

Programmierung in C/C++

Entwicklungsgeschichte

Die **Programmiersprache C** wurde in den 70-er Jahren von Kernighan und Ritchie entwickelt. C entstand im Zuge der Entwicklung des Betriebssystems UNIX. Mit Hilfe der Sprache C sollte UNIX flexibel und weitgehend Hardware unabhängig (leicht portierbar) werden.

So entstand ursprünglich der **Kernighan-Ritchie-Standard** (K&R).

In den 80-er Jahren wurde die Sprache C vom American National Standardisation Institute (ANSI) überarbeitet und als neuer **ANSI-C-Standard** festgelegt.

Das Grundkonzept war eine Programmiersprache, die schnelle Programmausführung, kleinen Programmcode und leichte Portierbarkeit ermöglichte.

Die Erweiterung für objektorientiertes Programmieren wurde von Bjarne Stroustrup mit C++ im Jahr 1984 eingeführt. C++ enthält dabei C als Untermenge und wird C immer mehr ablösen.

Die **Programmiersprache C/C++** ist heute eine der am meisten verbreiteten Sprachen und wird auf vielen Plattformen, wie unter UNIX und auch Windows, eingesetzt.

C ist mittlerweile auch der Sprachstandard für programmierbare Mikrocontroller geworden.

Neuere Sprachentwicklungen, wie Java, orientieren sich auch stark an der Sprache C++.

C/C++ - Programm Übersetzung

Die Übersetzung vom Quelltext in ein lauffähiges Programm geschieht über 3 Schritte.

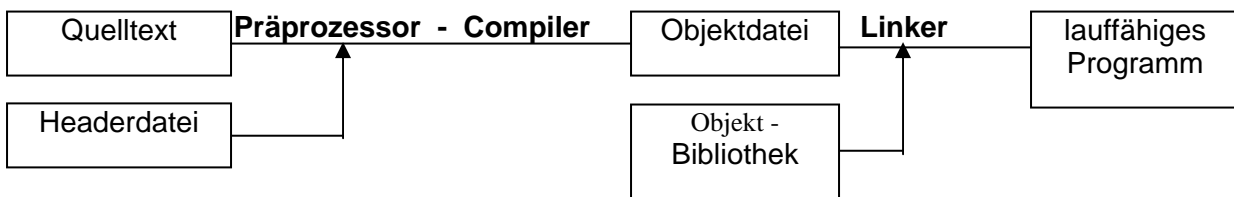
Zuerst wird das Quellprogramm durch den **Präprozessor** bearbeitet und es werden alle Präprozessor-Anweisungen ausgeführt (Textersatz, Einbinden von Quellprogrammteilen).

Der Präprozessor kann dabei natürlich nur auf Textebene (Quellprogrammebene) arbeiten.

Der **Compiler** übersetzt danach das Quellprogramm in eine Objektdatei. Die Objektdatei enthält den Maschinencode mit symbolischen Adressen für externe Daten und Programmteile.

Der **Linker** bindet dann Objekt-Bibliothekdateien ein, löst alle externen Adressen auf und erzeugt schließlich ein lauffähiges Programm.

Schema der Programmübersetzung



Entwicklungsumgebung

Zur Programmentwicklung wird eine Entwicklungsumgebung (IDE = Integrated Development Equipment) verwendet, die alle notwendigen Werkzeuge zur Programmerstellung enthält, wie Editor, Compiler, Linker, Projektverwaltung und Debugger.

Der Editor dient zur Programmerstellung im Quelltext, der Compiler u. Linker zur Übersetzung, der Debugger als Hilfsmittel zur Fehlersuche und die Projektverwaltung zur Erstellung größerer Programmaufgaben mit mehreren Quelldateien.

DevC++ ist eine frei verwendbare IDE unter der GNU General Public License (GPL) und verwendet die GNU Compiler Collection (GCC). GCC ist der Standardcompiler der LINUX- u. UNIX-Welt.

Die Windows Version von GCC erzeugt Win32-ausführbare Programme, die als Konsolen- oder auch GUI-Anwendungen laufen.

Informationen und Software zu DevC++ erhält man unter www.bloodshed.net

Die aktuelle C/C++ Sprachreferenz ist unter www.cppreference.com als Webmanual erhältlich.

Programm Aufbau

Ein C/C++ - Programm hat folgenden Aufbau :

```
#include <Headerdatei>      // Headerdatei einbinden
                           // zur Verwendung von Bibliotheksfunktionen
int main()                 // Hauptprogramm definieren
{   Anweisungen;          // Anweisungsblock
}
```

erstes einfaches Beispiel :

```
/*-----
   Hello World
-----*/

#include <iostream.h>      // Headerdatei einbinden

int main()                // Hauptprogramm definieren
{
    cout <<" hello world "; // Ausgabe eines Strings
}
```

Dieses Quellprogramm kann unter DevC++ über den **Menüpunkt „Kompilieren und Ausführen“** übersetzt und gestartet werden. Während des Übersetzungsvorgangs wird das „Compile Progress“-Fenster eingeblendet. Bei Programmfehlern wird die Liste mit Fehlermeldungen ausgegeben. Bei fehlerfreier Übersetzung öffnet sich ein Konsolenfenster (Textfenster) und zeigt die Textausgabe. Damit jedoch das Textfenster sichtbar bleibt, muss am Ende des Programms ein Haltebefehl gegeben werden, da es sonst unter DevC++ sofort wieder geschlossen wird.

Das Programm sollte auch immer eine ausführliche Beschreibung über Autor, Erstellungsdatum, Version und Aufgabe am Programmanfang und weitere Kommentare enthalten.

Daher sind **folgende Verbesserungen** notwendig :

```
////////////////////////////////////
//   Programmname : Schablone.cpp
//   Autor       : Name
//   Datum      : 01.09.2005
//   Version    : 1
//   Aufgabe    : Programmaufbau eines C++ - Programms
////////////////////////////////////

#include <iostream>          // neue Version von iostream.h
                           // verwendet den Namensraum std
#include <stdlib.h>          // Headerdatei für system()-Funktion

using namespace std;        // Namensraum für Standardbibliothek definieren

////////////////////////////////////
//   Hauptprogramm
////////////////////////////////////

int main()
{
    cout <<" hello world ";

    system("PAUSE");        // Programm am Ende anhalten
    return 0;              // Returncode für fehlerfreie Ausführung ausgeben
}
```

Kommentare

Kommentare dienen dem Programmierer zur Beschreibung des Quelltextes und werden vom Compiler ignoriert. In C/C++ können Kommentare auf zwei Arten ausgeführt werden.

Kommentare über mehrere Zeilen werden mit `/*` eingeleitet und mit `*/` abgeschlossen.

Einzeilige Kommentare werden mit `//` eingeleitet und automatisch mit Zeilenende abgeschlossen.

Beispiele :

```
/*  Kommentar über mehrere Zeilen
    ...
*/
//  Kommentar bis Zeilenende
```

Anweisungen

Ein Programm besteht aus Anweisungen (Befehlen).

Jede Anweisung muß in C/C++ mit einem Strichpunkt `;` abgeschlossen werden.

Es können dabei durchaus mehrere Anweisungen in einer Zeile ausgeführt werden.

Beispiele :

```
int i,j;                // Variablendefinition als Anweisung
cin >> i; j=2*i;        // zwei Anweisungen in einer Zeile
```

Anweisungsblock :

```
{ Anweisungen
}
```

Ein Anweisungsblock dient dazu, mehrere Anweisungen zu einer Anweisung zusammenzufassen.

Der Blockanfang wird dabei mit dem Zeichen `{` und das Blockende mit `}` ausgeführt .

Anweisungsblöcke werden beispielsweise bei Verzweigungen, Schleifen, Funktionen verwendet.

Um die Lesbarkeit des Programmtextes zu verbessern, werden üblicherweise alle Anweisungen im Block mit mindestens 2 bis 4 Leerzeichen eingerückt. Die Blockklammern für Anfang und Ende werden an gleicher Position untereinander platziert, damit der Block deutlich erkennbar wird.

Variablendefinition

In C/C++ müssen alle Variablen und Konstanten vor ihrer Verwendung zuerst vereinbart werden.

Die Variablendefinition gibt zuerst den Typ und dann den Namen der Variablen an.

Definition von Variablen :

Typ VariablenName ;

Beispiele :

```
int i;                // Integer Variable mit Namen i
int a1,a2;            // zwei Variable in einer Anweisungszeile
float x,y,z;          // Gleitkomma Variable
float n = 0;          // mit Initialisierung (Anfangswert zuweisen)
char c;               // Character Variable (für Zeichen)
```

Variable gelten **lokal** in dem Block, wo sie definiert werden, und auch in weiteren inneren Blöcken.

Globale Variablen werden außerhalb jedes Blockes definiert und gelten im gesamten Programm.

Beispiel :

```
int n = 10;           // globale Variable
int main()
{
    int i;            // lokale Variable i ist im main-Block gültig
    {
        i = 1;        // auch im inneren Block ist i gültig
    }
}
```

Definition von Konstanten : **const Typ KonstantenName = Wert ;**

Konstante bekommen bei der Definition einen Wert zugewiesen und sind unveränderlich.

Beispiel :

```
const int NMAX = 100;      // Integer Konstante
const float PI = 3.1415;   // Float Konstante
const K = 100              // Integer Konstante !
```

! Achtung : bei fehlenden Typangaben wird der Typ int automatisch angenommen !

! Fehler Beispiel :

```
const K_1_2PI = 1/6.28;    // ergibt k2=0 !!!
```

Namensregeln :

- Namen können Buchstaben, Ziffern und den Unterstrich ('_') enthalten
- Groß- und Kleinbuchstaben werden unterschieden (case sensitiv)
- Namen müssen mit einem Buchstaben oder '_' beginnen
- Namen dürfen nicht gleich einem C/C++ -Schlüsselwort sein (wie int, float, main, ...)

Beispiele :

```
int a_min;                  // Variablenname mit '_'
float wert,Wert;            // 2 unterschiedliche Variable
```

Zuweisungs-Anweisung

Eine Zuweisungsanweisung mit '=' hat immer die Wirkung, dass der rechte Ausdruck der linken Variablen zugewiesen wird. Damit wird i.a. auch der Wert der linken Variablen geändert.

Zuweisung : Variable = Ausdruck ;

Beispiel :

```
y = 2*x - 1 ;              // Wertzuweisung für y
i = j = 0 ;                // verkettete Zuweisungen
```

Typ-Umwandlung

C/C++ kann in Ausdrücken auch kombinierte Datentypen enthalten, wobei die Typ-Umwandlungen teils automatisch (implizit) durchgeführt werden. Wenn notwendig, kann auch eine aufgezwungene oder ausdrückliche (explizite) Typumformung ausgeführt werden.

Beispiel für automatische Typumwandlung :

```
int i; float x;
x = 0.5*i;                  // automatische Typumformung auf float
```

explizite Typumformung (type cast) : (Typ) Ausdruck

Beispiel :

```
int i; float y;
y = (float)1/i;           // notwendige explizite Typumformung
                           // um die Gleitkomma-Division zu erzwingen
```

! Achtung ! :

```
y = 1/i;                  // ergibt eine ganzzahlige Division
                           // und damit für i>1 immer 0
```

richtig hingegen ist auch :

```
y = 1.0/i;                // ergibt eine Gleitkomma-Division,
                           // da ein Operand vom Typ float ist
```

Ein - u. Ausgabe

Über Eingabefunktionen können Werte über Tastatur in Variable eingelesen werden und über Ausgabefunktionen können Ergebniswerte am Bildschirm ausgegeben werden. In C und C++ werden Ein-Ausgaben unterschiedlich ausgeführt. C stellt in der Bibliothek <stdio.h> die Funktionen scanf u. printf zur Verfügung, C++ bietet die Klassenbibliothek <iostream> für Datenströme.

Ein-Ausgaben in C++ :

Einlesen von Tastatur : ***cin >> Variable ;***
Ausgabe auf Bildschirm : ***cout << Ausdruck ;***

die erforderliche Headerdatei für cin, cout ist **<iostream>** oder <iostream.h>

Beispiel :

```
#include <iostream>
using namespace std;
int main ()
{ float x;
  cout << " Bitte geben sie eine Zahl ein ";
  cin >> x;
  cout << x << endl;
}
```

über die vordefinierte Konstante **endl** wird ein Zeilenvorschub ausgeführt. Die Ein- u. Ausgaben können auch mehrfach verkettet werden.

