,18);`
   Hier wird der Rückgabewert der Funktion direkt ausgegeben.

3. `printf("Probetext");`
   Der Funktionsaufruf ist hier wie eine normale Anweisung.
   Hier geht der Rückgabewert verloren : eine solche Funktion dient als normales Unterprogramm.

In C ist die Deklaration einer Variablen immer am Beginn eines Blocks {..} möglich. Die Lebensdauer der Variablen umfaßt den Block, in dem sie definiert wurde.

C++ erlaubt es, die Deklaration einer Variablen als Statement zu betrachten : Variablen können in C++ daher an beliebiger Stelle in einem Block definiert werden. Dies ist insbesondere hilfreich für Hilfsvariablen und Schleifenvariablen. C++ Programme gewinnen dadurch an Übersichtlichkeit.

Die obigen Funktionen summe und main werden vom Compiler zunächst als getrennte Programmeinheiten übersetzt und danach zu einem Programm zusammengebunden.

Die Namen der formalen Parameter x und y sind innerhalb der Funktion summe lokal, ebenso wie die definierte lokale Variable z : alle diese drei Namen sind nur innerhalb der Funktion summe bekannt: sie sind vom **Speichertyp auto (für automatic)** .

Eine Funktion kann auch ganz ohne Parameter ausgeführt werden. Mögliche Beispiele dafür sind die Eingabe von Daten bei Programmbeginn oder die Ausgabe von Fehlermeldungen.

Im folgenden Beispiel wird die Verwendung von Werteparameter in verschiedenen Unterprogrammen gezeigt.

4.1.2 Funktionen mit Ergebnisparametern

Eine Alternative zu den Werteparametern sind die **Ergebnisparameter, auch Variablenparameter ('call by name'), gelegentlich auch Referenzparameter genannt**.

Hier wird beim Funktionsaufruf anstelle einer Variablen bzw. eines Konstantenwertes ein **Zeiger (Adresse) auf die Variable bzw. auf den Konstantenwert** als aktueller Parameter verwendet. Wie bei den Werteparametern gelangt eine Kopie des Zeigerwertes auf den Stapel.

Das Funktionsprogramm kann nun über diese Zeiger - Kopie auf das Original zugreifen und es ändern. Ein solcher Zeigerparameter kann auch den Beginn eines Arrays oder einer Zeichenfolge darstellen. Das nachfolgende Programm zeigt ein Beispiel : Hier werden die Werte zweier Variablen im Unterprogramm vertauscht.

```
#include <stdio.h>
void vertausche (int * a,  int * b)
{ int  c  =  *a ;          //  *a  ist die durch den aktuellen Zeigerwert referierte Variable
```

```
  *a   = *b;
  *b   =  c;              //  Da der Rückgabewert der Funktion als void deklariert ist, ist kein return nötig.
}
main()
{   int   x = 10,  y  = 20 ;
    printf("%d  %d\n",  x,  y);    // Ausgabe der Werte für x und y
    vertausche( &x,  &y);          // Die Adresswerte &x  und &y  sind die aktuellen Parameter
    printf("%d  %d\n", x, y);      // Ausgabe der vertauschten Werte
   return 0;
}
```

Und noch eine Beispiel :

```
#include <iostream.h>
double   sumdiff( double a,  double b,  double * c)
{  *c  =  a  +  b;        //  Summe über  Ergebnisparameter (Variablenparameter : call by name)
    return  a  - b ;      //  Differenz als Funktionsergebnis
}

main()
{   double  *p1 = new double,   *p2 = new double,  *p3,  *p4;
    p3  = new double;   p4  = new double;                    // Auch bei Definition möglich !
    cout   <<"\n1. Zahl reell : ";   cin  >>   *p1;
    cout  <<"\n2. Zahl reell : ";    cin  >>  *p2;
    *p4  =  sumdiff( *p1,  *p2,  p3);              //  p3 ist Ergebnisparameter : Adresse !
    cout  <<  "\n"  <<*p1  <<" +"   << *p2   << " = "   <<  *p3;
    cout  << "\n"   << *p1  << " -"  << *p2   << " =  "  << *p4;
  return 0;
}
```

Alle in einer Parameterliste durch * gekennzeichnete Namen sind Zeiger und sind innerhalb des Funktionsblocks auch als Zeiger zu behandeln.

In einer formalen Parameterliste können Werteparameter und Ergebnisparameter auch gemischt vorkommen : void differenz(int x, int y, int * z)

Eine solche Funktion mag z.B. dazu dienen, die Differenz x - y zu bilden und in *z abzuspeichern.

Beim Aufruf einer Funktion **müssen die aktuellen Parameter in Reihenfolge, Anzahl und Datentyp mit den formalen Parametern übereinstimmen.**

Enthält die formale Parameterliste einen Zeigertyp, so ist als aktueller Parameter entweder ein Zeigersymbol oder ein Arrayname (Zeigerkonstante) oder eine Adressnotation, wie &a , zu spezifizieren.

4.1.3 Funktionen mit Referenzparameter

Als Referenz (reference) wird in C++ ganz allgemein ein alternativer Name für ein Objekt bezeichnet : eine Referenz führt ein Synonym für ein Objekt ein :

```
int    n;

int& nn=n;            // nn ist ein Synonym für n

nn++;                 // n wird um 1 erhöht.
```

Durch einen Referenzparameter bekommt eine Speicherstelle einen neuen Namen !

```
double a[10];
double& last=a[9];        // last ist ein Synonym für a[9]
```

Um einen Zeiger auf das Objekt der Referenz zu richten, kann &nn geschrieben werden.

Eine Referenz muß immer initialisiert werden, sobald sie definiert wird; sie muß für etwas stehen.

Eine Referenz kann auch auf eine Konstante intialisiert werden. In diesem Fall wird eine Kopie der Konstanten erstellt, ggf. nach erforderlicher Typenumwandlung.

```
int& i=1;        // i wird eine Referenz auf eine Kopie von 1, nicht auf 1 selbst.

char&   new_line='\n';   //    erzeugt Referenz zu gespeichertem '\n'
```

In C++ werden Referenzen vornehmlich dazu benützt, um formale Parameter in Funktionsdefinitionen als Variablenparameter sowie um Funktionsresultate zu spezifizieren.

Eine Referenz zeigt wie ein Zeiger auf eine andere Adresse, im Gegensatz zu der veränderlichen Zeigervariable ist sie jedoch immer an diese Adresse gebunden.

Für Referenzen gibt es kein Analogon zu einem NULL-Zeiger. Nach der Initialisierung kann eine Referenz selbst nicht mehr verändert werden.

Ein Referenzparameter für eine Funktion f ist einer, der bei seiner Änderung auch eine Änderung des entsprechenden Arguments in einem Aufruf von f bewirkt. Dies steht im Gegensatz zu Werteparameter.

Solche Referenzparameter werden innerhalb der Funktionsdefinition als Variablen und nicht als Zeiger behandelt.

```
void incr(int& aa)
      { aa++; }            // aa selbst wird nicht verändert, sondern das referierte Objekt.
```

Durch das & hinter der Typangabe int wird hier aa zu einer Referenz auf das Objekt, das beim Aufruf an die Funktion übergeben wird.

Auf Referenzen selbst können keine Operatoren angewendet werden.

Intern wird bei diesem **'call by reference'** als Parameter ein Zeiger auf das Objekt übergeben, ganz so wie bei den oben behandelten Ergebnisparametern mit Zeigern.

Bei jedem Zugriff auf den formalen Parameter dereferenziert der Compiler den Zeiger, wodurch automatisch auf das Objekt selbst zugegriffen wird.

Die entsprechenden aktuellen Parameter sind Variablen ohne besondere Kennzeichnung.

```
void f()  {  int x = 1;
                incr(x);    // x = 2
             }
```

Alternativen zu incr() mit Referenz: eine Funktion kann explizit den Wert zurück liefern, oder Pointer-Argumente erwarten :

```
int next (int p)  { return p+1;}
void  inc(int* p)  { (*p)++;  }
void g()
      { int x = 1;
        x = next(x);       // x=2;
        inc(&x);           //x = 3
       }
```

Unser obiges Vertauschen - Programm können wir nun mit Referenzen einfacher schreiben :

```
#include <stdio.h>
void vertausche (int & a,  int & b)
{ int  c  =  a ;                        //  a   und b referieren hier nun die aktuellen Parameter !
   a  =  b;
   b  =  c;
    // Da der Rückgabewert der Funktion als void deklariert ist, ist kein return nötig.
}
main()
{   int   x = 10,  y  = 20 ;
    printf("%d  %d\n",  x,  y);            // Ausgabe der Werte für x und y
    vertausche( x,  y);                     // Die Werte  x  und  y  sind die aktuellen Parameter
    printf("%d  %d\n", x, y);              // Ausgabe der vertauschten Werte
    return 0;
}
```

In der Zeigerversion ist die Tatsache, daß die Funktion ihr Argument ändert, auf eine praktische Weise in einem Aufruf der Funktion dokumentiert, wohingegen die Referenzversion keine solche Hinweise gibt.

Das nachfolgende Beispiel zeigt die Verwendung von Funktionen bei einer Keller - Anwendung.
Im Beispiel wird eine eingelesene Folge von Zeichen zunächst gekellert, und dann in umgekehrter Reihenfolge wieder ausgegeben.