Beispiel zu verketteten Ein- u. Ausgaben :

```
cin >> x1 >> x2 ;
cout << endl << " Ergebnis = " << 1/(1+x1*x2) << endl;
```

Formatsteuerung

Ausgabebreite und Anzahl der Nachkommastellen können über Methoden und Manipulatoren der iostream-Klasse gesteuert werden.

Ausgabeweite : ***cout.width (Wert);***
Anzahl der Nachkommastellen : ***cout.precision (Wert);***

Beispiel :

```
cout << " Ausgabebreite 8 Zeichen mit 4 Nachkommastellen : " << endl;
cout.setf(ios::fixed);
cout.width(8); cout.precision(4);
cout << " Die Zahl ist = " << x << endl;
```

Ausgabeweite mit setw : ***setw (Wert);***

Dieser Manipulator kann direkt in den Ausgabestrom eingefügt werden, die erforderliche Headerdatei für setw und weitere Manipulatoren ist **<iomanip>** .

```
cout << " formatierte Ausgabe als Tabelle : " << endl;
cout << setw(4) << i << setw(10) << x << endl;
```

Ein- Ausgaben in C :

Einlesen von Tastatur : ***scanf (Formatstring, Variable);***
Ausgabe auf Bildschirm : ***printf (Formatstring, [Variable],...);***

erforderliche Headerdatei : **<stdio.h>**

Beispiel :

```
#include <stdio.h>
int main ()
{   int i;
    printf (" Bitte geben sie eine Zahl ein ");
    scanf ("%d",&i);
    printf(" i = %4d  \n",i );
}
```

bei der Funktion scanf muss die Adresse der Variablen x mit **&i** als Parameter eingesetzt werden
(& ist der Adressoperator vor der Variablen und wird mit „Adresse“ ausgesprochen)
"\\n" ist das Steuerzeichen für einen Zeilenvorschub (= line feed)

der String "%d" ist der **Formatstring** zur Formatangabe :

- %d** ist der Formatstring für Integer (int) Zahlen
- %4d** für die Ausgabebreite 4 einer Integer Zahl
- %f** ist der Formatstring für Gleitkomma (float) Zahlen
- %6.4f** für die Ausgabe einer Gleitkomma (float) Zahl
mit der Ausgabebreite 6 und mit 4 Nachkommastellen
- %c** ist der Formatstring für Zeichen (char)

über die Formatstring-Angabe können auch verkettete Ein- u. Ausgaben erfolgen

Beispiel zu verketteten Ein- u. Ausgaben :

```
scanf ("%f%f",&x1,&x2);
printf(" %d -ter Wert : %6.4f  %6.4f \n", i,x1,x2);
```

Beispiel : Addition zweier Zahlen

```
#include <stdio.h> // für Ein- u. Ausgabe-Funktionen
#include <conio.c> // für Bildschirmsteuerungs-Funktionen
```

```
void main()
{   int  x1,x2,y; // Variablendefinitionen

    clrscr(); // Bildschirm löschen
    gotoxy(30,5); // Cursor positionieren
    printf(" A D D I T I O N "); // Bildschirm Ausgabe
    gotoxy(5,10); printf("x1="); // Ausgabe der Eingabe-Anforderung
    scanf ("%d",&x1); // Einlesen in die Variable x1
    gotoxy(5,11); printf("x2="); // Ausgabe der Eingabe-Anforderung
    scanf ("%d",&x2); // Einlesen in die Variable x2
    y = x1 + x2; // Addition ausführen
    gotoxy(5,12);
    printf("Summe= %6.2f",y); // formatierte Ausgabe
    getch(); // Programm anhalten bis Tastendruck
}
```

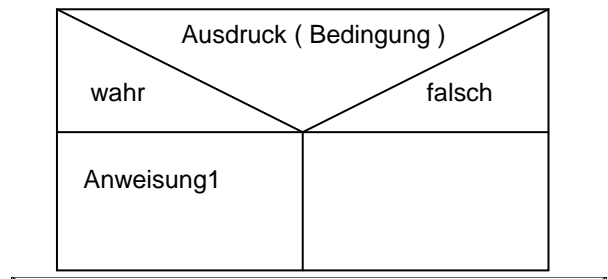
Auswahl (Verzweigung)

Programmverzweigungen können mit der if - Anweisung für Einfach-Verzweigungen oder mit der switch - Anweisung für mehrere Fallunterscheidungen erfolgen.

if - Anweisung :

```
if ( Ausdruck )  
    Anweisung1;
```

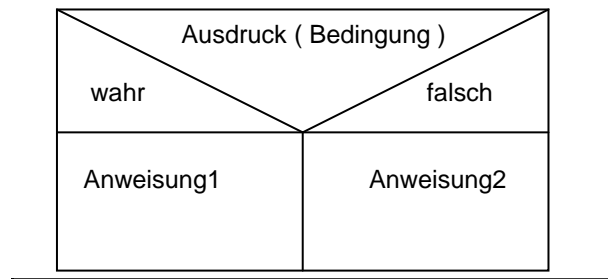
Struktogramm



if-else-Anweisung :

```
if ( Ausdruck )  
    Anweisung1;  
else  
    Anweisung2;
```

Struktogramm



if – Anweisung : Wenn der Ausdruck in den if-Klammern logisch wahr ist, dann wird die folgende Anweisung (Anweisung1) ausgeführt, ansonsten wird sie übersprungen.
if- else – Anweisung : Wenn der Ausdruck in den if-Klammern logisch wahr ist, dann wird die folgende Anweisung (Anweisung1) ausgeführt, wenn sie falsch ist, dann wird die Anweisung vom else-Zweig (Anweisung2) ausgeführt.

Sollen mehrere Anweisungen im if- oder else- Zweig ausgeführt werden, so kann das über einen Anweisungsblock erfolgen :

```
if ( Ausdruck )  
{ Anweisung;  
  Anweisung; ...  
}
```

Bedingungen können mit Hilfe von Vergleichsoperatoren und logischen Operatoren erstellt werden.

Vergleichsoperatoren :

==	Gleichheit
!=	Ungleichheit
> (>=)	größer (gleich)
< (<=)	kleiner (gleich)

logische Operatoren :

&&	logisch Und
	logisch Oder
!	logisch Nicht

Beispiel 1 :

```
if (x == 0 ) cout << " x ist gleich 0  ";
if (x != 0 ) cout << " x ist ungleich 0 ";
if (x < 0 ) cout << " x ist negativ  ";
if (x >= 0 ) cout << " x ist positiv  ";
```

Beispiel 2 : 1/0 - Fehler abfangen

```
if (x != 0) // wenn x ungleich Null ,
    y = 1/x; // dann ist y = 1/x
else y = ymax; // andernfalls ist y = ymax
```

Beispiel 3 : Bedingungen mit logischen Operatoren

```
if (x >= -1.0 && x < 1.0 ) y = 0; // wenn x innerhalb -1 bis +1
if (x < -1.0 || x >= 1.0 ) y = 1; // wenn x außerhalb -1 bis +1
```

```
/* alternativ kann die zweite if - Anweisung auch durch Negation
   der ersten Bedingung formuliert werden : */
```

```
if (!(x>= -1.0 && x< 1.0)) y = 1; // wenn x außerhalb -1 bis +1
```

Beispiel 4 : if mit Anweisungsblock

```
if ( y > 25 )
{
    y = 1 ;
    gotoxy(5,y);
    cout << '|' ;
}
```

Beispiel 5 : geschachtelte if - Anweisungen

```
if ( x>0 )
{
    if ( y>0 ) cout << " x und y größer Null " ;
    else      cout << " x größer und y kleiner Null " ;
}
else
{
    if ( y>0 ) cout << " x kleiner und y größer Null " ;
    else      cout << " x und y kleiner Null " ;
}
```

Ein ganzzahliger C/C++ - **Ausdruck hat auch ohne Bedingung** einen logischen Wert :

Ausdruck **gleich 0** ergibt falsch

Ausdruck **ungleich 0**. ergibt wahr

Beispiel 6 : if - Anweisung ohne Bedingung

```
if (i) cout << "Zahl ungleich Null";
else   cout << "Zahl gleich Null";
```

Beispiel 7 : 1/0 - Fehler abfangen wie Bsp 2.

```
if (i) y = 1/i;
else   y = ymax;
```

! häufige Fehler mit der if-Bedingung :

```
if (i=0) ... // ist immer falsch !!! , daher nutzlos
if (i==0) ... // korrekt bei Vergleich mit 0
if (i=1) ... // ist immer wahr !!! , daher nutzlos
if (x=y) ... // weist x den Wert von y zu und ist wahr ,
              // wenn y ungleich Null ist !
if (x==y) ... // ist hingegen der Vergleich auf Gleichheit !
```


switch - Anweisung :

Struktogramm

```
switch ( Ausdruck )
{ case Wert1 : Anweisung1;break;
  case Wert2 : Anweisung2;break;
  ...
  default   : Anweisung0;
}
```

Ausdruck			
Wert1	Wert2	... default	
Anw.1	Anw.2		Anw.0

Die switch - Anweisung erlaubt die Auswahl aus mehreren Fällen :

ist der Wert des switch - Ausdruckes gleich Wert1, dann wird die Anweisung1 ausgeführt,

ist er gleich Wert2, dann wird die Anweisung2 ausgeführt, usw.

Die default - Anweisung wird immer dann ausgeführt, wenn keiner der Fälle erfüllt ist.

Die **break** - Anweisung ist notwendig, um den switch - Block zu verlassen, wird sie nicht ausgeführt, dann werden auch alle weiteren Anweisungen des case - Blockes abgearbeitet.

(switch arbeitet als Programmschalter oder Einsprunganweisung !)

Die switch - Anweisung ist bei Mehrfachauswahlen übersichtlicher als die if - Anweisung, kann jedoch nur mit ganzzahligen Ausdrücken (Datentyp int, char, long, ...) verwendet werden.

Beispiel 1 :

```
int note ;
switch (note)
{ case 1 : cout << " Sehr gut      ";break;
  case 2 : cout << " Gut          ";break;
  case 3 : cout << " Befriedigend  ";break;
  case 4 : cout << " Genügend     ";break;
  case 5 : cout << " Nicht genügend ";break;
  default: cout << " falsche Eingabe ";
}
```

Beispiel 2 :

```
switch (note)
{ case 1: case 2:
  case 3: case 4: cout << " positiv ";break;
  case 5:        cout << " negativ ";break;
}
```

Beispiel 3 : einfache Menü - Verzweigung

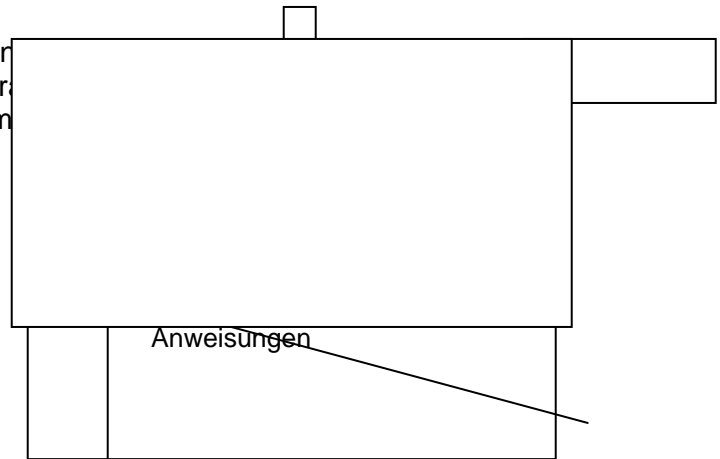
```
char menu;
cout << " Menueauswahl [+],[-],[*],[/] ";
cin << menu;
switch (menu)
{ case '+' : y = a+b ; break;
  case '-' : y = a-b ; break;
  case '*' : y = a*b ; break;
  case '/' : y = a/b ; break;
}
cout << " Ergebnis = " << y ;
```

Schleifen (Wiederholungen)

Schleifen dienen zur wiederholten Ausführung von Anweisungen.
In C/C++ stehen 3 unterschiedliche Formen für Programmierer zur Verfügung.
Alle C/C++ - Schleifen haben gemeinsam, dass sie mindestens einmal durchlaufen werden.

while - Anweisung :

```
while ( Laufbedingung )
{ Anweisung ;
  ...
}
```



Die while – Schleife hat die Wirkung, dass die Anweisungen innerhalb des Schleifenblockes wiederholt ausgeführt werden, solange die Laufbedingung erfüllt ist.

Die Laufbedingung wird bei der while-Schleife am Anfang vor dem Schleifenblock abgefragt.