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

const int  maxlen  = 500;
enum boolean  {FALSE,  TRUE  };
enum  { EMPTY =  -1,  FULL = maxlen - 1  };

struct stack   {  char  zeichen[maxlen];                  // Feld zeichen dient als Keller ( Stack )
                  int  top;                               //  top indiziert die obere Kellermarke
               } ;

                                                                  //  ----------  Funktionsdefinitionen ------------
-----
void  reset(stack *  stkz)   { stkz  ->  top  = EMPTY;    }                  //  reset intialisiert den leeren Stack
void  push(char c, stack * stkz) { stkz  ->top++;                            // push  kellert ein neues Zeichen
                                       stkz -> zeichen[stkz ->top]  = c;
                                     }
char pop(stack* stkz)   {return (stkz  -> zeichen[stkz -> top-- ]  ) ; }        //  pop hölt oberstes Zeichen aus dem Keller

boolean leer(stack * stkz) {return (enum boolean) (stkz -> top  == EMPTY);  }  //  leer prüft auf leeren Keller
boolean voll(stack*  stkz) {return (enum boolean)(stkz ->top  == FULL);    }     //  voll prüft auf vollen Keller ( Überlauf ).
```

```
main()            // ---------------------  Hauptprogramm
{
  stack  keller;
  static char adresse[40]  = { "Anna" } ;            // Die Zeichenfolge "Anna" soll gekellert werden !
  int i = 0;
  cout << adresse  << "\n";                          //  Ausgabe vor dem Kellern

  reset(&keller);
  while (adresse[i] )                                //  adresse kellern auf dem Stack
    if (!voll(&keller))
            push(adresse[i++],  &keller ) ;

  while(!leer(&keller) )                            // rückwärts wieder ausgeben
    cout  << pop(&keller);
  cout << "\n";
  return 0;     // Rücksprung nach System
}
```

4.2 Funktionsprototyp; Deklaration versus Definition von Funktionen

Bei den obigen Beispielen kommen im Quellprogramm die aufzurufenden Funktionen vor der Funktion main(). Statt nur einer aufzurufenden Funktion können es natürlich auch mehrere umfangreichere Funktionsdefinitionen sein, die vor main() stehen. Da der Compiler die Definitionen dieser Funktionen vor ihrem Aufruf liest, sind ihm später die Namen und Parameter der Funktionen bekannt.

Ein C-Programm läßt sich aber auch so organisieren, daß man zunächst die Funktion main() im Quellprogramm voranstellt und danach erst die ausführlichen Funktiondefinitionen platziert.

In diesem Fall muß man zu Beginn des Quellenprogramms, also noch vor main(), sogenannte **Funktionsdeklarationen bzw. Funktionsprototypen** platzieren. Damit macht man dem Compiler gewissermaßen den Funktionsnamen, den Funktionstyp und die formale Parameterliste im voraus bekannt.

Ganz allgemein ist zu unterscheiden zwischen Funktionsdeklaration, Funktionsdefinition, und Funktionsaufruf. Das folgende Programm ist unser früheres Beispiel in der Programm - Organisation mit einem Funktionsprototypen :

```
#include  <iostream.h>

int summe( int x,  int y) ;              // Funktionsprototyp
main()
{  int a= 5,  b = 6,  c;
   c  =  summe(a,b);
   cout << "\nSumme von "  << a  <<'+'  << b   << " ist : " << c ;
   c  = summe(12 , 18);
   cout << "\nSumme von 12+18 ist : " <<  c;
  return 0;
}

int summe( int x,  int y)    // Funktion summe liefert Zahlenwert vom Typ int.
{   int z;
    z  =  x + y;
    return z;
}
```

Dank dem Funktionsprototypen sind die Datentypen der Funktionsparameter dem Compiler gleich zu Beginn der Übersetzung bekannt. Die Übergabe der aktuellen Parameterwerte beim Funktionsaufruf geschieht wie oben beschrieben.

Die im Quellenprogramm vorangehende Formulierung des Prototyps nennt man auch eine **Funktions-Deklaration,** den nach main() nachfolgenden vollständigen Funktionsblock nennt man eine **Funktions-Definition.**

Eine Funktionsdefinition wird immer mit dem Typ der Funktion, dem Funktionsnamen und den Parameternamen einschließlich ihrer Datentypen eingeleitet. In einer Funktionsdeklaration genügen in der Parameterliste auch die Datentypen, ohne Namen der Parameter:

```
int summe(int,  int );
```

Jede solche Deklaration ist mit einem Semikolon (;) abzuschließen.

In der Funktionsdeklaration (Prototyp) müssen die Anzahl und die Datentypen mit den Datentypen in der Funktionsdefinition übereinstimmen, die Variablennamen können in der Funktionsdeklaration verschieden zu den Namen der formalen Parameter in der Funktionsdefinition sein.

Eine besondere Möglichkeit, mehreren Funktionen einen gemeinsamen Wert zugänglich zu machen, sind **globale Variablen,** welche außerhalb aller Funktionen, auch außerhalb von main(), ganz zu Beginn des Quellenprogramms definiert werden. Auf solche Namen können alle nachfolgenden Funktionsprogramme zugreifen, ohne Parameterübergabe.

Der Nachteil ist allerdings, daß solche Programme sehr schwierig zu ändern sind, da bei einer die globale Variable betreffende Programmänderung die Auswirkung der Änderung auf alle nachfolgenden Funktionen sorgfältig untersucht und überprüft werden muß.

Je umfangreicher ein Programm wird, um so wichtiger wird die Vermeidung von allgemeinen globalen Variablen. Dieses Ziel wird ganz wesentlich beim **Objektorientierten Programmieren** angepackt.

4.3 Speicherklassen auto, static, register; Gültigkeitsbereiche für Namen

In früheren Kapiteln wurde schon vereinzelt auf Speicherklassen eingegangen. Hier sollen die verschiedenen Speicherklassen in C zusammenfassend behandelt werden.

1. Lokale Variablen(nicht static), formale Parameter : **Speicherklasse auto**

Variablen, welche innerhalb eines Funktionsblocks oder in einem inneren Block { } deklariert werden, sind automatisch vom Typ **auto.** Sie haben ihre Gültigkeit nur innerhalb des Blocks, in dem sie deklariert wurden.

Die Lebensdauer beschränkt sich ebenfalls auf den Block. Beim Verlassen des Blocks bzw. der Funktion gehen die Werte verloren. Wird die Funktion später noch einmal aufgerufen, wird die Variable auf dem Stack neu angelegt.

Die Werte für solche **lokalen Variablen und Parameter** werden bei Funktionsbeginn auf dem Stapel plaziert, und sind nur während der Funktionsausführung innerhalb der Funktion verfügbar.

Sollte die lokale Variable bei ihrer Deklaration initialisiert werden, erhält die Variable bei jedem neuen Funktionsbeginn auf dem Stapel diesen Anfangswert. Andernfalls ist der Wert zufällig.
Diese Regeln gelten auch für alle inneren Blöcke innerhalb eines Funktionsblock's.

Die Namen für solche lokale Variablen können innerhalb einer Funktion völlig unabhängig von anderen Funktionen gewählt werden. Sind im Programm globale Deklarationen mit gleichem Namen vorhanden, so verdeckt innerhalb der Funktion die lokale Deklaration die globale Deklaration .

Alle innerhalb eines Blocks deklarierten Namen sind entsprechend in allen Unterblöcken { } gültig, sofern nicht neu deklariert.

Zu beachten ist noch, daß in C zu Beginn jedes inneren Blocks neue Deklarationen stehen dürfen, in C++ können neue Deklarationen irgendwo innerhalb eines Blocks stehen. Es fördert gelegentlich die Programmübersicht in C++, wenn solche lokalen Deklarationen zu Blockbeginn stehen. Bei der Anweisung

```
for (int i=0;  i<10; i++)  {  ...   ....          }      //    in C++ erlaubt !
```

wird die i-Variable innerhalb der for-Schleife deklariert, **ihr Gültigkeitsbereich ist der umschließende Block, nicht nur der die for-Schleife bildende innere Block !**

2. Lokale, statische Variablen : **Speicherklasse static**

Bei lokalen statischen Variablen wird die Variable beim ersten Eintritt in die Funktion, in der sie deklariert wurde, in einem besonderen statischen Speicherbereich neu eingerichtet, **- nicht auf dem Stapel !**

Beim Verlassen der Funktion bleibt die Variable mit ihrem Inhalt erhalten, bei einem späteren neuen Funktionsaufruf steht die Variable mit ihrem zuletzt gültigen Wert zur Verfügung. Aber sie kann nicht außerhalb der Funktion angesprochen werden. Eine in einem inneren Block { } deklarierte statische Variable kann nur innerhalb dieses inneren Blocks referiert werden.

Alle Variablen vom Typ static, falls nicht im Programm explizit initialisiert, werden automatisch mit Null vorbesetzt.

Eine explizite Initialisierung einer static-Variable in der Deklaration wird nur beim ersten Funktionsaufruf ausgeführt. Beispiel :

```
double funktion()
     {
       static double  x = 1;
       x++;     // Bei jedem Funktionsaufruf erhält x einen um 1 höheren Wert.
                      ......
       return;
      }
```

Noch ein Beispiel : Ein Programm soll die Summe der Diagonalwerte einer Matrix ausgeben.

```
void main()
{
    int i, summe = 0;

    ar[3][3] =  {    {  10,  20,  30  } ,
                     {  40,  50,  60  } ,
                     {  70,  80,  90  }
                } ;
    for (i=0; i < 3;  i++)
        {    static  int j;
             summe  += ar[i][j++] ;
        }
    printf("\n  %d", summe);
}
```

Hätten wir als Deklaration für j geschrieben : int j = 0, so wäre j bei jedem neuen Blockeintritt neu initialisiert worden, und wir hätten nicht die Summe der Diagonalelemente bekommen, sondern die Summe der ersten Spaltenelemente (mit jeweils j=0) !

3. Register – Variablen : **Speicherklasse register**

In ANSI - C ist es für lokale Variablen innerhalb einer Funktion möglich, sie in einem Register zu halten, indem man solchen Variablen von Typ auto bei der Deklaration den Zusatz register gibt.

```
register int i;
```

Da es im Prozessor nur wenige Register gibt, muß der Compiler diese register-Angabe nicht berücksichtigen, er kann es aber tun. Dies kann beispielsweise dadurch geschehen, daß beim Aufruf einer Funktion die Inhalte der Datenregister zwischengespeichert werden. Nach Verlassen der Funktion können die alten Inhalte wieder in den Registern gespeichert werden.

4. Globale (nicht statische) Variablen

Variablen welche außerhalb aller Funktionen deklariert werden, sind vom **Typ extern** und heißen **global.** Sie können in allen nachfolgenden Funktionen in der Quelldatei benutzt werden. Die Lebensdauer erstreckt sich über den gesamten Programmablauf.

Diese Variablen liegen im Speicher im besonderen statischen Speicherbereich und werden mit dem **Anfangswert Null** belegt, wenn nicht explizit bei der Definition initialisiert.

Solche externen Deklarationen von globalen Variablen können in einem Quellenprogramm irgendwo außerhalb der Funktionsdefinionen stehen, also nicht nur am Anfang.

Bei größeren Programmen ist es allerdings der Übersicht wegen dringend zu empfehlen, globale Variablen beim Programmbeginn zu deklarieren. Globale Variablen werden zu Programmbeginn mit 0 vorbesetzt.

Beispiel :

```
#include <stdio.h>

void funktion1(double);                    //  Deklaration für Funktionsprototyp
double umfang;                             // globale (externe) Variable

void main()
{
   double radius;
   printf("\nGib Wert für radius ein: ");  scanf("%lf",  &radius);
   funktion1(radius);
   printf("\Umfang = %lf", umfang);
  return;
}

void funktion1(double radius)
{
    double pi = 3.14159;
    umfang = 2*pi*radius;       // Hier wird die globale externe Variable   umfang verändert.
    return;
}
```

Beide Funktionen main() und funktion1() sind hier vom Typ void, daher genügt bei beiden am Schluß ein einfaches return.

Die globale Variable umfang kann in allen Funktionen referiert werden.

Der Parameter radius in funktion1 ist nur lokal in dieser Funktion bekannt, ebenso ist die Variable radius in main() lokal und hat mit dem formalen Parameternamen radius in funktion() nur den Namen gemeinsam. Die Namengebung für Variablen und formale Parameter ist in jeder Funktion lokal und unabhängig von gleichen Namen in anderen Funktionen.

Durch den Zugriff auf globale Variablen können zwar die Parameterlisten bei Funktionen entfallen; jedoch geht bei größeren Programmen mit vielen Funktionen die Kontrolle über die Werte von globalen Variablen leicht verloren.

Große Programme mit globalen Variablen sind schwierig zu ändern bzw. zu korrigieren

Werden global vereinbarte Bezeichner in einer Funktion nochmals für lokale Größen verwendet, so übersteuern die lokalen Vereinbarungen innerhalb ihres Gültigkeitsbereichs die globalen.

Wir unterscheiden im folgenden **zwischen dieser Definition einer Variablen, und ihrer Deklaration.**

Eine globale, externe Variable muß genau einmal **definiert** werden, und erhält dabei ihren Speicherplatz zugewiesen. Situationen für zusätzliche Deklarationen der gleichen Variable ergeben sich, wenn

a) die Variablendefinition im Quellprogramm nach einer Funktion steht, in der die globale Variable benutzt wird.

b) die globale Variable in einer separaten Quelldatei benutzt wird.

Für beide Situationen folgt nun je ein Beispiel.

Im ersten Beispiel will man eine globale Variable nur ganz bestimmten Funktionen im Programm bekannt machen. Hier kann man dann die externe Deklaration auf diese Funktionen beschränken und die Definition im Programm später folgen lassen :

```
 main()
 {   extern  float   gvariable;       // externe Deklaration innerhalb von main()
     ....
 }
 float  funktion1(...)
 {  extern  float  gvariable;       // externe Deklaration innerhalb von funktion1
    ...
 }
float funktion2( .. )
{
   .....
}
float gvariable;                  // Definition der globalen Variable gvariable

float funktion3( .. )
{
   .....
}
```

Beim obigen Beispiel ist die globale Variable gvariable in main, funktion1 und in funktion3 bekannt, nicht aber in funktion2.

5. Globale statische Variablen: **Speicherklasse static**

Bekommt eine globale Variable bei der Definition den **Zusatz static,** so steht sie nur in dem Programmodul zur Verfügung, in welchem sie definiert wurde. Sie kann also nicht mit der Angabe extern in anderen Moduln bekannt gemacht werden. Sie existiert während des gesamten Programmablaufs.

Deklariert man in einem anderen Modul eine Variable gleichen Namens, so handelt es sich für den Compiler um verschiedene Variablen mit eigenen Speicherplätzen und auf ihre Moduln beschränkte Gültigkeitsbereiche.

Nur Funktionen in der eigenen Moduldatei können auf solche statischen Variablen und statische Funktionen zugreifen. Durch das Wort static ist der Zugriff zu diesen Namen von anderen Moduln aus gesperrt.

Damit sind nicht alle Funktionen eines Moduls als offizielle Schnittstelle nach außen zugelassen: **'Information Hiding'.**

In C ist es möglich, ein Quellprogramm in mehreren Teilen getrennt in verschiedenen Dateien zu halten. Solche einzelne **Moduln** können dann getrennt compiliert und anschließend zu einem gemeinsamen Programm zusammen - gelinkt werden.

Nun muß man dem Compiler mitteilen, daß evtl. eine in einer Datei definierte globale (externe) Variable auch in einer anderen Datei (Modul) benutzt werden kann.

Dies geschieht in der anderen Datei durch eine **externe Deklaration :**
Quelldatei Progr1.cpp:

```
float xkoordinate;                    // globale,  externe Variable

main()
{    int index = 5;
     float  y;
     y  =  xkoordinate * index;
    ..  ...
}
```

Quelldatei Progr2.cpp :

```
extern  float xkoordinate ;            //   extern Deklaration für xkoordinate

float berechne()
{  ....
    xkoordinate  =  200;
   .....
}
```

In der Datei Progr1.cpp ist die globale Variable xkoordinate definiert.

In der Datei Progr2.cpp steht mittels des Zusatzes extern nur eine Deklaration (keine Definition) für diese Variable. Dieser explizite Zusatz extern dient hier also dazu, dem Compiler und dem Linker solche globalen Variablen in der Datei Progr2.cpp bekannt zu machen.

Für die Variable xkoordinate wird im übersetzten Programm für Progr1.cpp ein Speicherplatz reserviert.

Anstatt der globalen externen Deklaration in Progr2.cpp könnte hier auch innerhalb spezieller Funktionen eine solche externe Deklaration stehen: dann wäre in dieser Quelldatei diese externe Variable eben nur innerhalb dieser Funktionen bekannt gemacht.

Die Speicherklasse extern läßt sich auch auf Funktionsdeklarationen anwenden und hat eine dementsprechende Bedeutung.

Ein Programm hat dann in einer ersten Quelldatei eine Funktionsdefinition, mit oder ohne einen Prototyp, in der anderen Quelldatei steht nur noch die Funktionsprototyp - Deklaration mit vorangehenden extern :

```
extern  int  funkt1(.....);          //  extern ist bei Funktionen optional
```

5. Globale statische Variablen: **Speicherklasse static**

Bekommt eine globale Variable bei der Definition den **Zusatz static,** so steht sie nur in dem Programmodul zur Verfügung, in welchem sie definiert wurde. Sie kann also nicht mit der Angabe extern in anderen Moduln bekannt gemacht werden. Sie existiert während des gesamten Programmablaufs.

Deklariert man in einem anderen Modul eine Variable gleichen Namens, so handelt es sich für den Compiler um verschiedene Variablen mit eigenen Speicherplätzen und auf ihre Moduln beschränkte Gültigkeitsbereiche.

Nur Funktionen in der eigenen Moduldatei können auf solche statischen Variablen und statische Funktionen zugreifen. Durch das Wort static ist der Zugriff zu diesen Namen von anderen Moduln aus gesperrt.

Damit sind nicht alle Funktionen eines Moduls als offizielle Schnittstelle nach außen zugelassen: **'Information Hiding'.**

4.4 Felder, Strukturen und Zeiger als Parameter

Feldelemente werden durch eine Indexposition zwischen eckigen Klammern bezeichnet und haben als formale Parameter die gleiche Wirkung wie einfache Datentypen int oder double.

Beispiel :

```
void quadrat( int a, double * y)
{
    *y   = a  *  a;                  // Wertzuweisung  an  Variablenparameter
}

main()
{  int i,  ix[5];                    // Feld aus 5 int-Werten.
   double  dx[5];                    // Feld aus 5 double - Werten.
   for (i=0;  i<5; i++)
      {     ix[i]  = i;              //  Wertzuweisung
            quadrat(ix[i],  &dx[i] );    //Aufruf mit Feldelementen.
       }
}
```

Ganze Felder werden nicht als Parameter an eine Funktion übergeben, sondern nur der Zeiger auf das erste Element. Der Feldname ist ein Adresszeiger auf das erste Feldelement.

Bei der Funktionsdefinition werden die entsprechenden formalen Parameter mit leeren eckigen Klammern als Felder bezeichnet.

Ein Beispiel :