Beispiel 1 :

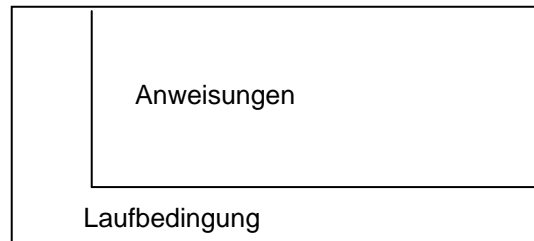
```
int i=1;           // Anfangswert für Laufvariable
while ( i<=10 )    // Laufbedingung
{ cout << i;       // Laufvariable i ausgeben
  i=i+1 ;          // Laufvariable erhöhen
}
```

im Beispiel 1 werden durch die Schleifenanweisung alle Zahlen von 1 bis 10 ausgegeben.

do - while - Anweisung :

```
do
{ Anweisung ;
  ...
} while ( Laufbedingung )
```

Struktogramm



die do - while Schleife hat die Laufbedingung am Ende des Schleifenblocks und wird damit zumindest einmal durchlaufen .

Beispiel 2 :

```
do           // Programmwiederholung bis ESC
{           // Programmteil
    ...

    printf(" Programm Ende mit [Esc]      ");
    antwort=getch();           // Tasteneingabe abfragen
} while ( antwort != 27 )      // Laufbedingung
```

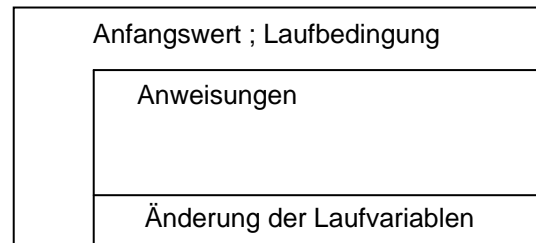
for - Anweisung :

Struktogramm

```

for ( Anfangswert ; Bedingung ; Änderung )
{ Anweisung ;
  ...
}

```



die for - Schleife hat die übersichtlichste Form für die Angabe von Anfangswert, Laufbedingung und Änderung der Laufvariablen und sollte deshalb immer bevorzugt verwendet werden, wenn nicht die Notwendigkeit für eine andere Schleifenform besteht.

Beispiel 3:

```

for ( i=1;i<=10;i=i+1 )    // Anfangswert; Bedingung; Änderung
{ cout << setw(3) << i;    // Laufvariable i ausgeben
}

```

Im for – Kopf können auch **weitere Anweisungen** stehen, die mit Beistrich getrennt werden.

Beispiel 4:

```

for ( i=1,j=10; i<=10; i=i+1,j=j-1 )
{ cout << i << j ;
}

```

im Beispiel 4 werden für die Variable i die Werte von 1 bis 10 hochgezählt und für die Variable j die Werte von 10 bis 1 heruntergezählt.

Schleifenabbruch - Anweisungen :

break Abbruch der Schleife
continue Sprung zum Schleifenende

Beispiel 5 : Schleife mit Abbruchbedingung

```

for(float x=-10; x<10; x=x+0.1)
{ if ( x>0 ) break ;      // Abbruchbedingung
}

```

Endlos - Schleifen :

Endlos - Schleifen sind Schleifen, die nie mehr verlassen werden und damit ein Programm blockieren. Sie ergeben sich mitunter unerwünscht durch eine falsch formulierte Laufbedingung oder werden auch beabsichtigt im Zusammenhang mit Abbruch-Anweisungen programmiert.

Beispiel 6 : Endlosschleife mit Abbruchbedingung

```

while (1)                      // Endlos-Schleife mit while
{ c=getch();
  if (c == '\13') break ;      // Abbruch mit Enter Taste
  putch(c);
}

```

Beispiel 7 : Endlosschleife mit for

```

for(;;)                        // Endlos-Schleife mit for
{ ...
}

```

Zuweisungsoperatoren += -= :

Die Änderung der Laufvariablen kann auch mit den Zuweisungsoperationen +=, -= erfolgen.

i += 1 entspricht i = i + 1
i -= 2 entspricht i = i - 2 usw.

Inkrement- und Dekrement- Operatoren ++, -- :

Das Erhöhen und Verringern von ganzzahligen Variablen um Eins kann über den Inkrement (++) und Dekrement (--) - Operator vereinfacht ausgeführt werden.

++ Inkrement = Erhöhung um 1
-- Dekrement = Verringerung um 1

i++ und ++i ersetzt somit i=i+1
i-- und --i ersetzt somit i=i-1

Die Position der Operationszeichen ++, -- hat den Effekt, dass in einem Berechnungsausdruck entweder mit dem bereits erhöhten Wert oder mit dem noch unveränderten Wert gerechnet wird.

```
    x = 1;
// Erhöhung mit ++x  -> es wird zuerst x erhöht und dann y berechnet
    y = ++x*2;        // ersetzt x=x+1;y=x*2; und ergibt y=4

// Erhöhung mit x++  -> es wird zuerst y berechnet und dann x erhöht
    y = x++*2;        // ersetzt y=x*2;x=x+1; und ergibt y=2
```

Beispiel 7: Schleife mit abnehmender Laufvariable

```
int i;
for ( i=10;i>0;i-=1 )          // i läuft von 10 bis 1 herunter
{   cout << i ;
}
```

Beispiel 8: alle geraden Zahlen ausgeben

```
for ( i=0;i<10;i+=2 )          // i wird jeweils um 2 erhöht
{   cout << i ;
}
```

Beispiel 9: Minimalform für n-Schleifendurchläufe

```
while ( n-- )                  // von n-1 bis 0 herunter
{   cout << n ;
}
```

Beispiel 10: Berechnung von Summe und Mittelwert

```
int sum=0;                     // Summenanfangswert = 0
for ( i=1;i<=n;i++ )          // Schleife von 1 bis n
{   sum += i ;                 // Summenalgorithmus sum = sum + wert
}
cout << " Summe von 1 bis n = " << sum ;
cout << " Mittelwert = Summe/Anzahl = " << (float) sum/n ;
```

geschachtelte Schleifen :

Schleifen können auch ineinander verschachtelt ausgeführt werden.

Beispiel 11: eingeschachtelte for-Schleifen

```
int i,j;
for ( j=0;j<10;j++ )           // Hauptschleife
{   for ( i=0 ; i<10 ; i++ )    // Unterschleife
    {   cout<<j<<i<<" ";
        }
    cout << endl;
}
```

Beispiel 12: eingeschachtelte while-Schleifen

```
int j=0;
while ( j<10 )                 // Hauptschleife
{   i=0;
    while ( i<10 )             // Unterschleife
    {   cout<<j<<i<<" ";
        i++;
    }
    j++;
    cout << endl;
}
```

Beispiel 13: Einmaleins von 1 bis 10

```
for ( j=1;j<=10;j++ )         // j-Schleife
{   for ( i=1 ; i<=10 ; i++ )  // i-Schleife
    {   cout<<setw(4)<<j*i;
        }
    cout << endl;
}
```

Berechnung des größten gemeinsamen Teilers (ggT) :

Beispiel 14: größter gemeinsamer Teiler (ggT) zweier Zahlen

```
int a,b;
cout << " 1.Zahl ="; cin >> a;
cout << " 2.Zahl ="; cin >> b;
cout << " ggT von " << a << " und " << b << " = ";

while ( b>0 )                 // wiederhole solange b größer 0
{   if (b>a)                  // wenn b größer a
    {   hilf=a; a=b; b=hilf;  // dann tausche a u. b
    }
    a=a-b;                    // neuer Teiler
}
cout << a << endl;            // ggT ausgeben
```

der Berechnungsalgorithmus des größten gemeinsamen Teilers (ggT) erfolgt nach Euklid.

Datentypen

Jede Variable in C/C++ wird mit einem Datentyp definiert. Der Typ einer Variablen bestimmt, welche Werte in einer Variablen gespeichert werden können. Der Datentyp gibt auch an, wieviel Speicherplatz für eine Variable verwendet wird und bei der Variablendefinition reserviert wird.

Bei der Verwendung von Variablen wird bereits vom Compiler überprüft, ob die Typen passen und eventuell eine automatische Typumformung oder andernfalls eine Fehlermeldung erfolgt.

Es gibt **einfache Datentypen (Grunddatentypen)** und **abgeleitete Datentypen**.

Abgeleitete Datentypen (wie Felder, Strukturen und Klassen) werden aus den Grunddatentypen hergeleitet und können als benutzdefinierte Datentypen erstellt werden.

Übersicht über die einfachen Datentypen :

Die angegebenen Werte gelten für 32-Bit-Compiler und sind implementierungsabhängig!

Typ	Speichergröße	Zahlenbereich
char	1 Byte	-128 bis 127
short int	2 Bytes	-32768 bis 32767
int	4 Bytes	$\sim -2 \cdot 10^9$ bis $\sim 2 \cdot 10^9$
long int	4 Bytes	$\sim -2 \cdot 10^9$ bis $\sim 2 \cdot 10^9$
float	4 Bytes	$\sim 10^{-38}$ bis $\sim 10^{+38}$
double	8 Bytes	$\sim 10^{-308}$ bis $\sim 10^{+308}$
long double	10 Bytes	$\sim 10^{-4932}$ bis $\sim 10^{+4932}$

Modifizierer : **unsigned** für vorzeichenlose ganzzahlige Datentypen

short für kleinen Zahlenbereich

long für großen Zahlenbereich

Beispiel 1:

```
unsigned char    ch;        // vorzeichenlose char Variable
unsigned int     ui;        // vorzeichenlose int Variable
long int         x;         // long Variable
double          d;         // double für doppelte Genauigkeit
```

Die **Größe eines Datentyps** oder einer Variablen kann mit dem Operator **sizeof** bestimmt werden.

Beispiel 2:

```
cout << sizeof(int);        // zeigt die Typgröße in Bytes
cout << sizeof(float);
```

Die **Wertegrenzen eines Datentyps** können über die vordefinierten Konstanten der Headerdatei „limits.h“ abgefragt werden.

Beispiel 3:

```
cout << "Integer-Bereich" << INT_MIN << "..." << INT_MAX ;
cout << "Long-Bereich" << LONG_MIN << "..." << LONG_MAX ;
```

oder über den Funktionsaufruf „numeric_limits<Datentyp>“ der C++ -Bibliothek „limits“ :

Beispiel 4:

```
cout << numeric_limits<int>::min() << "..." << numeric_limits<int>::max();
```

Format String zu den Datentypen für Ein-Ausgabefunktionen

allgemeine Form : **% [flags] [width] [.prec] [mod] Typzeichen**

Typzeichen

für Zahlen :

d	Integer-Wert mit Vorzeichen
i	Integer-Wert mit Vorzeichen
u	Integer-Wert ohne Vorzeichen (unsigned)
o	oktaler Integer-Wert
x	hexadezimaler Integer-Wert (0,1, ..e,f)
X	hexadezimal mit Großbuchstaben (0,1, .. E,F)
f	Gleitkomma-Darstellung 1234.5678
e	Gleitkomma mit „e“ als Exponent 1.2e+3
g	Gleitkomma der Form e oder f
E	Gleitkomma mit „E“ als Exponent 1.2Ee+3

für Zeichen :

c	für ein Zeichen
s	für eine Zeichenkette (String)

flags Steuerzeichen für Ausrichtung

-	Ausrichtung links (standardmäßig rechts)
+	bewirkt die Angabe eines pos. Vorzeichens
blank	positive Zahlen werden ohne + Zeichen ausgegeben

width Feldbreite für die Mindestzahl von Zeichen

n	mindestens n Zeichen ausgegeben
0n	mit führenden Nullen ausgegeben

.prec Genauigkeitsangabe

0	Ausgabe ohne Dezimalpunkt
n	Nachkommastellen

mod Konvertierungsangabe

l	für long int
L	für long double

Beispiel 4: Datentypen und ihre Formatstrings

```
char c;
int i;
unsigned int ui;
long l;
float x;
double d;

printf("\n char - Zeichen           : %c",c='a');
printf("\n int-Zahl                   : %d",i=12345);
printf("\n int-Zahl hexadezimal          : %x",i);
printf("\n Zahl linksbündig              : %-4d",i);
printf("\n unsigned int-Zahl             : %u",ui=56789U);
printf("\n long-Zahl                     : %ld",l=123456L);
printf("\n float-Zahl                    : %f",x=0.12345f);
printf("\n Zahl in Exponentialform        : %8.5e",x);
printf("\n double-Zahl                   : %1.10lf",d=0.123456789L);
```

Datentyp integer

Für die Verarbeitung von ganzen Zahlen steht der Datentyp „int“ zur Verfügung.