```
#include  <iostream.h>
#include  <stdlib.h>                          // für randomize und random
#define  N  10                               //  Anzahl der Feldelemente

void zufall (int x[],  int n)            // Parameter x[] ist Zeiger auf ein Feld   (Feldname)
                                         // der zweite Parameter kennzeichnet die Feldgröße.
   {    int i;                               // i ist lokal in zufall()
        for  (i=0;  i < n;  i++)   x[i]  =  random(n) + 1;     //  n int-Zufallszahlen von  1..n
   }

main()
{  int  wert[N],  i;                  //  i  ist   lokal in main()
   randomize();
   zufall(wert,  N);                  //   Parameter : Feldadresse und Anzahl der Feldelemente
   cout  <<"\nZufallszahlen im Feld :  ";
   for  (i=0; i < N; i++)  cout  << "   "  << wert[i];
   return 0;
}
```

In der obigen Parameterliste für die Funktion zufall hätte als erster formaler Parameter auch int x[N] stehen können, jedoch hätte hier ein Wert innerhalb der eckigen Klammer keine Bedeutung für das übersetzte Programm. Übergeben wird hier wirklich ein Zeiger auf ein Feld.

Der zweite Parameter (int n) dient dabei zur Spezifizierung der Feldgröße. Bei Zeichenketten als Parameter ist das Ende durch '\0' in der Zeichenkette erkennbar.

Als Parameter bei der Definition einer Funktion **sind char s[] und char * s äquivalent.**

Wird ein Vektorname an eine Funktion übergeben, so kann die Funktion je nach Belieben annehmen, daß ein Vektor oder ein Zeiger übergeben wurde und den Parameter entsprechend verwenden.

Anstelle von Zeigern erster Stufe (int * z1) für eindimensionale Felder (Vektoren) kann man also auch Felder unbestimmter Länge angeben (int z1[]).

Das folgende Beispiel erlaubt einen Vergleich zweier Funktionen, die mit Arrays arbeiten :

```
#include  <stdio.h>

void iausgabe( int a[],  int anzahl)    // Hier  benötigen wir auch die Anzahl als Parameter
{  int i;
   for (i=0; i< anzahl; ++i;)   printf("%d ",  a[i]  ) ;
   putchar('\n');
}

void strausgabe(char s[])          // bei Array mit Zeichenkette kennzeichnet '\0' das Ende !
{ int i;
  for  ( i=0;  s[i]  !=  '\0';  ++i)  putchar(  s[i] ) ;
  putchar('\n');
}

main()
{  char  zeichenk[81] ;
    int  ia[] =  {  1,  2,  3,  4,  5  };
    printf("\nBitte, einen Satz eingeben :  ");      gets(zeichenk);
    strausgabe(zeichenk);                                  // Feldname als Parameter genügt hier.
```

```
    iausgabe(ia,  sizeof(ia) / sizeof(int)  ) ;    //  sizeof(ia) ist die Feldlänge in Bytes, sizeof(int)
                            //die Bytes/int .  Der  zweite aktuelle Parameter spezifiziert hier die Anzahl !
}
```

Vollständige Strukturen können als Parameter einer Funktion oder auch als Funktionsergebnis verwendet werden. Dazu muß allerdings zunächst der Strukturtyp im Programm deklariert werden.

Ein Beispiel soll diese Zusammenhänge verdeutlichen :

```
#include <iostream.h>

struct styp  {   int nr;
                    double wert;
              };
                                          // Struktur als Funktionsergebnis  :
styp anfang (int ganz, double reell)   // Deklaration in C++
{    styp hilfe;                          //  Hilfsstruktur
     hilfe.nr  = ganz;
     hilfe.wert  = reell;
     return hilfe ;                       //  Rückgabe über Hilfsstruktur
}

                                        //  Struktur als Parameter,   Variablenparameter wie Zeiger
void  kopiere (styp quelle,  styp *  zielp)
{  zielp  ->  nr  = quelle.nr;              //  äquivalent zu (*zielp).nr
   (*zielp).wert = quelle.wert;          //  äquivalent zu zielp  ->  wert
}
main()
{   styp  x, y;
     x  = anfang(14,  27.42);            //  belegt Struktur x;
    cout  <<"\nx-Struktur : Nr: " <<  x.nr  << "  Wert:  " << x.wert;
    kopiere(x,  &y);                      //  kopiert x  nach  y
    cout  <<"\ny-Struktur : Nr: "  << y.nr  <<"  Wert:  "  << y.wert;
   return 0;
}
```

```
/*  Beispiel für verschieden Funktionsaufrufe
 *
 */

#include <iostream.h>
#include <iomanip.h>

#define N 20

   //--------   F u n k t i o n s p r o t o t y p e n ----------------

void einlesen(int feld[], int& index, int maxind);
void bubble( int feld[], int anzahl);
void abfragen(int feld[], int anzahl  );
int  binsearch( int feld[], int anzahl, int suchwert);
void ausgabe( int feld[], int anzahl );

   //------------ H a u p t p r o g r a m m --------------------------
void main()
{  int zahlen[N],       //  Feld von einzulesenden und zu bearbeitenden Zahlen
       ober=0;            //  Index für nächsten freien Eintrag

   einlesen(zahlen,ober,N-1); // ober wird als Referenzparameter behandelt !
                                 // Bei Rückkehr von einlesen ist ober-1 gleich dem Index
```

```
                                // für die zuletzt eingelesene Zahl

  bubble(zahlen,ober-1);   // Die Einträge im Feld zahlen werden sortiert

  abfragen(zahlen,ober-1);              //  Nachfragen nach eingelesenen Werten,
                                        //  mit Hilfe der Funktion binsearch().

  ausgabe(zahlen,ober-1);  // Das sortierte Feld wird ausgegeben.

 return;
}

void einlesen(int feld[], int& index, int maxind )    //******  einlesen **************
{
  index = 0;       //  Erster Wert soll in Position 0 eingelesen werden
  do
  { cout <<"\nBitte, eine ganze Zahl eingeben,  Ende mit ^Z ";
    cin  >> feld[index++];    // Referenzparameter index wird nach dem Einlesen hochgezählt.

  } while  ( (cin.eof() == 0)  && ( index <=maxind) ) ;

  if (index <= maxind) index--;          //  den gegebenfalls eingelesenen Wert für ^Z
                                // ignorieren wir !

  cin.clear();
  return;
}

void bubble( int feld[], int anzahl)        // ***********Bubble Sort Funktion *********
{ int i,j,temp;

 for ( i = anzahl; i>0; i--)
   for ( j = 0;j<i; j++)

    if  ( feld[j] > feld[i] )
     {  temp   = feld[i];               // vertauschen
           feld[i] = feld[j];
           feld[j] = temp;
     }
 return;
}
void abfragen( int feld[], int anzahl)   // ************ Abfragen Funktion ***************
{  int such,         // einzulesende Suchzahl
      gefunden;     // Index für in feld gefundene Zahl
 do
 { cout <<"\nBitte, eine zu suchende ganze Zahl eingeben, Ende mit ^Z ";
   cin >> such;

   gefunden = binsearch( feld,    anzahl, such);

   if( gefunden == -1 )  cout <<"\n Die Zahl " <<such  <<" ist nicht im Feld vorhanden ";
   else             cout<< "\n Die Zahl " <<such <<"ist im Feld in Position " <<gefunden;

 } while ( cin.eof() == 0);

 cin.clear();
 return;
}

int  binsearch( int feld[], int anzahl, int suchwert)
{ int unten=0, oben=anzahl, mitte;
 while ( unten < oben )
 { mitte =  ( unten + oben ) / 2;
   if (suchwert > feld[mitte] )
           unten = mitte+1;
```

```
   else
          oben = mitte;
  }
  return suchwert == feld[unten] ? unten : -1;

}

void ausgabe( int feld[], int anzahl )
{  cout <<"\n\n Ausgabe--------  :  \n\n";
  for (int i=0; i<=anzahl; i++)
    cout  <<feld[i]  <<"   ";

  return;

}
=======================================================
=======================================================

#include <iostream.h>
#include <string.h>

#define N 3                          // Dimension für Mitarbeiterliste[N]

struct  Name       { char nachname[30];
                     char vorname [30];
               };

struct Adresse     {    int   plz;         // Postleitzahl
                        char  ort[30];
                        char  strasse[30];
                              int   hausnr;
               } ;

struct Mitarbeiter { Name  name;
                              int alter;
                              Adresse adresse;
                      } ;

typedef Mitarbeiter  Mitarbeiterliste[N];

void ausgabe(Mitarbeiterliste personl, int n);



void main()         // -------------   Hauptprogramm  --------------------------
{
   Mitarbeiterliste liste  =
          {  "Theml", "Gudrun"    , 45, { 70563, "Stuttgart", "Bachstr ", 28 } ,
                    "Binz ", "Gerrit "    , 48, { 12345, "Ettlingen", "Herzweg ", 15 } ,
                          "Koller" , "Robert"  , 61, { 75321, "Wangen"   , "Säntisweg", 8 }  };

   ausgabe( liste, N);
   return;

}


void ausgabe(Mitarbeiterliste  personal, int n)   // ---------  Ausgabe  ----------------
{
  for (int i = 0; i<n; i++)
  { cout << "\n\n  Name : "  << personal[i].name.vorname  <<" "  <<personal[i].name.nachname;
    cout << "\n  Alter: "  << personal[i].alter;
           cout << "\n  Ort  : "  << personal[i].adresse.plz   <<" "  << personal[i].adresse.ort;
           cout << "\n         "  << personal[i].adresse.strasse  <<" "  << personal[i].adresse.hausnr;
  }
```

}

4.5 Zeiger auf Funktionen, Funktionsnamen als Parameter

Es ist möglich, die Adresse einer Funktion zu nehmen und in einem sogenannten Funktionszeiger abzulegen. Der Zeiger kann dann zum Aufruf einer Funktion verwendet werden.

Wenn die Adresse einer Funktion genommen wird, kann der resultierende Pointer dazu verwendet werden, die Funktion aufzurufen :

,

```
 void error(char* p)  {  .... }      //     Normale Funktionsdefinition

 void (*efct)  (char*);  //efct ist ein Zeiger auf Funktion mit Parameter char* und Resultat void

 void f()
       {
          efct = &error ;          // efct zeigt nun auf error
          (*efct) ("error");       // ruft error über efct auf ;   oder nur    efct("error");
        }
```

Diese Notation für Funktionszeiger macht es auch möglich, in der Parameterliste einer Funktion die Adresse einer weiteren Funktion als Parameter anzugeben.

Beispiel :

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

double quadrat( double x, double ( * funkt) (double z)  )
    //  Der zweite Parameter ist ein Funktionszeiger,  für eine Funktion mit double Parameter,
   //   und Resultat vom Typ double.
         {
           return  funkt(x)  * funkt(x) ;        //  quadrat  multipliziert Funktionsergebnisse
         }

double ksin(double x)                          //  ksin paßt zu funkt
               {  return  sin(x * M_PI / 180 ) ;
               }

double kcos(double x)                          // kcos paßt auch zu funkt !
              {  return cos( x * M_PI / 180 );
              }

main()
{  double  x, sinq, cosq ;
   cout.precision(16);
   cout <<"\nBitte, Winkel in Grad -> " ;  cin >> x;    cin.ignore(80,10);
```

```
   sinq = quadrat(x, ksin) ;     // Funktionsname ksin als aktueller Parameter->  sin(x) * sin(x)
   cosq = quadrat(x, kcos);      // Funktionsname kcos als aktueller Parameter -> cos(x) * cos(x)
   cout <<"\nSumme der Quadrate von sin(x) und cos(x) : "    << sinq  + cosq ;
   return 0;
}
```

4.6 Inline Funktionen, Makros

In Programmschleifen wie bei Rekursion können häufig wiederholte Funktionsaufrufe den Programmablauf merklich verlangsamen. Bei den bisher behandelten Funktionen kommt der Funktionsblock nur einmal im Programm vor, und bei jedem Aufruf wird zu dieser Funktion als Unterprogramm verzweigt.