Die Speicherplatzgröße für „int“ ist implementierungsabhängig und kann entweder 2 oder 4 Bytes betragen und der damit verbundene Wertebereich kann daher unterschiedlich sein!

Die Typen „short int“ und „long int“ sind speziell für kleine bzw. große Zahlenbereiche, die Typangabe kann auch kurz mit „short“ und „long“ erfolgen.

Die Angabe „unsigned“ steht für vorzeichenlose ganze Zahlen.

Darstellung von ganzen Zahlen :

dezimal : beginnen mit Ziffer oder Vorzeichen, keine führende Nullen (wie 12, -1, ...)

oktal : beginnen mit 0 (wie 037, 030, ...)

hexadezimal : beginnen mit 0x oder 0X (wie 0x1F, 0xA1, ...)

Durch eine Endung (Suffix) kann der Datentyp für eine Zahl vorgegeben werden :

l oder L für long (wie 123L, 456l, ...)

u oder U für unsigned (wie 789U, 123u, 456LU, ...)

Operatoren

arithmetische Operatoren :

+ - * / Addition , Subtraktion , Multiplikation , Division

% ganzzahliger Rest (Modulo)

++ Inkrement (Erhöhung um Eins)

-- Dekrement (Verringerung um Eins)

diese Operatoren können auch in Kombination mit einer Zuweisung (=) ausgeführt werden :

+= x += i entspricht x = x + i

-= x -= i entspricht x = x - i

*= x *= i entspricht x = x * i

/= x /= i entspricht x = x / i

%= x %= i entspricht x = x % i

Vergleichsoperatoren :

== gleich

!= ungleich

< kleiner

> größer

<= kleiner gleich

>= größer gleich

logische Operatoren :

&& logisch Und

|| logisch Oder

! logisch Nicht

bitweise Operatoren :

<< Shift left (Bits nach links verschieben)

>> Shift right (Bits nach rechts verschieben)

& bitweise AND (alle einzelnen Bits werden UND verknüpft)

| bitweise OR

^ bitweise EXOR

~ bitweise NOT

Bitweise Operatoren verknüpfen alle einzelnen Bits des einen Operanden mit dem anderen.

Logische Operatoren (&& , || , !) hingegen verarbeiten nur die Werte True (!=0) und False (==0) korrekt.

Beispiel 1: Darstellung dezimal, hexadezimal, oktal

```
int i;
cout << endl << " Darstellung dezimal,hexadezimal,oktal :";
cin >> i;
printf("\n Ausgabe über printf ");
printf("\n dezimal=%d hexadezimal=%x oktal=%o ",i,i,i);

cout << endl << « " Ausgabe über cout "<< endl;
cout.setf(ios::dec); cout << " dezimal="<< i; cout.unsetf(ios::dec);
cout.setf(ios::hex); cout << " hexadezimal="<< i;cout.unsetf(ios::hex);
cout.setf(ios::oct); cout << " oktal=" << i; cout.unsetf(ios::oct);
```

Beispiel 2: ganzzahlige Division mit Rest

```
int z,n ;
cout << " Zaehler = "; cin >> z;
cout << " Nenner   = "; cin >> n;
d = z/n; // ganzzahlige Division
r = z%n; // ganzzahliger Rest
cout << endl << " Division = " << d;
cout << endl << " Rest      = " << r;
```

Beispiel 3: Prüfung auf gerade/ungerade Zahl

```
int i;
cout << " Zahl = "; cin >> i;
if ( i%2 == 0 ) cout << " gerade Zahl ";
else cout << " ungerade Zahl ";
```

Beispiel 4: Umrechnung von binär auf dezimal

```
int i,b0,b1,b2,b3,b4,b5,b6,b7;
cout << endl << " Binärzahl = " << endl;
cin >>b7>>b6>>b5>>b4>>b3>>b2>>b1>>b0; // Einlesen einer 8-Bitzahl
i = b7*128 + b6*64 + b5*32 + b4*16 + b3*8 + b2*4 + b1*2 + b0;
cout << endl << " Dezimalzahl = " << i;
```

Beispiel 5: Umrechnung von dezimal auf binär

```
cout << endl << " Dezimalzahl = "; cin >> i;
b0=i%2; i=i/2; // Umrechnung auf binär
b1=i%2; i=i/2;
b2=i%2; i=i/2;
b3=i%2; i=i/2;
b4=i%2; i=i/2;
b5=i%2; i=i/2;
b6=i%2; i=i/2;
b7=i%2;
cout << endl << " Binärzahl = " <<b7<<b6<<b5<<b4<<b3<<b2<<b1<<b0;
```

Bitverarbeitung :

Über die bitweisen Operatoren kann eine Bitverarbeitung ausgeführt werden.

Die Betrachtung dieser Operationen kann nur über die Bitdarstellungen der Variablen erfolgen.

Wichtig dabei ist auch die Kenntnis der internen Bit-Darstellung der Datentypen.

Für die Bitverarbeitung eignen sich vor allem vorzeichenlose Datentypen, wie unsigned int.

```
Bit Darstellung int   i = 1      : 00000000000000000000000000000001
                     i = 0      : 00000000000000000000000000000000
                     i = -1     : 11111111111111111111111111111111
Bit Darstellung unsigned int
                     ui = 1     : 00000000000000000000000000000001
                     ui = 65535 : 00000000000000001111111111111111
```

Beispiel : bitweise Operatoren

```
// Verschieben nach rechts
b = a >> 2;      // Bits bei a = 9 : 0000000000001001
                 // Ergebnis b = 2 : 0000000000000010

// bitweise UND
c = a & 5;        // Bits bei a = 9 : 0000000000001001
                 //           & 5 : 0000000000000101
                 // ergibt c =   : 0000000000000001

// bitweise ODER
c = a | 5;        // Bits bei a = 9 : 0000000000001001
                 //           | 5 : 0000000000000101
                 // ergibt c =   : 0000000000001101

// bitweise NOT
c = ~a;           // Bits bei a = 9 : 0000000000001001
                 //           ~a : 1111111111110110
```

Bitmaskierung :

unter Bitmaskierung versteht man das Herausfiltern einzelner Bits mit bitweise UND (&).

```
Schema :   b       = b7 b6 b5 b4 b3 b2 b1 b0
           Maske    = 0  0  0  0  1  1  1  1
           -----
           b & Maske = 0  0  0  0  b3 b2 b1 b0
```

Beispiel : Bitdarstellung eines Dezimalwertes

```
int i,b,d;
printf("\n Eingabe Dezimalwert = ");
scanf("%d",&d);
printf("\n Ausgabe Binärwert   = ");
for ( i=15;i>=0;i--) // von der 15-ten bis zur 0-ten Stelle
{ b=d>>i;             // um i-Stellen nach rechts schieben
  b=b&1;              // niedrigstes Bit herausfiltern
  printf("%1d",b);
}
```

Zufallszahlen (random number) :

Die Funktion „**rand()**“ der Bibliothek „**stdio.h**“ erzeugt ganzzahlige Zufallszahlen.

Die Zufallswerte liegen im Bereich von 0 bis **RAND_MAX**.

Die Anfangsbedingung für die Zufallsfolge kann über die Funktion **srand(seed)** dadurch am besten gestreut werden, indem der Zeitwert über die Funktion **time()** als Saat (seed) eingesetzt wird.

Beispiel : Zufallszahlen erzeugen

```
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;

int main()
{
    int i,r,n=5;
    float x;

    srand(time(NULL));          // Zufallsfolge initialisieren

    cout << endl << " Zufallszahlen  " << endl;
    cout << endl << " im Zahlenbereich von 0 bis "<<RAND_MAX<<" : ";
    for (i=0;i<n;i++)
    {   r=rand();                // Zufallszahlen von 0 bis RAND_MAX
        cout << " " << r;

    cout << endl << " von 0 bis 99  : ";
    for (i=0;i<n;i++)
    {   r=rand()%100;            // Zufallszahlen von 0 bis 99
        cout << " " << r;
    }

    cout << endl << " würfeln von 1 bis 6 : ";
    for (i=0;i<n;i++)
    {   r=rand()%6 + 1 ;         // Zufallszahlen von 1 bis 6
        cout << " " << r;
    }

    cout << endl << " von 0 bis 1 : ";
    for (i=0;i<n;i++)
    {   x=(float) rand()/RAND_MAX; // Gleitkomma-Zufallszahlen von 0 bis 1
        printf(" %1.6f",x);
    }

    system("PAUSE");
    return 0;
}
```

Datentyp float

Für die Verarbeitung von Gleitkommazahlen stehen die Datentypen „float“, „double“ und „long double“ zur Verfügung.

Float -Variable weisen eine Genauigkeit von ca. 7 Dezimalstellen auf,

Double-Variable haben die doppelte Genauigkeit mit ca. 15 Dezimalstellen,

Long double-Variable ca. 19 Stellen Genauigkeit und einen Zahlenbereich von ca. $10^{\pm 5000}$.

Darstellung von Gleitkommazahlen :

Gleitkommazahlen werden mit Mantisse und Exponent dargestellt, wobei der Exponent mit der Angabe 'E' oder 'e' gekennzeichnet wird.

wie 1.23E-2, -1.23e+1, ...

Durch eine Endung (Suffix) kann der Typ der Gleitkommazahl vorgegeben werden :

f oder F für float (wie 1.23f, 1E-3F, ...)

l oder L für long double (wie 1.23l, 1e-3L, ...)

Operatoren

für Gleitkommavariablen gelten auch die gleichen Operatoren wie für den Datentyp int, außer den logischen und bitweisen Operatoren und den Operatoren % ,++ , -- .

Bibliotheksfunktionen in <math.h> :

Viele mathematische Funktionen stehen in der C-Bibliothek <math.h> oder der C++-Bibliothek <cmath> zur Verfügung. Diese Funktionen sind mit dem Datentyp **double** definiert, können jedoch über die automatische Typumformung auch direkt mit float-Variablen aufgerufen werden.

Alle trigonometrischen Funktionen (sin, cos, ...) rechnen im Bogenmaß (Radiant), die Umrechnung auf Radiant oder Grad erfolgt mit folgenden Formeln :

$$\text{rad} / 2\pi = \text{grad} / 360$$

$$\rightarrow \text{rad} = \text{grad} * \pi / 180$$

$$\rightarrow \text{grad} = \text{rad} * 180 / \pi$$

Funktionen in <math.h>:

abs(i)	Absolutbetrag der Integerzahl i
fabs(x)	Absolutbetrag der Gleitkommazahl x
floor(x)	größte ganze Zahl kleiner gleich x
ceil(x)	kleinste ganze Zahl größer gleich x
sqrt(x)	Quadratwurzel von x
sin(x)	Sinusfunktion
cos(x)	Cosinusfunktion
tan(x)	Tangensfunktion
asin(x)	Arcussinus
acos(x)	Arcuscosinus
atan(x)	Arcustangens
atan2(y,x)	Arcustangens mit Angabe von y/x (Ergebnis von $-\pi$ bis $+\pi$)
exp(x)	Exponentialfunktion e^x
log(x)	natürlicher Logarithmus $\ln(x)$
log10(x)	Logarithmus zur Basis 10
pow(x,y)	Potenzfunktion x^y
fmod(x,y)	x Modulo y (Rest von x/y)
...	

Konstanten in <math.h>:

M_PI	3.14159265358979323846	π
M_1_PI	0.31830988618379067154	$1/\pi$
M_SQRT2	1.41421356237309504880	$\sqrt{2}$
M_E	2.7182818284590452354	e (Eulerzahl)
...		