Nun kann man einen Unterprogrammteil auch **als Makro definieren**, wobei dann dieser Programmabschnitt überall dort, wo er aufgerufen wird, den Aufruf ersetzt und eingefügt wird.

In C kann dazu die Preprozessoranweisung **#define** benutzt werden, in C++ gibt es zusätzlich noch die Möglichkeit von sog. **inline Funktionen.** Die Verwendung solcher Makros macht ein C/C++ - Programm mit vielen kleinen Funktionen auch hinsichtlich der Ausführungszeit effizient, teilweise allerdings auf Kosten von mehr Programmspeicher.

Wir befassen uns hier zunächst mit #define. Diese Preprozessoranweisung haben wir früher schon bei der Definition von Symbolkonstanten kennengelernt :

```
#define   N  500
```

Mit #define können wir für jeden Namen einen Ersatztext definieren :

```
#define  forever  for  (;;)    /* Endlosschleife  */
```

Normalerweise ist der Rest der Zeile der Ersatztext; eine lange Definition kann über mehrere Zeilen fortgesetzt werden, indem man \ an das Ende jeder Zeile stellt, die fortgesetzt werden soll.

Man kann auch **Makros mit Parametern** definieren, so daß der Ersatztext bei verschiedenen Aufrufen des Makros verschieden sein kann. Solche Makros sehen dann wie richtige Funktionen aus, sind aber keine Funktionen, da der Ersatztext an der Aufrufstelle im Programm vom Preprozessor eingefügt wird :

```
#define    AUSGABE(z)      putchar(z)
                  ---->>  Aufruf :  AUSGABE('P'); --> putchar('P');

#define   MULTI(x,y)     x  *  y
              ---->>  Aufruf :  MULTI(3 , 4)   --->      3 * 4
              ---->>  Aufruf :  MULTI(a+b, c-d)  -->  a + b*c-d     Fehler   !!

#define MULTI(x,y)     (x) * (y)
                  ---->> Aufruf :  MULTI(a+b, c-d)  -->  (a+b) * (c - d)  richtig!
```

Bei solchen Makros ist also sorgfältig auf das Setzen von Klammern im Ersatztext zu achten. Sind als aktuelle Parameter solcher Makros Ausdrücke zu erwarten, dann sollten die formalen Parameter wegen der Vorrangregeln geklammert werden.

Zu beachten ist auch, daß bei solchen Makros die Korrektheit der übergebenen Parameter nicht überprüft wird. Wird beim Aufruf z.B. eine Text statt Zahlenwerten als Parameter verwendet, so wird der Compiler diesen Fehler nicht immer merken.

Durch das **Kennwort inline** lassen sich in C++ (nicht in C !) auch Funktionen aufbauen, die nicht aufgerufen, sondern direkt in den Code eingebaut werden. Solche Funktionen nennt man **inline Funktionen oder auch Makros**. Beispiele für solche inline Funktionen sind :

```
1.)  inline int quadrat(int x) {  return  ( x * x); } // Funktionsaufruf z.B. quadrat(56),  oder
                                                      //  quadrat(r)

2.)  inline  int maximum(int a,  int b)  {  return  (a > b) ? a : b  ;   }
```

Solche inline Funktionen müssen immer vor ihrem Aufruf als Definitionen stehen. Prototyp Deklarationen dürfen hier nicht verwendet werden.

Bei diesen inline Funktionen überprüft der Compiler die aktuellen Parametertypen gegen die formalen Parametertypen. Dies ist ein wesentlicher Vorteil in C++ gegenüber den #define Makros in C.

Auch hier hat sich die Programmiersprache C++ vom Preprozessor mehr und mehr unabhängig gemacht.

5.) Strukturierung von unstrukturierten Programmablaufplänen

Die sequentielle Notation der Anweisungen bestimmt die sogenannte **statische Struktur des Programms.** Dabei können einzelne Anweisungen (wie z.B. if, while) Klammerfunktionen mit umschlossenen Anweisungen aufweisen. Auch können beliebige Verschachtelungen von solchen Klammerfunktionen auftreten.

Als **dynamische Struktur eines Programms** sei hier die während einer Programmausführung resultierende Reihenfolge der ausgeführten Anweisungen bezeichnet.

Die Forderung für ein gut **strukturiertes Programm** lautet : die statische und die dynamische Struktur des Programms müssen übereinstimmen.

Diese Forderung lässt sich nur in Programmen **ohne goto Anweisungen** erfüllen.
Im Jahr 1966 haben Boehm unf Jacobini ein Theorem aufgestellt, dass drei Programmkonstrukte, nämlich
Sequenz,
Selektion, und
Iteration
ausreichen, um jeden beliebigen Algorithmus zu beschreiben.

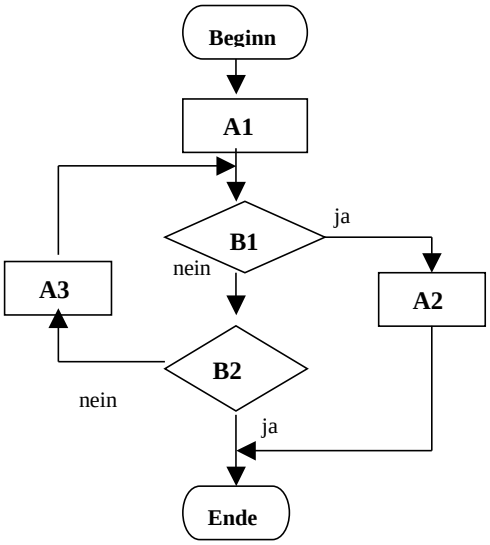
In einem so strukturierten Programmm hat jedes Programmsegment einen Eingang und einen Ausgang, mit beliebigen inneren Abläufen.
Nach DIN 66261 kann man jedes strukturierte Programm mit Hilfe eines Struktogramms darstellen.

Bei der Verfeinerung einer Operation kann ein PAP entstehen, der nicht in dem eben formulierten Sinn strukturiert ist. Es können dabei Operationen entstehen, welche der Forderung 'ein Eingang und ein Ausgang' nicht genügen. Gegebenenfalls müssen dann solche Operationen im Sinne der *strukturierten Programmierung'* neu geordnet werden.
Insbesondere das Vorkommen von goto-Anweisungen erzeugt Unterschiede zwischen statischer und dynamischer Struktur eines Programms.

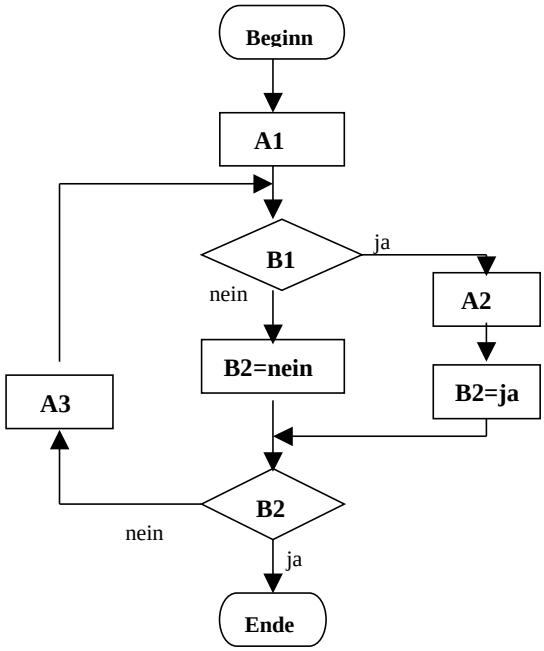
Auch beim Reengeneering älterer Programmprojekte sind gelegentlich unstrukturierte Programme zu bearbeiten. Hier entsteht dann die besondere Aufgabe, die entsprechenden Programmablaufpläne zu strukturieren.

Die Vorgehensweise wird anhand von einem Beispiel erläuter :

Unstrukturierter Programmablaufplan :



Strukturierter Programmablaufplan:



Im unstrukturierten Beispiel fürhren die beiden Zweige von B1 zu keinem gemeinsamen Ausgang.
Im strukturierten Beispiel endet der ja-Zweig von B1 vor der Bedingung B2 : um den Algorithmus beizubehalten, wurde für diesen Zweig die Bedingung B2 durch zusätzliche Anweisungen wirkungslos gemacht.

Wie hier beispielhaft gezeigt, lassen sich Einssprungstellen durch Einfügen von speziellen Anweisungen verschieben; mit solchen zusätzlichen Anweisungen werden gegebenenfalls Bedingungen wirkungslos gemacht.

6.) Präcompiler

A.) Allgemeine Aufgaben :

- Der Präcompiler läuft vor dem C / C++ Compiler. Er verarbeitet alle Anweisungen, welche mit einem # beginnen.
- Präcompiler Anweisungen können überall im Programm stehen, und irgendwo in einer Zeile beginnen, wenn in dieser Zeile nur Leerzeichen oder Tabulatorzeichen vorangehen.
- Am Ende einer Präcompiler Anweisung steht kein Semikolon !
- Inhalte anderer Dateien (Files) werden im Quellcode eingefügt (#include Anweisung)
- Texte im Quellkode werden ersetzt (#define Anweisung) : Macro Facility

- Steuerung von Ausblenden bestimmter Quelltextsegmente bei der nachfolgenden Übersetzung : ' Bedingte Compilation ' (#if u. a.)

- Der Präcompiler meldet keine Fehler, da bei der Text-Substitution keine Syntax und keine Semantik überprüft wird.

- **Borland C++ liefert im separaten Programm CPP.EXE** eine Möglichkeit. die Wirkung des Präprozessors anzuschauen: für ein C++ Programm Beisp1.cpp geben wir einfach ein : CPP Beisp1 .

B.) #include Anweisung

- Die #include Anweisung spezifiziert eine Datei, deren Inhalt diese #include Anweisung ersetzt. Entsprechende Quelltexte werden also eingefügt.

- #include <iostream.h> // Der Präcompiler sucht die Datei im Standardverzeichnis
  // für Bibliotheksroutinen

- #include "A.Eingabe.cpp" // Dateisuche beginnt im angegeben bzw. im aktuellen
  // Verzeichnis, und wird nötigenfalls im
  // Standardverzeichnis fortgesetzt.

- Header Dateien in der Form <iostream.h> , "projekta.h" beinhalten normalerweise nur Deklarationen (z.B. Funktionsprototypen, ADT's).

- Header Dateien in der Form "projekta.cpp" enthalten dann den dazugehörenden ausführbaren Quellcode (z. B. die Funktionsdefinitionen).

- Quelltexte, welche durch #include eingefügt werden, können wiederum #include Anweisungen beinhalten : Solche Verschachtelungen können auch mehrfach auftreten.

C.) #define, #undef Anweisungen : Macro Facility

- Diese Präcompiler Anweisung #define liefert zwei Möglichkeiten, um spezielle, im Quelltext auftretende Namen durch einen in #define festgelegten Text zu ersetzen.
- Ein Beispiel für die erste Möglichkeit ist : **Makro ohne Parameter**

```
#define  N  20
#define FEHLER1  "Die eingegebene Zahl war zu groß "
.......
int feld[N];      // N wird durch 20 ersetzt
....
cout <<FEHLER1 ;  // wird ersetzt durch   cout <<"Die eingegeben Zahl war zu groß" ;
```

- Bei dieser ersten Möglichkeit spricht man auch von einem **Makro ohne Parameter.** Kommt ein Makroname (wie hier N oder FEHLER1) innerhalb einer durch "-Zeichen eingeklammerten Zeichenfolge vor, so wird er nicht ersetzt.

  Im obigen Beispiel würde also cout <<"FEHLER1"; nicht ersetzt werden.

- Die #define Zeile darf nicht mit einem Semikolon abgeschlossen werden, weil dieses Semikolon sonst auch im ersetzten Text erscheint !

- Ein längerer Ersatztext kann am Zeilenende nach dem Zeichen \ in der nächste Zeile fortgesetzt werden :

```
#define  TEXT  Die ist ein langer Text,  \
                der nicht in eine einzige Zeile paßt.
```

- In C++ kann man #define Anweisungen durch const - Deklarationen ersetzen. In ANSI C kann man solche in const definierte Namen nicht in sog. konstanten Ausdrücken verwenden (z.B. case - Anweisung). C++ hat sich damit vom Präcompiler unabhängiger gemacht.

- In C++ ist es üblich, für solche **Makronamen** große Buchstaben zu verwenden.

- Und nun ein Beispiel für die zweite Möglichkeit: **Makro mit Parameter**.

```
#define MULT( w1, w2)       (w1) * (w2)
....
int x= 5,  y = 20,  z1, z2;
...
z1 = MULT(x,y);              // ersetzt durch   z1 =  (x) * (y);
..
z2 = MULT(x+8, y-1);      // ersetzt durch  z2 = (x+8) * (y-1);
```

- Für die vorangehende Berechnung für z2 sind die zusätzlichen Klammern in #define unbedingt nötig !

- Bei solchen **Makros mit Parametern** wird wie bei Funktionen unterschieden zwischen **formalen und aktuellen Parametern.** Beim Makroaufruf wird jeder formale Parameter durch einen entsprechenden aktuellen Parameter ersetzt, allerdings **ohne Typprüfung !**

- Das Makro wirkt hier wie eine Funktion, wobei der Funktionsaufruf durch die Funktionsdefinition direkt ersetzt wird.

- In C++ kann man statt solcher define - Makros auch sog **inline Funktionen** spezifizieren :

```
inline int mult(int w1, int w2)  {  return  w1*w2;  }

void main ()  { int x= 5,  y = 20,  z1, z2;
                  ......
                  z1 = mult(x,y);
                  .........
                  z2 = mult(x+8, y-1);    // Auf dem Stack liegen Kopien
                                          // der aktuellen Parameterwerte !
                }
```

- Bei inline Funktionen unterliegen die aktuellen Parameter denselben Typprüfungen wie bei normalen Funktionen. In C++ sind also solche #define Makros mit Parametern eigentlich unnötig.

- Aufrufe von Präcompiler Makros bewirken lediglich eine reine Textersetzung; erst der ersetzte Text muß syntaktisch richtig sein.

- **#undef :**
  Damit werden entsprechende, mit #define getroffenen Vereinbarungen wieder rückgängig gemacht:

```
#undef  N
```

D.) #if, #else, #elif, #endif Anweisungen : Bedingte Compilation (1)

- Mit diesen Anweisungen lassen sich bestimmte Programmteile bedingt ausschließen.
- Ein einfaches Beispiel :

```
#if konst. - Ausdruck

    Quellprogrammteil

#endif
```

Ist in diesem Beispiel der konst. - Ausdruck ungleich 0, werden die nachfolgenden Zeilen bis #endif, #elif, oder #else eingefügt, sonst nicht.

Jede #if Anweisung endet mit #endif; dazwischen können Folgen von #elif und #else stehen.

Im konst. - Ausdruck dürfen keine variablen Werte vorkommen, da der Präcompiler die Bedingung noch vor dem eigentlichen Übersetzen auswerten muß. Die Operanden eines konst. - Ausdrucks müssen also symbolische Konstanten oder konstante Ausdrücke sein.

- Ein weiteres Beispiel : Bei einer Übersetzung soll zwischen einer Demoversion und Vollversion unterschieden werden .

```
#define DEMO  1
  .....
 #if DEMO
     double vector[10];          // Demoversion
#else
     double vector[1000];        // Vollversion
#endif
```

Hier kann zu Programmbeginn festgelegt werden, ob beim Übersetzen eine Demoversion oder eine Vollversion des Programms erzeugt wird.

- Im vorangehenden Beispiel kann (nur in C++ !) statt #define DEMO 1 auch const DEMO = 1; stehen.

- Eine vorangehende Formulierung #define DEMO führt bei nachfolgendem #if DEMO zu einer Fehlermeldung, da DEMO zwar als Makroname definiert ist, jedoch ohne Ersatzwert.
  Zu beachten ist der Unterschied zu #ifdef (siehe unten).

♦ #elif funktioniert wie else if !

E.) #ifdef, #ifndef Anweisungen : Bedingte Compilation (2)

♦ Definierte Größen können in der bedingten Compilierung mit #ifdef bzw. #ifdef Anweisungen abgefragt werden.

#ifdef ist dabei eine Abkürzung für #if defined,
#ifndef ist eine Abkürzung für #if not defined .

♦ Ein Beispiel :

```
  #define  GROSS
    ....
#ifdef GROSS
    int feld[1000];
#else
    int feld[5];
#endif
```

♦ Der Operand in der #ifdef - Anweisung ist hier der **Makroname GROSS**.

Die Formulierungen #define GROSS = 1 oder #define GROSS = 0 führen hier bei #ifdef GROSS zur gleichen Entscheidung wie einfach #define GROSS.

♦ Mit einem im Quellprogramm nachfolgenden #undef GROSS wird die Wirkung von #def GROSS aufgehoben.

♦ Die #include - Anweisungen können verschachtelt werden. Die folgende einfache Konstruktion zu Beginn jeder Include - Datei verhindert dabei doppelte Definitionen :

```
#ifndef   NAME
#define NAME

  ..... Quellenprogramm,  z.B.  Include - Datei
#endif
```

Falls, bei verschachtelten Include - Dateien, das gleiche Programm mit obiger Konstruktion mehrfach angezogen wird, erfolgt das wirkliche Einfügen nur ein einziges mal.

Ohne diese Konstruktion würde der Compiler gelegentlich gleiche Programmteile innerhalb einer Datei mehrfach antreffen, und entsprechende Fehlermeldungen produzieren.

♦ Die eben besprochene Konstruktion spielt beim Aufteilen größerer Programme in mehrere Dateien (Modularisierung von Programmen) eine wichtige Rolle !

F.) #error

♦ Beim Antreffen von

```
#error  Fehlertext
```

wird der Übersetzungsprozess an dieser Stelle abgebrochen, an der #error erscheint, und es wird eine Compiler Fehlermeldung mit dem Fehlertext ausgegeben.

Diese #error Anweisung wird gelegentlich beim Austesten benützt.

7.) Abstrakte Datentypen und Modularisierung

7.1 Abstrakte Datentyypen

Beim Entwurf von Programmen entwickeln wir gelegentlich eigene, sog. **benutzerdefinierte Datentypen,** welche eine Erweiterung der in der Programmiersprache vordefinierten Typen darstellen. Beispiel dafür sind Felder und Strukturen.