Beispiel 1 : Wurzelberechnung mit sqrt(x)

```
float a,b,c;
c = sqrt( a*a + b*b );           // Pythagoras  $c = \sqrt{a^2+b^2}$ 
```

Beispiel 2: Sinusberechnung mit sin(x);

```
double grad,rad,y;
for (grad=0;grad<=360;grad+=30)// Sinus von 0 bis 360 Grad
{   rad = grad*M_PI/180;           // Umrechnung von grad auf rad
    y = sin(rad);                  // Sinus Funktionsaufruf
}
```

Beispiel 3: Potenzfunktion x^n mit pow(x,n)

```
float x ;double y;
y = pow(x,4);                     //  $y = x^4$ 
y = pow(x,-2);                    //  $y = x^{-2}$ 
y = pow(2,1.0/12);                //  $y = 2^{1/12}$  (chromatischer Halbton)
```

Beispiel 4: Umrechnung von kartesischen Koord. auf Polarkoordinaten

```
/*Die Funktion atan(x)liefert Ergebnisse im Bereich von -pi/2 bis pi/2.
Die Funktion atan2(y,x) mit dem Ergebnisbereich von -pi bis pi liefert
hingegen in allen 4 Quadranten korrekte Ergebnisse.
*/
float phi,x,y,z;
z = sqrt( x*x + y*y );           // Betrag
phi = atan2(y,x);                // Winkel
// phi = atan(y/x);liefert nur im 1.u.4. Quadranten richtige Ergebnisse!
```

Beispiel 5: Lösung der quadratischen Gleichung

```
// quadratische Gleichung :  $x^2 + p \cdot x + q = 0$ 
// Lösung :  $x_1 = -p/2 + \sqrt{p^2/4-q}$  und  $x_2 = -p/2 - \sqrt{p^2/4-q}$ 
#include <stdio.h>
#include <iostream>
#include <cmath>
using namespace std;

int main()
{   double x1,x2,p,q,d,y;

    cout << endl << " Quadratische Gleichung  $x^2 + p \cdot x + q = 0$  " << endl;
    cout << endl << " Koeffizient p = "; cin >> p;
    cout << endl << " Koeffizient q = "; cin >> q;

    d = p*p/4-q;
    cout << endl << " Diskriminante = " << d;
    cout << endl << " Loesung : ";
    if (d>=0)
    {   x1 = -p/2 + sqrt(p*p/4-q);
        x2 = -p/2 - sqrt(p*p/4-q);
        cout << " x1= " << x1 << " x2= " << x2;
    }
    else
    {   cout << " komplexe Lösung ";
    }
    system("PAUSE");
    return 0;
}
```

Beispiel 6: Funktionstabelle

```
////////////////////////////////////
// Darstellung einer Funktions-Tabelle (mit Bildschirmsteuerung)
////////////////////////////////////

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

const char ESC=27;                // Konstante für Esc-Wert
float x,x1,x2,xi,y;               // globale Variable

int main()
{ char antwort;
  int i,ok;

  do
  { textcolor(BLUE);               // Textfarbe einstellen
    textbackground(LIGHTGRAY);     // Hintergrund einstellen
    clrscr();
    gotoxy(25,2);printf ("  FUNKTIONS - TABELLE  \n");
    gotoxy(25,3);printf ("  -----  \n");
    textcolor(BLUE);
    do
    { gotoxy(10,6);delline();printf (" Anfangswert x1 = ");
      scanf("%f",&x1);              // Berechnungsgrenzen einlesen
      gotoxy(10,7);delline();printf (" Endwert      x2 = ");
      scanf("%f",&x2);
      gotoxy(10,8);delline();printf (" Wertzuwachs xi = ");
      scanf("%f",&xi);              // Werterhöhung einlesen

      ok = (x2>x1);                // Eingabe-Überprüfungen !
      ok = ok && (xi>0);
      ok = ok && ((x2-x1)>=xi);
    } while(!ok);

    clrscr();                     // Tabellenkopf ausgeben
    gotoxy(10,2);printf("      x          y=sin(x)/x      ");
    gotoxy(10,3);printf(" -----  ");
    i=1;
    for(x=x1;x<=x2;x+=xi)
    { if (x==0) y=1;               // y=1 bei x=0 !
      else y=sin(x)/x;            // y=sin(x)/x berechnen

      gotoxy(10,4+i);
      printf("    %8.4f    %8.4f    ",x,y); // Tabellenwerte ausgeben

      if (i==20)                  // seitenweise Darstellung
      { i=0; getch();
        }
      i++;
    }
    textcolor(MAGENTA);gotoxy(25,25);printf(" Programm Ende mit [Esc] ");
    antwort=getch();
  } while (antwort!=ESC);         // Programmende bei ESC-Taste

  textcolor(WHITE);textbackground(BLACK); // Farben rücksetzen
  clrscr();
  return 0;
}
```

Datentyp char

Variable vom Typ char werden intern über 1 Byte dargestellt und können alle ASCII-Zeichen und Zeichen des erweiterten Zeichensatzes aufnehmen.

Zeichenkonstante werden durch Hochkommata eingeschlossen angegeben (wie 'a', 'A', ...), können aber auch über den Ordinalwert oder über Escape-Sequenzen beschrieben werden.

Beispiel :

```
char c;
c = 'A';           // Zeichenkonstante
c = 65;            // Zeichenkonstante über den Ordinalwert
c = 0x41;          // Zeichenkonstante über Ordinalwert als Hexadezimalzahl
c = '\n';          // Escape-Sequenz für Line Feed
```

Escape – Sequenzen :

\n	LF	Line Feed (Zeilenvorschub)
\r	CR	Carriage Return (Wagenrücklauf)
\v	VT	Vertical Tabulator
\b	BS	Backspace (Rückwärtsschritt)
\f	FF	Form Feed (Seitenwechsel)
\a		Bell (Klingelzeichen)
\\		Backslash (\)
\ddd		mit ddd als Ordinalwert
\xhhh		mit hhh als Ordinalwert in Hexadezimal Darstellung

Operatoren

Für char-Variable können auch die Operatoren des Datentyps int verwendet werden.

```
c = 'A' + 1 ;           // ergibt 'B'
if ((c>='a')&&(c<='z')) printf("Kleinbuchstabe");
for (c=65;c<=90;c++) cout << c;
```

Bibliothek <ctype.h> :

Funktionen für die Umwandlung auf Groß- oder Kleinbuchstaben, sowie der Analyse von Zeichen sind in der Bibliothek <ctype.h> enthalten.

Funktionen in <ctype.h> :

isdigit(c)	Abfrage ob das Zeichen eine Ziffer ist
isascii(c)	Abfrage auf ASCII-Zeichen
tolower(c)	Umwandlung auf Kleinbuchstaben
toupper(c)	Umwandlung auf Großbuchstaben
...	

Bibliothek <conio.h> :

Die Bibliothek <conio.h> wurde von Borland entwickelt und zählt nicht zu den Standardbibliotheken. Sie enthält Funktionen zur Bildschirmsteuerung sowie für das Ein- und Ausgeben von Zeichen.

Funktionen in <conio.h> :

clrscr()	Bildschirm löschen
gotoxy(x,y)	Cursor auf x,y positionieren
getch()	ein Zeichen einlesen
putch(c)	Zeichen ausgeben
textcolor(color)	
...	

```

////////////////////////////////////
//  Programmname : zeichen.cpp
////////////////////////////////////
#include <stdio.h>
#include <ctype.h>          // für toupper,tolower
#include <stdlib.h>         // für randomize,random
#include <iostream>
#include <conio.h>          // für getch,putch
using namespace std;

//  Ersatztexte für Steuerzeichen definieren
#define ENTER  0xD          // Ordinalwert für Entertaste
#define ESC    0x1B         // Ordinalwert für Esc-Taste

//  Hauptprogramm
int main()
{  char c,antwort;
   do
   {  clrscr();
      // ----- Zeichensatz ausgeben -----
      int i;
      cout << "  Zeichensatz-Tabelle  " << endl;
      for (i=32;i<=255;i++)          // nur druckbare Zeichen
      {  if (i%16==0) cout << endl << "  ";    // 16 Zeichen je Zeile
          cout << (char)i;                // Ordinalzahl als Zeichen ausgeben
      }
      cout << endl << endl << "  weiter mit beliebiger Taste  " << endl;
      getch();                        // weiter mit beliebiger Taste

      // ----- Ordinalwerte ausgeben -----
      clrscr();
      printf("\n  Ordinalwerte (Abbruch mit [Enter]) \n");
      do
      {  gotoxy(20,5);c=getch();          // ein Zeichen einlesen
          gotoxy(10,5);delline();
          printf("  Zeichen='%c' Ordinalwert= %d ",c,(unsigned char)c);
      } while(c!=ENTER);              // bis Enter-Taste

      // ----- Zeichen analysieren -----
      clrscr();
      printf("\n  Zeichen analysieren (Abbruch mit [Enter]) \n");
      do
      {  gotoxy(10,5);i=getch();delline();    // Zeichen einlesen
          if ((i>='0')&&(i<='9'))              // = Ziffer
          {  printf("  Ziffer "); }
          if (isprint(i))                      // = druckbares Zeichen
          {  printf("  durckbares Zeichen "); }
      } while(i!=ENTER);

      // ----- Zufalls-Zeichen -----
      clrscr();
      printf("\n  Zufallszeichen von a bis z (weiter mit [Enter]) \n");
      srand(0); gotoxy(10,5);
      for (int i=0;i<50;i++)
      {  c=rand()%26;                          // Zufallszahl von 0 bis 25
          putch('a'+c);                        // Zeichen von 'a' bis 'z'
      }
      getch();                                // weiter mit beliebiger Taste

      gotoxy(25,25); printf("  Programm Ende mit [Esc] ");
      antwort=getch();
   } while (antwort!=ESC);                // Programm-Ende mit ESC
}

```


Felder (Arrays)

Felder können mehrere Elemente eines gleichen Datentyps aufnehmen.

Feldvariable sind für jene Aufgaben vorteilhaft einzusetzen, wo eine Vielzahl von Daten eines gleichen Datentyps notwendig ist. Jedes Feldelement kann dabei wie eine einzelne Variable verwendet werden, wird jedoch über den gemeinsamen Feldnamen und einen Index angesprochen.

Definition eines Feldes : **Typ Feldname[Anzahl];**

Beispiel : `float a[10];`

vereinbart eine Feldvariable mit dem Namen `a` mit 10 Feldelementen vom Typ `float`.

Die Definition eines Feldes kann auch **über einen eigenen Typnamen** erfolgen :

`typedef Typ Feldtypname[Anzahl];`
`Feldtypname Feldname;`

Beispiel : `typedef int IntFeld[10];`
 `IntFeld a1,a2;`

Mit **`typedef`** wird dabei ein eigener Datentyp mit dem Namen `IntFeld` definiert, der für ein Feld mit 10 Feldelementen vom Typ `integer` gilt. Die Feldvariablenvereinbarung für `a1,a2` erfolgt dann über diesen Typnamen. Die Verwendung von benutzerdefinierten Feldtypen ist dann von Vorteil, wenn mehrere Feldvariable mit diesem Feldtyp vereinbart werden.

Die **Anzahl der Feldelemente** wird bei der Definition des Feldes in den eckigen Klammern `[n]` angegeben. Der **Indexbereich** des Feldes ist dann immer von **0 bis n-1** festgelegt.

Für das obige Feld `a` z.B. von `a[0],a[1]...a[9]` und ergibt insgesamt 10 Feldelemente.

Die **Definition der Anzahl** der Elemente kann auch vorteilhaft über eine Konstante erfolgen :

```
const int NMAX=10;
int a[NMAX];
```

Diese Konstante wird dann konsequent für den Indexbereich des Feldes verwendet und hat dann den Vorteil, dass das Programm lesbarer und fehlersicherer wird. Eventuelle Änderungen des Indexbereiches können dann auch sehr einfach über diese Konstante ausgeführt werden.

Feldvariable können als globale oder lokale Variable definiert werden.

Globale Variablen werden automatisch mit dem Wert 0 initialisiert, lokale Variable bekommen einen undefinierten Wert. Der erforderliche Speicherplatz für ein Feld ergibt sich aus der Anzahl der Feldelemente multipliziert mit dem Speicherbedarf des Grundtyps. Große Felder sollten als globale Variable angelegt werden.

Beispiel : `float a[10]; // benötigt 10*sizeof(float) = 40 Byte`
 `sizeof(a); // Größe des Arrays in Bytes bestimmen`
 `sizeof(a)/sizeof(float); // Anzahl der Elemente`

Initialisierung von Arrays : bei der Definition werden zugleich Anfangswerte zugewiesen

Beispiele : `int a[3]={1,2,3}; // -> a[0]=1;a[1]=2;a[2]=3;`
 `int a[] = {1,2,3}; // auch ohne Angabe der Feldgröße`
 `int a[3]={0}; // alle Elemente = 0`

Zugriff auf Feldelemente :

Ein Feldelement wird über den Feldnamen und der Angabe des Index angesprochen.
Der Indexbereich eines Feldes mit n-Elementen ist dabei immer von 0 bis n-1 festgelegt.