Grundsätzlich können wir dann im Programm mit solchen selbstdefinierten Datentypen in zwei verschiedenen Arten arbeiten :

- Bei allen Operationen referieren wir die Elemente der Datentypen direkt.
- Für alle einen Datentyp betreffende Operationen werden jeweils spezielle Funktionen implementiert; nur über diese Funktionen werden dann die gewünschten Operationen verwirklicht.

Einen solchen benutzerdefinierten Datentyp nennt man in Verbindung mit den dazu gehörenden Funktionen einen **abstrakten Datentyp.** Dieses Konzept eines abstrakten Datentyps wird insbesondere bei Datenstrukturen höheren Abstraktionsgrades verwendet.

Ein einfaches Beispiel soll den Unterschied der beiden Verfahren zeigen:

Bei beiden nachfolgenden Verfahren liege die folgende Deklaration zugrunde :

```
typedef struct bruch { float zaehler;
                       float nenner ;
                     }
```

1. Direkter Zugriff zu den Elementen einer Struktur

```
void main()
{ bruch b1 = {3,5 }, b2 = { 2,7}, bsumme;

  bsumme.zaehler = b1.zaehler * b2.nenner + b2.zaehler * b1.nenner;
  bsumme.nenner = b1.nenner * b2.nenner;

   // Ausgabe !
}
```

2. Zugriff zu den Elementen mittels einer Funktion

```
bruch addbruch(bruch * a,* b );

void main
{ bruch b1 = {3,5 }, b2 = { 2,7 }, bsumme;

  bsumme = addbruch( &b1, &b2 );

 // Ausgabe !
}

bruch addbruch( bruch * a, *b)
{ bruch bs;
   bs.zaehler = (*a).zaehler * (*b).nenner + (*b).zaehler * (*a).nenner;
   bs.nenner = (*a).nenner * (*b).nenner ;
}
```

Beim Vergleich der beiden Verfahren ist zu beachten, daß das Hauptprogramm main() die eigentliche Anwendung repräsentiert. Beim zweiten oberen Beispiel ist der C++ Quellcode unabhängig von den internen Implementationsdetails der Funktion addbruch.

Beispielsweise könnte die oben vorangehende typedef Deklaration auch ein zweistelliges Feld darstellen. Dann würde sich das main( ) - Programm beim obigen ersten Verfahren ändern, ebenso die Definiiton der Funktion addbruch() beim zweiten Verfahren, nicht jedoch die Anwendung beim zweiten Verfahren.

Für einen ADT bieten die Funktionen eine vollständige Schnittstelle zur Anwendung. Einzelheiten der internen Implementierung bleiben dabei dem Benutzer verborgen ('information hiding'). Der Programmierer kann für jeden ADT einen speziellen Satz von Funktionen definieren; solche speziellen Funktionen bilden dann eine Ergänzung der Operatoren aauf die Grunddatentypen.

Auch haben die einzelnen Funktionen noch eine eigene Kapselungseigenschaft : alle innerhalb der Funktion definierten Daten sind vor fremdem Zugriff geschützt (*information hiding).*

Dieses ADT Konzept erleichtert insbesondere die **Bildung einzelner Moduln** bei der Organisation von großen Programmen **:** jedes ADT ist zusammen mit seinen Schnittstellen - Funktionen als eine einzelne Programmdatei (Modul) implementiert. Dadurch wird es leichter, ein umfangreiches Programm zu verstehen.

7.2 Modularität innerhalb einer einzigen C++ Datei

- #include – Anweisungen für

  Bibliotheksroutine : #include <iostream.h>

  Deklaration, ADT's: #include "declar.h"

  Anwendungsprogramme: #include "funct.cpp"

Ein kleines Beispiel :

a) das Hauptprogramm :

```
#include  <iostream.h>
#include   "add.cpp"

void main()
{ double a=5, b= 10, c ;
  c = addiere(a,b) ;
  cout  << "\n Summe =  "    << c ;
}
```

Das übersetzte Hauptprogramm enthält hier auch die Funktion addiere.

b) Die Datei add.cpp :

```
double addiere(double x, double y)
        {    return (x + y ) ;
         }
```

- Globale Variable außerhalb von Funktionen

  Variablen, welche außerhalb aller Funktionen deklariert werden, sind vom Type *extern* und heißen *globale Variablen.* Sie können in der gleichen Datei in allen nachfolgenden Funktionen benützt werden.

- Deklaration von Funktionsprototypen

  Ein Funktionsprototyp ermöglicht es, die Funktionsdefinition dem Aufruf im Quellenprogramm nachfolgen zu lassen : lediglich die Deklaration des Prototyps der Funktion muß im Quellenprogramm vor dem Aufruf stehen. Die Funktionsdefinition und jeder Funktionsaufruf muß mit dem Prototyp übereinstimmen.
  Ein Beispiel :

```
#include <iostream.h>

void berechne(double radius);       // Funktionsprototyp

double umfang;                      // Globale Variable umfang

void main()
{ double radius;
  cout   <<"\n Bitte einen Wert für den Durchmesser eingeben ";
  cin    >>  radius;
  berechne(radius);
  cout <<"\n Der Umfang beträgt : "  << umfang;
}



void berechne(double radius )       // Funktionsdefinition
{ umfang = 2 * pi *radius ;
}
```

- Lokale Variablendeklaration mit dem Zusatz extern

Im folgenden Beispiel will man eine globale Variable nur ganz bestimmten Funktionen im Programm bekannt machen. Dazu kann man die externe Deklaration dieser globalen Variable auf diese Funktionen beschränken und die Definition im Programm später folgen lassen :

```
main()
{   extern  float   gvariable;       // externe Deklaration innerhalb von main()
       ....
}
float  funktion1(...)
{  extern  float  gvariable;       // externe Deklaration innerhalb von funktion1
           ...
}
float funktion2( .. )
{
```

```
   .....
}
float gvariable;                  // Definition der globalen Variable gvariable

float funktion3( .. )
{
   .....
}
```

Beim obigen Beispiel ist die globale Variable gvariable in main, funktion1 und in funktion3 bekannt, nicht aber in funktion2.

- Lokale statische Variable

  Beispiel :

```
#include <stdio.h>

void main()
{ int i, summe = 0,
      feld[3][3] = { 1,2,3,  4,5,6,   7,8,9 };

  for (i=0; i<3; i++ )
    {                          //  Schleifenblock
     static int j;             //   j ist nur in diesem Block bekannt.
                               //   Beim ersten Blockeintritt wird j = 0 gesetzt

     summe += feld[i][j++];   // Summe der Diagonalwerte !
    }
  printf("\n%d", summe);

  return;
}
```

- Globale statische Variable

  Auf globale statische Variablen kann nur innerhalb der gleichen Quelldatei zugegriffen werden.
  Damit kann also eine globale Variable für eine einzelne Datei lokal gemacht werden.

7.3 Mehrere Dateimodule innerhalb eines C++ Projekts

In C ist es möglich, ein Quellprogramm in mehreren Teilen getrennt in verschiedenen Dateien zu halten. Solche einzelne **Moduln** können dann getrennt compiliert und anschließend zu einem gemeinsamen Programm zusammen - gelinkt werden.

Nun muß man dem Compiler mitteilen, daß evtl. eine in einer Datei definierte globale (externe) Variable auch in einer anderen Datei (Modul) benutzt werden kann.

Dies geschieht in der anderen Datei durch eine **externe Deklaration :**
Quelldatei Progr1.cpp:

```
float xkoordinate;                  // globale,  externe Variable
```

```
main()
{    int index = 5;
     float  y;
     y  =  xkoordinate * index;
    ..  ...
}
```

Quelldatei Progr2.cpp :

```
extern  float xkoordinate ;                //   extern Deklaration für xkoordinate

float berechne()
{  ....
    xkoordinate  =  200;
   .....
}
```

In der Datei Progr1.cpp ist die globale Variable xkoordinate definiert.

In der Datei Progr2.cpp steht mittels des Zusatzes extern nur eine Deklaration (keine Definition) für diese Variable. Dieser explizite Zusatz extern dient hier also dazu, dem Compiler und dem Linker solche globalen Variablen in der Datei Progr2.cpp bekannt zu machen.

Für die Variable xkoordinate wird im übersetzten Programm für Progr1.cpp ein Speicherplatz reserviert.

Anstatt der globalen externen Deklaration in Progr2.cpp könnte hier auch innerhalb spezieller Funktionen eine solche externe Deklaration stehen: dann wäre in dieser Quelldatei diese externe Variable eben nur innerhalb dieser Funktionen bekannt gemacht.

Die Speicherklasse extern läßt sich auch auf Funktionsdeklarationen anwenden und hat eine dementsprechende Bedeutung.

Ein Programm hat dann in einer ersten Quelldatei eine Funktionsdefinition, mit oder ohne einen Prototyp, in der anderen Quelldatei steht nur noch die Funktionsprototyp - Deklaration mit vorangehenden extern :

```
extern  int  funkt1(.....);          //   extern ist bei Funktionen optional
```

- Externe Referenzen für Variablen und Funktionen

Module A

```
  float  f;          // Neuer Speicher
static int  x;       //  Gültigkeit auf
                     //   Modul A beschränkt

 int berechne  (...)
  {  ....
     .....   }

static int ermittle()   // Gültigkeit auf
 { ...                  // Modul A beschränkt
       .....  }
```

Module B

```
extern float f;     // kein  neuer
                    // Speicher vorgesehen

int berechne;    // Nur Prototyp .
```

♦ In C / C++ werden Module Interfaces in *Header Files* mit der Dateierweiterung .h zusammengefaßt,

funktionale Implementationen (Module Implementations) werden in Dateien mit der Erweiterung .CPP zusammengefaßt.