Feldelement Zugriff : ***Feldname[Index];***

```
Bsp.: a[0] = 1;           // 1-tes Feldelement mit Index 0
      a[1] = 2;           // 2-tes Feldelement mit Index 1
      ...
      a[9] = 10;          // 10-tes Feldelement mit Index 9
```

Achtung !

der C/C++ Compiler prüft nicht die Indexgrenzen, es ist daher für den Programmierer ein großes Augenmerk auf die Einhaltung des gültigen Indexbereiches zu legen. Indexüberschreitungen führen zu schlimmen und teilweise unentdeckten Laufzeitfehlern, wie Datenverlusten in anderen Variablen.

Alle Feldelemente können am einfachsten über **eine Schleifenanweisung** verarbeitet werden, wobei die Laufvariable der Schleife zugleich als Index des Feldes verwendet wird.

Beispiel 1 : Feldvariable definieren und verarbeiten

```
const int NMAX=10;           // Konstante NMAX
int a[NMAX];                 // Feldvariable mit NMAX Feldelementen
typedef float FloatFeld10[NMAX]; // Typdefinition FloatFeld10
FloatFeld10 a1,a2;           // Variablendefinition für a1,a2

int main()
{ int i,summe;
  float mittel;

  for (i=0;i<NMAX;i++)       // allen Elementen einen Wert zuweisen
  { a[i] = i+1;
  }

  for (i=0;i<NMAX;i++)       // alle Feldelemente ausgeben
  { cout<<setw(4)<<a[i];
  }

  summe = 0;
  for (i=0;i<NMAX;i++)       // Summe und Mittelwert berechnen
  { summe += a[i];
  }
  mittel=(float)summe/NMAX;

  for (i=0;i<NMAX;i++)
  { a1[i]=a[i]-mittel;        // Abweichungen zum Mittelwert berechnen
  }

  for (i=0;i<NMAX;i++)
  { a2[i]=a1[NMAX-1-i];       // Reihenfolge der Elemente umkehren
  }

  srand(time(NULL));
  for (i=0;i<NMAX;i++)
  { a[i]=rand()%100+1;        // Zufallszahlen von 1 bis 100 zuweisen
  }
}
```

Felder kopieren :

Felder können nur über die Zuweisung der einzelnen Feldelemente kopiert werden und nicht durch Zuweisung der Feldnamen.

Beispiel 2 : Felder kopieren

```
int a[10],b[10];
for (int i=0;i<10;i++)
{   b[i]=a[i];                       // alle Elemente a[i] auf b[i] zuweisen
}
```

Felder sortieren über Sortieralgorithmen :

Das Sortieren von Daten ist eine häufige Aufgabe und wird über Sortieralgorithmen erledigt. Bubble Sort und Insertion Sort sind einfache Sortierverfahren, die nur für eine kleine Anzahl von Elementen ausreichend sind. Für große Datenmengen werden effizientere und damit raschere Sortieralgorithmen wie Quicksort (qsort) verwendet.

Beispiel 3 : Felder sortieren

```
const int NMAX=10;
int a[NMAX],b[NMAX];           // Feldvariable a,b

for (i=0;i<NMAX;i++)
{   a[i] = rand()%100 + 1;      // Zufallszahlen von 1 bis 100 zuweisen
    b[i]=a[i];                  // Feld a auf b kopieren
}

//----- Bubble Sort -----
for (i=0;i<NMAX;i++)           // nehme i-tes Element von a[i],
{   for (j=i+1;j<NMAX;j++)      // vergleiche mit den restlichen Elementen
    {   if (a[i]>a[j])           // und tausche die Elemente wenn notwendig
        {   int hilf=a[i];
            a[i]=a[j];
            a[j]=hilf;
        }
    }
}

cout << endl << " sortiertes Feld a[i] : " << endl;
for (i=0;i<NMAX;i++)           // Feldelemente ausgeben
{   cout << setw(4) << a[i];
}

//----- Insertion Sort -----
for(i=1;i<NMAX;i++)             // nehme i-tes Element von b[i],
{   int m;
    m=b[i];                     // Wert speichern
    for(j=i;b[j-1]>m&&j>0;j--)    // kleineren Wert als m suchen
        b[j]=b[j-1];
    b[j]=m;                     // Lücke füllen
}

cout << endl << " sortiertes Feld b[i] : " << endl;
for (i=0;i<NMAX;i++)           // Feldelemente ausgeben
{   cout << setw(4) << a[i];
}
```

Mehrdimensionale Felder :

Es können auch zwei- oder mehrdimensionale Felder definiert werden.

Definition eines zweidimensionalen Feldes : Typ Feldname[n][m];

Beispiel : `int m[4][3];`

vereinbart ein zweidimensionales Feld mit der 1.Dimension mit der Größe = 4 und der 2.Dimension mit der Größe = 3.

Die Feldelemente werden im Speicher hintereinander folgend angelegt -

`m[0][0], m[0][1], m[0][2], m[1][0], m[1][1], m[1][2], m[2][0], ... , m[3][2]`

Die Darstellung eines zweidimensionalen Feld kann übersichtlich in Matrixform erfolgen -

```
m[0][0], m[0][1], m[0][2]
m[1][0], m[1][1], m[1][2]
m[2][0], m[2][1], m[2][2]
m[3][0], m[3][1], m[3][2]
```

Definition über einen Typnamen :

```
typedef int matrix[4][3];
Matrix m;
```

oder auch über den Typ eines eindim. Feldes :

```
typedef int IntFeld[3];
typedef IntFeld matrix[4];
```

Zugriff auf Feldelemente :

Die Feldelemente eines zweidimensionalen Feldes werden über geschachtelte Schleifen verarbeitet, wobei die Laufvariable der Hauptschleife als Index der 1. Dimension und die Laufvariable der Unterschleife als Index der 2. Dimension verwendet wird.

Beispiel 4 : zweidimensionales Array

```
const int NMAX=4;           // Größe der 1. Dimension
const int MMAX=3;           // Größe der 2. Dimension

int m[NMAX][MMAX];          // 2-dim.Feld m

int main()
{   int i,j;

    for (i=0;i<NMAX;i++)      // Schleife für 1.Dim. i=0..NMAX-1
    {   for (j=0;j<MMAX;j++)    // Schleife für 2.Dim. j=0..MMAX-1
        {   m[i][j]= j+MMAX*i; // laufende Nummer zuweisen
        }
    }

    cout << " Feld m[i][j] : ";
    for (i=0;i<NMAX;i++)      // alle Feldelemente ausgeben
    {   cout << endl;
        for (j=0;j<MMAX;j++)
            cout << setw(4) << m[i][j];
    }
}
```

```

////////////////////////////////////
// Programmname : matrix.cpp
////////////////////////////////////

#include <stdio.h>
#include <iostream>
#include <iomanip>
#include <time.h>
using namespace std;

// ----- globale Definitionen -----
const int NMAX=5;           // Größe der 1. Dimension
const int MMAX=4;           // Größe der 2. Dimension
typedef int matrix[NMAX][MMAX]; // Typdefinition matrix
matrix t;                   // 2-dim.Feld t

// ----- Hauptprogramm -----
int main()
{
    int i,j;

    system("CLS");           // Bildschirm löschen
    cout << " Matrix " << endl << endl;
    // ----- Zufallszahlen zuweisen -----
    cout << endl << endl << " Feld t[i][j] mit Zufallswerten : ";
    srand(time(NULL));
    for (i=0;i<NMAX;i++)      // Schleife für 1.Dim. i=0..NMAX-1
    {
        for (j=0;j<MMAX;j++)  // Schleife für 2.Dim. j=0..MMAX-1
        {
            t[i][j]=rand()%100+1; // Zufallszahlen zuweisen
        }
    }
    // ----- Zeilen- u. Spaltensummen berechnen -----
    cout << endl << endl << " Summe der Zeilen von t[i][j] : " << endl;
    for (i=0;i<NMAX;i++)
    {
        cout << endl;
        int sum_zeile=0;      // Summe über die Zeile
        for (j=0;j<MMAX;j++)
        {
            sum_zeile += t[i][j];
            cout << setw(4) << t[i][j];
        }
        cout << " = " << setw(4) << sum_zeile;
    }
    cout << endl << " ----- " << endl;
    int sum_gesamt=0;         // Gesamtsumme
    for (j=0;j<MMAX;j++)
    {
        int sum_spalte=0;     // Summe über die Spalte
        for (i=0;i<NMAX;i++)
        {
            sum_spalte += t[i][j];
        }
        cout << setw(4) << sum_spalte;
        sum_gesamt += sum_spalte;
    }
    cout << " = " << setw(4) << sum_gesamt;
    cout << endl << endl;
    system("PAUSE");
    return 0;
}

```

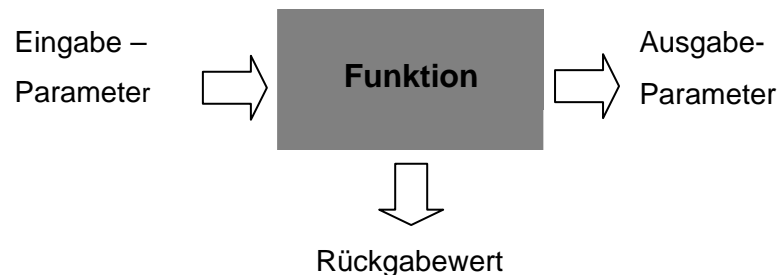
Unterprogramme (Funktionen)

Ein Unterprogramm ist die **Zusammenfassung mehrerer Anweisungen** unter einem Namen. Ein einmal definiertes Unterprogramm kann dann beliebig oft aufgerufen werden und erleichtert damit die Programmierung gleicher Programmteile.

Die **Aufteilung in Unterprogramme** ist ein wesentlicher Punkt der modernen Programmiertechnik. Große Programme können dadurch übersichtlich entwickelt, einfacher getestet und durch Verwendung fertiger Unterprogramme (Bibliotheksfunktionen) effizienter erstellt werden. Damit Unterprogramme möglichst universell verwendbar sind, ist eine **Parameter-Schnittstelle** für die flexible Datenübergabe notwendig.

In C/C++ werden Unterprogramme einheitlich **Funktionen (functions)** genannt. In anderen Programmiersprachen wird häufig zwischen Prozeduren (Unterprogramme ohne Rückgabewert) und Funktionen (Unterprogramme mit Rückgabewert) unterschieden.

Schema einer Funktion mit Parameter-Schnittstelle und Rückgabewert



Über die Parameter-Schnittstelle können Daten in die Funktion über **Eingabeparameter** oder auch Daten aus der Funktion über **Ausgabeparameter** übergeben werden. Ein Funktionsergebnis kann auch als **Rückgabewert** ausgegeben werden.

Funktionsdefinition und Funktionsaufruf

Die **Funktionsdefinition** ist die Festlegung einer Funktion mit Funktionsnamen, Parametern, Rückgabewert und den Anweisungen.

Der **Aufruf einer Funktion** ist die Ausführung der Funktion durch die Angabe des Funktionsnamens und Angabe von aktuellen Parametern.

Funktionsdefinition :

```
Rückgabety Funktionsname ( formale Parameter )  
{ lokale Definitionen  
  Anweisungen  
}
```

Funktionsaufruf :

```
Funktionsname ( aktuelle Parameter );
```

Funktionen ohne Parameter haben eine leere Parameterliste, Funktionen ohne Rückgabewert werden bei der Definition mit dem Rückgabotyp „void“ festgelegt.

Beispiel : Funktion ohne Parameter

```
// Funktionsdefinition
void trennlinie()
{   cout << " ----- " << endl ;
}

// Funktionsaufruf in main
int main()
{   ...
    trennlinie();
    ...
}
```

Auch das Hauptprogramm „int main()“ ist eigentlich als Unterprogramm ausgeführt, das vom Betriebssystem aufgerufen wird.

Funktionen mit Parameter haben bei der Definition ein oder mehrere Parameter angegeben, die wie Variablendefinitionen aussehen.

Die Funktion „trennlinie“ ist uns zu wenig universell, da sie nur eine Linie mit konstanter Länge erzeugt. Wir wollen daher eine Funktion „linie“ mit dem Parameter „laenge“ definieren, der die Linienlänge flexibel steuerbar macht.

Beispiel : Funktion mit Parameter

```
// Funktionsdefinition
void linie ( int laenge )
{   for( int i=0; i<laenge; i++) cout << "-";
}

// Funktionsaufrufe
int main()
{   int a=40;
    ...
    linie(20);                // Linie mit 20 Zeichen
    ...
    linie(a);                 // Linie mit a=40 Zeichen
    ...
}
```

Der **Parameter „laenge“** wird bei der Funktionsdefinition wie eine Variable verwendet, beim Aufruf wird als aktueller Parameter (Argument) ein beliebiger Wert (oder auch eine Variable) eingesetzt, für den die Funktion dann konkret ausgeführt wird.