7.4 Visual C++ : Verwalten von Projekten und Projektdateien

1. Ein Projekt mit leerem Hauptprogramm (int main(..)) erstellen

♦ Menue Datei | Neu | Register **Projekte**

Ø Aus der nun angebotenen Liste auswählen: **Win32 Konsoleanwendung**
Ø **Projektnamen** eingeben
Ø Pfad für (evtl. neuen) **Projektordner** eingeben
Ø Eingabe mit OK bestätigen.

♦ Im neuen Dialogfenster

Ø Auswählen : **Ein einfache Anwendung**
Ø Schalter *Fertigstellen* im Dialogfenster betätigen, und auch das nachfolgende Dialogfenster mit *OK bestätigen*

♦ Jetzt arbeitet der Rechner ein kurze Zeit. Danach erscheinen im Arbeitsbereichs – Fenster die **Register Klassen, und Dateien.**

2. **Ein C++ Quellenprogramm als Hauptprogramm im Projekt erzeugen**

♦ Über das Register Klassen die Datei mit der Funktion main(..) öffnen: diese Funktion enthält zunächst noch keine Anweisungen

♦ In der geöffneten Datei ein C++ Quellenprogramm in main(..) – Funktion einfügen. Wo wir die Anweisung *#include <iostream>* einfügen, hängt von der weiteren Programmierung ab. Im einfachsten Fall (nur eine einzige Programmdatei) kommt diese Anweisung an den Kopf dieser Datei.
Kommen weitere Module hinzu, so ist es besser, diese #include - Anweisung in eine gemeinsame Kopfdatei einzufügen, welche dann in alle Moduke eingebunden wird.

3. **Das Projekt übersetzen und ausführen**

♦ Die folgenden Ikonen bzw. Menueoptionen können aktiviert werden :

Ø Das Menue **Erstellen** kann mit seinen Untermenues benützt werden.

Ø **Erstellen F7** :ein lauffähiges Programm wird erstellt. Alle Programmteile werden dabei neu übersetzt.

Ø **Kompilieren Strg+F7 :**Eine Syntaxprüfung wird durchgeführt Es werden nur die .OBJ Dateien erstellt.

Ø **Ausführen F5**: Programmstart imDebugmodus

Ø **Programm ausführen Strg+F5**:Programmstart ohne Debugging
Ikone : Ausrufezeichen !

♦ Unter dem Debugger blitzt das Ergebnis des Programms nur ganz kurz auf. Ohne Debugging bleibt jedoch das Ergebnis sichtbar stehen.

3. Das Projekt schließen

♦ Ein offenes Projekt läßt sich schließen mit dem Menü :

Datei | Arbeitsbereich schließen

4. Ein Programm schrittweise ausführen

♦ Wir setzen mit **F9** eine Haltepunkt auf die erste Zeile unseres eingefügten C++ Programms.

♦ Das Programm wird nun gestartet mit **AusFühren F5** im Debug Modus. Am Haltepunkt bleibt das Programm stehen.

♦ Schrittweise weitere Ausführung :

Ø **F11 oder Ikone Schritt hinein:** wir gehen einen Schritt weiter.

Ø **F10 oder Ikone Schritt über :** wir gehen einen Schritt weiter, Funktionsaufrufe werden übersprungen.

Ø **Umsch.+F11 oder Ikone Schritt heraus :** Weiter bis Funktionsende

Ø Setzt man zwischendurch den Cursor auf eine Variable im

Quellentext und wartet einen Augenblick, so erscheint der Wert derVariablen dort angezeigt.

Ø Mit dem Start des Programms erscheint ein **Variablen – Fenster anstelle des Ausgabefensters.** In diesem Fenster können wir über die Register Auto, Lokal, this die Anzeige der Variablen steuern:

Auto : Der Debugger wählt selbst die wichtigsten Variablen zur Anzeige aus (z.B. alle in der aktuellen und der letzten Anweisung) .
Lokal : Erzwingt die Anzeige aller lokalen Variablen.

this : beschränkt die Anzeige auf das Objekt selbst.

5. Ein leeres Projekt erstellen.

- Zunächst verfahren wir wie beim allerersten Projekt, allerdings wählen wir im neuen Dialogfenster nicht *eine einfache Anwendung,* sondern **ein leeres Projekt,** und klicken auf *fertigstellen.*

- Wir müssen nun erneut das Menue Datei | Neu auswählen. Visual Studio wechselt nun automatisch auf die Registerkarte **Dateien.** Hier wählen wir den Typ **C++ Quellcodedatei** aus, markieren **dem Projekt hinzufügen**, und tragen einen neue Dateinamen sowie den geeigneten Ordnerpfad ein. Schließlich bestätigen wir mit *OK.*

- In die nun leere Datei geben wir unseren Quellcode ein.

7. Das Fenster Arbeitsbereich

Das Fenster *Arbeitsbereich* enthält die Register
Klassen,
Dateien,
Resourcen (nur bei Windows Anwendungen).

Diese Register sind allerdings nur sichtbar, wenn ein Projekt und damit zwangsweise ein Arbeitsbereich geladen ist.

Im Register *Dateien* sieht man alle zum Arbeitsbereich gehörigen Dateien, nach Projekten geordnet, angezeigt. Dies Zusammenstellung der Dateien ermöglicht die folgenden Funktionen :

- Anzeigen von Dateien durch Doppelklick auf den Dateinamen.

- Einfügen neuer Dateien in ein Projekt. Dies können insbesondere separate Module eines neuen Projekts sein. Durch Klicken mit der rechten Maustaste auf eine Projektnamen erhält man ein Kontextmenue, in dem sich die Auswahl **Dateien zum Projekt hinzufügen** auswählen läßt.

- Löschen von Dateien aus einem Projekt. Dazu wird der betreffende Dateiname markiert, und das **Menue Bearbeiten | Löschen** (bzw. die Entf. Taste drücken) ausgewählt.

8.) Rekursion

Als **Rekursion** bezeichnet man ein Verfahren, das auf sich selbst zurückgreift. In der Unterprogrammtechnik bedeutet dies, daß eine Funktion sich selbst aufruft.

Ein einfaches Beispiel ist die Berechnung von n! , die **rekursive Definition** lautet dafür :

$$0! = 1, \quad 1! = 1, \quad n! = n * (n - 1)! \qquad \text{fün } n>1.$$

Das folgende Beispiel zeigt eine einfache Anwendung :

```
#include  <iostream.h>

long double  fakul(int n)
{  if  ( n <= 1)  return 1;                  // Für diese beiden Zeilen könnten wir auch formulieren :
   else  return  n  *  fakul(n - 1) ;       //  return  n  ==  0  ?  1  :  n * fakul(n - 1);
}

main()
{  int n;
   cout << "\nBitte, ganze positive Zahlen eingeben, Ende der Eingabe mit  n < 0  ";
   do                              //  Leseschleife bis n negativ
     {   cout  << "\n ganze Zahl -> ";  cin >> n;
         if  ( n >= 0 )  cout  << "     "   << n  <<"! = "  << fakul(n);
     }
   while  ( n >= 0);
   return 0;
}
```

Mit Rücksicht auf den Zahlenbereich wurde in diesem Beispiel als Funktionstyp long double gewählt.

Das Hauptprogramm main() ist hier ganz unabhängig von der Arbeitsweise des Unterprogramms fakul . Anstelle der rekursiv arbeitenden Funktion fakul hätte auch eine iterativ mit einer for-Schleife arbeitende Funktion diese Arbeit tun können.

Im folgenden zweiten Beispiel kann man anhand der Bildschirmausgabe verfolgen, wie die Werte, die durch jeden neuen rekursiven Aufruf erzeugt werden, zwischengespeichert werden und beim jeweiligen Heraussprungen aus der Funktion wieder zur Verfügung stehen.

```
#include  <stdio.h>

void  lies()
{   float zahl;
    scanf("%f",  &zahl);
    if  (zahl  != 0)
        {
            printf("\n%p  zahl  =  %8.2f",  &zahl,  zahl);
            lies();                    // Rekursiver Aufruf von lies()
            printf("\n%p  zahl  =  %8.2f",  &zahl,  zahl) ;
        }
    return;
}

void  main()
{  printf("Gib Zahlen ein, letzte Zahl 0 :\n");
    lies();
```

```
    return ;
}
```

Am Bildschirm ergibt sich dabei beispielsweise :

```
Gib Zahlen, letzte Zahl 0;
5  6  7  0
7804 : 0EBA zahl  =   5.00
7804 : 0EB0  zahl  =   6.00
7804 : 0EA6  zahl  =   7.00

7804 : 0EA6  zahl  =  7.00
7804 : 0EB0  zahl  =  6.00
7804 : 0EBA zahl  =  5.00
```

In einer Funktion werden die Werte für Parameter und lokale Variablen auf dem Stack (Stapel) abgespeichert, so auch die Variable zahl im obigen Funktionsbeispiel.

Die Funktion lies beginnt mit dem Einlesen einer Zahl von der Tastatur; der eingelesene Wert wird in zahl auf dem Stack abgespeichert. Dann wird lies() wieder aufgerufen, auf dem Stack wird bei diesem neuen Aufruf eine neue Variable zahl plaziert, usw. bis eine 0 eingelesen wurde. Erst danach wird genauso oft die dem rekursiv aufgerufenen lies() folgende printf-Anweisung aufgerufen und der Zahlenwert für zahl erneut ausgegeben.

Die Zahlenwerte werden also in eine "Vorwärtsreihenfolge" auf dem Stack weggespeichert und beim jeweiligen Beenden der Funktion lies() wieder in einer "Rückwärtsreihenfolge" zurückgeholt.

Das Speicherprinzip des Stacks wird auch **Last in - First out** genannt.

Durch den wiederholten Funktionsaufruf sind rekursive Programme immer etwas langsamer als entsprechende Schleifenprogramme. Sie lassen sich aber oft einfacher und eleganter programmieren ! Beim rekursiven Aufruf von Funktionen kann es zu einem Stapel - Überlauf kommen, wenn bei jedem Aufruf viele Parameter und lokale Variablen auf dem Stapel neu gespeichert werden müssen.