Die aktuellen Parameter müssen dabei typmäßig zu den formalen Parameter übereinstimmen.

Man könnte die Steuerung der Linienlänge auch über eine globale Variable erledigen, die anstatt des Parameters auch im Unterprogramm verwendet wird. Der Zugriff auf globale Variable in Unterprogrammen führt jedoch zu sehr unübersichtlichen Programmen und sollte daher stets vermieden werden.

Zugriff auf globale Variable in einer Funktion

Auf globale Variable kann zwar prinzipiell in einer Funktion zugegriffen werden, jedoch können solche Funktionen dann nur sehr eingeschränkt verwendet werden.

Eine Funktion, die universell auch in weiteren Programmen (als Bibliotheksfunktion) eingesetzt werden soll, sollte daher alle Daten über die Parameterschnittstelle übergeben!

Beispiel : Zugriff auf globale Variable in einer Funktion

```
int laenge;                                // globale Variable

void linie ( )                             // Funktionsdefinition
{ for( int i=0; i<laenge; i++) cout << "-";
}

int main()
{
    ...
    laenge=40;
    linie();                                // Linie mit 40 Zeichen
    ...
}
```

Jedes Programm, das diese Funktion linie() verwendet, muss daher eine globale Variable „laenge“ definiert bekommen. Wären alle Bibliotheksfunktionen so ausgeführt, würde das zum Chaos führen! Funktionen mit Zugriff auf globale Variable sind daher **nicht modular** verwendbar.

Gültigkeit und Sichtbarkeit bei lokalen und globalen Variablen

Globale Variable werden außerhalb eines Blockes vereinbart, wie z.B. vor „main()“.

Lokale Variable werden innerhalb eines Blockes vereinbart, wie innerhalb eines Funktionsblockes oder Schleifenblockes, aber auch im Block von „main()“.

Gültigkeit von globalen und lokalen Variablen:

lokale Variable gelten nur im Block, in dem sie vereinbart sind.

globale Variable gelten in allen Blöcken und damit im gesamten Programm.

Sichtbarkeit :

Es können gleichnamige lokale Variable zu globalen Variablen vereinbart werden.

Dabei überdecken die lokalen Variablen innerhalb des Funktionsblockes die gleichnamigen globalen, wobei beide gültig sind, die globalen sind jedoch nicht sichtbar.

Damit wird bei gleichnamigen Variablen im Block automatisch die lokale angesprochen.

Namen- und Typ-Übereinstimmung von formalen zu aktuellen Parametern :

Die **formalen Parameter** bei der Funktionsdefinition gelten zugleich als lokale Variable im Funktionsblock. Die Namen der Parameter sind damit nur von lokaler Bedeutung zu anderen lokalen Variablen, jedoch ohne Namenskonflikt zu globalen Variablen.

Beispiel : globale und lokale Variablen

```
int i,n;                                    // globale Variable i,n
void leerzeilen (int n)                    // Parameter n = lokale Variable
{ int i;                                   // lokale Variable i
  for ( i=0;i<n;i++)
  { cout << endl;
  }
}
```

Innerhalb der Funktion „leerzeilen“ gelten zwar auch die globalen Variablen i und n, jedoch zugegriffen wird auf die lokalen, da sie im Funktionsblock sichtbar sind.

Funktionen mit Rückgabewert haben bei der Definition einen bestimmten Rückgabebetyp angeführt.

Über den Rückgabewert kann ein **Funktionsergebnis** ausgegeben werden.

Der Rückgabewert wird bei der Funktionsdefinition durch die **return Anweisung** erzeugt.

Funktionsdefinition mit return - Anweisung :

```
Rückgabotyp Funktionsname ( formale Parameter )
{ lokale Definitionen
  Anweisungen
  return Ausdruck;           // Definition des Funktionsergebniswertes
}
```

Der Rückgabotyp (Returtype) steht vor dem Funktionsnamen und muss mit dem Typ des Return-Ausdruckes übereinstimmen.

Der Funktionsaufruf ergibt dann das Funktionsergebnis, das in der Funktionsdefinition mit dem Return Ausdruck definiert wird. Der Funktionsaufruf erzeugt damit einen Wert mit dem Typ des Returntyps und kann unmittelbar in einem Ausdruck eingesetzt werden.

Beispiel : Funktion sqr mit Returnwert

```
float sqr (float x )           // berechnet das Quadrat von x
{ return x*x;                 // Funktionsergebnis erzeugen
}

int main()
{ float x,y,z;
  ...
  y = sqr(x);                 // Aufruf von sqr
  z = 1/sqr(x);              // Aufruf von sqr in einem Ausdruck
  ...
}
```

Die Funktion `sqr` gibt das Ergebnis über den Rückgabewert aus und kann z.B. direkt zur Weiterverarbeitung für die Berechnung `z = 1/sqr(x)` eingesetzt werden.

Die **return Anweisung** bricht dabei **die Funktion an dieser Stelle ab** und sollte daher stets als letzte Anweisung einer Funktion stehen oder unter einer if-Bedingung ausgeführt werden. Auch eine void-Funktion (mit ReturnTyp void) kann mit einer return Anweisung mit leerem Ausdruck abgebrochen werden.

Beispiel : Funktionen max und min

```
float max ( float x,float y)   // berechnet das Maximum von x,y
{ if (x>y) return x;           // return erzeugt Ergebnis und bricht ab
  else return y;
}

float min ( float x,float y)   // berechnet das Minimum von x,y
{ if (x<y) return x;           // return erzeugt Ergebnis und bricht ab
  else return y;
}
```

Funktionen mit Ausgabeparameter :

Ausgabeparameter sind immer dann notwendig, wenn Werte in einer Funktion erzeugt und an das rufende Programm zur Weiterverarbeitung übergeben werden.

Die Umsetzung von Ausgabeparametern ist in C über Zeigerparameter und in C++ über Referenzparameter möglich.

Prinzip der Parameterübergabe :

Bei der Programmausführung wird für den Mechanismus der Parameterübergabe und für den erforderlichen Speicherplatz von lokalen Variablen ein **Hilfsspeicher (Stack)** verwendet, der beim Rücksprung aus dem Unterprogramm automatisch wieder freigegeben und damit verworfen wird.

Parameter werden beim Aufruf der Funktion **als Kopie an den Stack** übergeben.

Damit können über die Parameterschnittstelle in direkter Weise nur Werte an die Funktion übergeben werden, aber keine Werte aus der Funktion heraus übernommen werden.

Sollen Werte aus einer Funktion (Ausgabeparameter) übernommen werden, so kann das nur **indirekt über die Adresse** (Referenz) von Variablen erfolgen. Über die übergebene Adresse kann dann der Wert der Variablen auch von der Funktion aus geändert werden.

Wertparameter (call by value) -

Parameter werden wertmäßig als Kopie an die Funktion übergeben.

Wertänderungen in der Funktion haben keine Rückwirkung.

Zeigerparameter u. Referenzparameter (call by reference) -

die Adresse oder Referenz einer Variable wird an die Funktion übergeben.

Der Zugriff auf eine Variable von der Funktion aus ist möglich.

Referenzparameter in C++ : **Typname &Parametername**

werden mit dem „kaufmännischen Und“- Zeichen '&' vor dem Parameternamen definiert.

Beispiel : Funktion mit Referenzparameter

```
int division( int z, int n, int &rest)    // ganzzahlige Division mit Rest
{
    rest = z%n;                          // ganzzahliger Rest
    return z/n;                          // ganzzahlige Division
}

int main()
{ int y,r;
  y = division(5,3,r);                   // Aufruf mit aktuellen Parametern
  cout << " Quotient = " << y << " Rest = " << r;
}
```

Die Funktion „division“ berechnet die ganzzahlige Division mit ganzzahligem Rest. Der Quotient wird als Rückgabewert, der ganzzahlige Rest über den **Referenzparameter „&rest“** ausgegeben.

Würde „rest“ nicht als Referenz- sondern nur als Wertparameter definiert sein, so könnte zwar in der Funktion der Wert für „rest“ zugewiesen, aber diese Wertänderung nicht an die beim Aufruf eingesetzte Variable „r“ übernommen werden!

Beispiel : Funktion tausche

```
void tausche(int &x,int &y)
{ int hilf=x;                          // Hilfsvariable
  x=y; y=hilf;                          // tauschen der Werte von x und y
}
```

Die Funktion „tausche“ soll die Werte der übergebenen Parameter x u. y tauschen.

Damit die Wertänderungen auch für die eingesetzten aktuellen Parameter wirksam werden, sind wiederum Referenzparameter erforderlich.

Zeigerparameter in C : **Typname *Parametername**
werden mit dem „Stern“- Zeichen '*' vor dem Parameternamen definiert.

Zeiger (Pointer) sind besondere Variable, die Adressen von Variablen aufnehmen.
Über den **Inhaltsoperator** '*' wird der Inhalt (Wert) einer Adresse (eines Zeigers) bestimmt.
Über den **Adressoperator** '&' kann die Adresse einer Variablen ermittelt werden.

Beispiel : Funktion division mit Zeigerparameter

```
int division( int z, int n, int *rest)      // *rest als Zeigerparameter
{
    *rest = z%n;                          // mit *rest auf den Inhalt zugreifen
    return z/n;
}

int main()
{ int y,r;
  y = division(5,3,&r);                   // Aufruf mit der Adresse &r
  cout << " Quotient = " << y << " Rest = " << r;
}
```

Die Funktion „division“ ist jetzt alternativ mit dem **Zeigerparameter** „*rest“ ausgeführt.
Dieser Zeiger übergibt dabei die Adresse der beim Aufruf eingesetzten Variablen „&r“.
Innerhalb der Funktion kann daher über den Inhaltsoperator „*rest“ auf den Wert der Variablen zugegriffen werden.

Merkmale für Zeigerparameter :

1. Zeigerparameter werden mit einem '*' definiert und übergeben Adressen
2. Über den Inhaltsoperator '*' kann auf den Wert an dieser Adresse zugegriffen werden
3. Beim Aufruf der Funktion wird für Zeigerparameter die Adresse '&' einer Variablen eingesetzt

Beispiel : Funktion tausche mit Zeigerparameter

```
void tausche(int *x,int *y)
{ int hilf=*x;                          // Inhalt von Zeiger x an hilf zuweisen
  *x=*y; *y=hilf;                       // Inhalte der Zeiger x und y tauschen
}

void main()
{ int a=1,b=-1;
  tausche(&a,&b);                        // Aufruf mit den Adressen von a und b
  ...
}
```

Wie würde die Tauscherei eigentlich ablaufen, wenn die Funktion „tausche“ so ausgeführt wäre :

```
void tausche(int *x,int *y)
{ int hilf=*x;
  x=y; *y=hilf;
}
```

Funktions Prototypen :

Ein Funktionsprototyp ist die Deklaration einer Funktion durch Angabe des Funktionskopfes.

ReturnTyp Funktionsname (formale Parameter);

Diese Deklaration muss mit der Funktionsdefinition hinsichtlich Parametertypen und dem Returntyp übereinstimmen. Parameternamen hingegen können bei Prototypen auch weggelassen werden.

Die Angabe der Funktionsdeklaration genügt bereits, um eine Funktion aufrufen zu können.

Die Funktionsdefinition kann dann auch erst nach dem Hauptprogrammblock erfolgen.

Die Headerdateien einer Bibliothek enthalten z.B. nur die Funktionsprototypen, um diese Funktionen dem Compiler bekanntzugeben. Die Funktionsdefinitionen werden dann in bereits übersetzter Form als Objektdateien nur mehr durch den Linker dazugebunden.

Beispiel : Funktionsprototypen

```
void trennlinie();
void linie ( int laenge );
float sqr (float);
int division( int z, int n, int &rest);
void tausche(int*,int*);
```

Spezifikation von Funktionen :

Die Aufgabe einer Funktion, die Verwendung der Parameter und des Rückgabewertes sollten als Kommentar nach dem Funktionskopf auch immer ausführlich angeführt werden.

```
int division( int z, int n, int &rest);
/*
    Berechnung der ganzzahligen Division mit Rest
    Import (Eingabedaten):  int z      ... Wert für Zähler
                           int n      ... Wert für Nenner
    Export (Ausgabedaten):  int &rest ... Rest der Division
    Return (Rückgabewert):  Quotient der Division

    Preconditions (Vorbedingungen):  Nenner ungleich Null ( n!=0 )
*/
```

Überladene Funktionen in C++ :

Überladene Funktionen sind Funktionen mit gleichen Namen aber unterschiedlichen Parametern. Überladene Funktionen sind dann sehr praktisch, wenn ähnliche Aufgaben mit unterschiedlichen Datentypen ausgeführt werden sollen.

Beim Funktionsaufruf entscheidet der Typ des Arguments, welche überladene Funktion vom Compiler ausgewählt und aufgerufen wird.

Beispiel : überladene Funktionen

```
void tausche(int &x,int &y)                // tausche int-Werte
{
    int hilf=x;
    x=y; y=hilf;
}

void tausche(float &x,float &y)            // tausche float-Werte
{
    float hilf=x;
    x=y; y=hilf;
}
```

Inline Funktionen :

Möchte man Funktionen bei zeitkritischen Aufgaben besonders rasch zur Ausführung bringen, so kann man sie als „inline“ definieren. Inline Funktionen werden nämlich bei der Übersetzung nicht als eigener Funktionsblock ausgeführt, sondern werden bei jedem Funktionsaufruf direkt ins Programm eingefügt. Dadurch werden sie einerseits rascher ausgeführt, da der Sprung zum Funktionsblock entfällt, andererseits wird das übersetzte Programm länger. Daher werden Inline Funktionen vorwiegend für kurze Funktionen, die rasch ausgeführt werden sollen, verwendet.

Die Umsetzung von inline Funktionen ist jedoch Sache des Compilers, der vielfältige Optimierungen automatisch durchführt und eventuell inline auch eigenmächtig abändert.

Definition einer Inline Funktion :

```
inline ReturnTyp Funktionsname ( formale Parameter )
{ ...
}
```

Beispiel : inline Funktionsdefinition

```
inline float sqr (float x )           // sqr als inline Funktion
{ return x*x;
}
int main()
{ float x,y,z;
  y = sqr(x);                        // Aufruf der inline Funktion wie üblich
}
```

static Variable :

Lokale Variable können als „static“ definiert werden. Statische Variablen haben auch nur lokale Gültigkeit, verlieren jedoch bei Verlassen des lokalen Bereiches (Funktionsblock) nicht ihren Wert. Damit hat man lokale Variable mit einer Merkfähigkeit für weitere Funktionsaufrufe.

Statische Variable sind also wie globale Variable, die jedoch nur im lokalen Block gültig sind.

Static Variable erhalten auch ihren Speicherplatz nicht am Stack sondern am Datenspeicher der globalen Variablen.

Beispiel : zaehlen mit static Variable

```
int zaehler( int i )                // Funktion mit static Variable
{ static int zaehlstand=0;          // static Variable initialisieren
  zaehlstand += i;                  // alten Zaehlstand erhöhen
  return zaehlstand;                // und als Returnwert ausgeben
}
int main()
{ ...
  cout << endl << " Zaehler mit static Variable : ";
  cout << endl << " Zaehlstand = " << zaehler(1);
  cout << endl << " Zaehlstand = " << zaehler(1);
  cout << endl << " Zaehlstand = " << zaehler(1);
  ...
}
```

Static Variable müssen bei der Definition mit einem Anfangswert initialisiert werden.

Bei jedem Aufruf der Funktion „zaehler“ wird der zuletzt gültige Wert der static Variable „zaehlstand“ um den aktuellen Wert des Parameters „i“ erhöht. Damit erfolgt zu den oben angeführten Funktionsaufrufen folgende Ausgabe :

```
Zaehler mit static Variable :
Zaehlstand = 1
Zaehlstand = 2
Zaehlstand = 3
```

Felder als Parameter :

Felder können auch als Parameter übergeben werden.

Felder werden immer per Referenz, nämlich über die Anfangsadresse des Feldes übergeben.

Es gibt drei mögliche Ausführungsformen für Felder als Parameter :

Feld-Parameter über benutzerdefinierten Feldtyp :

```
typedef int IntFeld[NMAX];           // Typdefinition für Array

void array_init(IntFeld a, int wert) // Feldparameter mit Feld-Typ
{
    int i;
    for (i=0;i<NMAX;i++)
    {
        a[i]=wert;
    }
}
```

Feld-Parameter über Feldklammern :

```
int array_sum( int a[], int n)        // Feldparameter mit Feldklammern
{
    int i,sum=0;
    for (i=0;i<n;i++)
    {
        sum += a[i];
    }
    return sum;
}
```

Feld-Parameter über Zeigerparameter :

```
void array_out( int *a, int n )       // Feldparameter als Zeigerparameter
{
    int i;
    for (i=0;i<n;i++)
        cout << setw(4) << a[i];
    cout << endl;
}
```

Der **Funktionsaufruf** ist bei allen 3 Formen gleich :

Beim Funktionsaufruf ist als aktueller Parameter der Feldname (ohne Indexklammern) anzugeben.

```
int main()
{
    IntFeld a;
    int b[10];
    ...
    array_init(a,1);
    cout << array_sum(b,10) << endl;
    array_out(b,10);
    ...
}
```

mehrdimensionale Felder als Parameter :

über Typnamen :

```
typedef int matrix[10][10];
void matrix_init( matrix m,float wert ) // mit Feld-Typnamen
```

oder über Feldklammernangabe :

```
float matrix_sum( float m[][10],int n ) // untergeordnete Dimensionen
                                           // müssen die Größe angeben
```

Beispiel : Summe, Mittelwert, Maximum und Minimum von Arrays

```
void array_rand( int a[], int n, int r1, int r2)
//    dem Array a Zufallszahlen im Bereich von r1 bis r2 zuweisen
{   int i;
    srand(time(NULL));
    for (i=0;i<n;i++)
    {   a[i]=rand()%(r2-r1+1)+r1;
    }
}

int array_sum ( int a[], int n )
//    Summe der ersten n-Feldelemente des Arrays a berechnen
{   int i,sum=0;
    for (i=0;i<n;i++)
    {   sum += a[i];
    }
    return sum;
}

float array_mittel ( int a[], int n )
//    Mittelwert der ersten n-Feldelemente des Arrays a berechnen
{   return (float)array_sum(a,n)/n;
}

int array_min ( int a[], int n )
//    Minimum der Feldelemente des Arrays a berechnen
{   int i,min=a[0];           // vorläufiges Minimum ist erstes Feldelement
    for (i=0;i<n;i++)
    {   if (min > a[i]) min=a[i];
    }
    return min;
}

int array_max ( int a[], int n )
//    Maximum der Feldelemente des Arrays a berechnen
{   int i,max=a[0];           // vorläufiges Maximum ist erstes Feldelement
    for (i=0;i<n;i++)
    {   if (max < a[i]) max=a[i];
    }
    return max;
}

//    Hauptprogramm
int main()
{   const int NMAX=10;
    int a[NMAX];

    system("CLS");
    cout << " Feldarbeiten  " << endl << endl;

    cout << " Feld a[i] : " ;
    array_rand(a,NMAX,1,10);
    array_out(a,NMAX);
    cout << " Summe           = " << array_sum(a,NMAX) << endl;
    cout << " Mittelwert      = " << array_mittel(a,NMAX) << endl;
    cout << " Minimum         = " << array_min (a,NMAX) << endl;
    cout << " Maximum         = " << array_max (a,NMAX) << endl;

    system("PAUSE");
    return 0;
}
```

Beispiel : Sortieren von Arrays

```
// globale Definitionen
const int NMAX=10;           // Feldgröße
int a[NMAX],b[NMAX];        // Feldvariable a,b

void bubble_sort( int a[], int n)
// sortiert das Array a[] ansteigend nach der Methode Bubble-Sort
{
    int i,j,hilf;
    for (i=0;i<n;i++)        // nehme i-tes Element,
    {
        for (j=i+1;j<n;j++)  // vergleiche mit den restlichen Elementen
        {
            if (a[i]>a[j])    // und tausche die Elemente wenn notwendig
            {
                int hilf=a[i];
                a[i]=a[j];
                a[j]=hilf;
            }
        }
    }
}

void selection_sort( int a[], int n)
// sortiert das Array a[] ansteigend nach der Methode Selection-Sort
{
    int i,j,hilf;
    for (i=0;i<n;i++)        // nehme i-tes Element,
    {
        int min=i;          // Index des kleinsten Elementes
        for (j=i+1;j<n;j++)  // vergleiche mit den restlichen Elementen
        {
            if (b[min]>b[j])  // und bestimme den Index des kleineren El.
            {
                min=j;
            }
        }
        if (min!=i)          // wenn neues Minimum, dann
        {
            int hilf=b[i];   // tausche das kleinste Element
            b[i]=b[min];     // an die i-te Position
            b[min]=hilf;
        }
    }
}

// Hauptprogramm
int main()
{
    int i, j;

    system("CLS");
    srand(time(NULL));
    for (i=0;i<NMAX;i++)
    {
        a[i] = rand()%100 + 1; // Zufallszahlen von 1 bis 100 zuweisen
        b[i]=a[i];            // Feld a auf b kopieren
    }

    //----- Bubble Sort -----
    cout << endl << " Bubble Sort : " << endl;
    bubble_sort(a,NMAX);
    array_out( a,NMAX);

    //----- Selection Sort -----
    cout << endl << " Selection Sort : " << endl;
    selection_sort(b,NMAX);
    array_out( b,NMAX);

    cout << endl << endl;
    system("PAUSE");
    return 0;
}
```


Beispiel : zweidimensionale Arrays als Parameter

```
// globale Definition des zweidimensionalen Arrays
const int NMAX=10;           // Größe der 1. Dimension
const int MMAX=10;           // Größe der 2. Dimension
typedef int matrix[NMAX][MMAX]; // Typdefinition matrix
matrix m;                     // 2-dim.Feld m

void matrix_random( matrix m, int r1, int r2 )
// der Matrix m Zufallszahlen im Bereich von r1 bis r2 zuweisen
// Import : matrix m ... Matrix m ( 2.dimensionales Feld )
//          int r1 ... kleinster Zufallswert
//          int r2 ... größter Zufallswert
// Return : void ... kein Rückgabewert
//
{
    int i,j;
    srand(time(NULL));
    for (i=0;i<NMAX;i++)
    {
        for (j=0;j<MMAX;j++)
        {
            m[i][j]=rand()%(r2-r1+1)+r1;
        }
    }
}

int matrix_sum( matrix m )
// Summe aller Feldelemente der Matrix m berechnen
// Import : matrix m ... Matrix m ( 2.dimensionales Feld )
// Return : int ... Summe aller Feldelemente
//
{
    int i,j,sum=0;
    for (j=0;j<MMAX;j++)
    {
        for (i=0;i<NMAX;i++)
        {
            sum += m[i][j];
        }
    }
    return sum;
}

//
// Hauptprogramm
//

int main()
{
    system("CLS");           // Bildschirm löschen
    cout << " Matrix " << endl << endl;

    cout << endl << " Zufallswerte von 1 bis 100 zuweisen : " << endl;
    matrix_random( m,1,100);
    matrix_out( m );
    cout << endl << " Summe aller Elemente = " << matrix_sum(m) << endl;

    cout << endl << endl;
    system("PAUSE");
    return 0;
}
```

Vorbelegte Parameter

In C++ können Parameter bereits bei der Funktionsdefinition mit bestimmten Werten belegt werden. Beim Aufruf der Funktion können die in der Parameterliste letzten vorbelegten Parameter dann auch weggelassen werden und erhalten somit automatisch die vorbelegten Werte.

Beispiel : vorbelegte Parameter

```
void linie ( int laenge=80, char zeichen='-' )
{   for( int i=0; i<laenge; i++) cout << zeichen;
    cout << endl;
}
int main()
{   ...
    linie();                // Aufruf mit laenge=80 u. zeichen='-'
    linie(20);              // Linie mit laenge=20 u. zeichen='-'
    linie(10, '~');          // Linie mit laenge=10 u. zeichen='~'
}
```

Parameter von main

Das Hauptprogramm „main()“ ist eine Funktion, die vom Betriebssystem aufgerufen wird und kann auch mit Parameter und Rückgabewert ausgeführt werden. Die Anzahl der beim Aufruf von main() übergebenen Parameter wird über den formalen Parameter „argc“, die Werte der Aufrufparameter über „argv[]“ ermittelt. Der Parameter argv[] stellt dabei ein String-Array mit der Größe argc dar und übernimmt die Werte der Aufrufparameter wie folgend angegeben -

argv[1] ... Wert des 1.Parameters
argv[2] ... Wert des 2.Parameters
argv[3] ... Wert des 3.Parameters

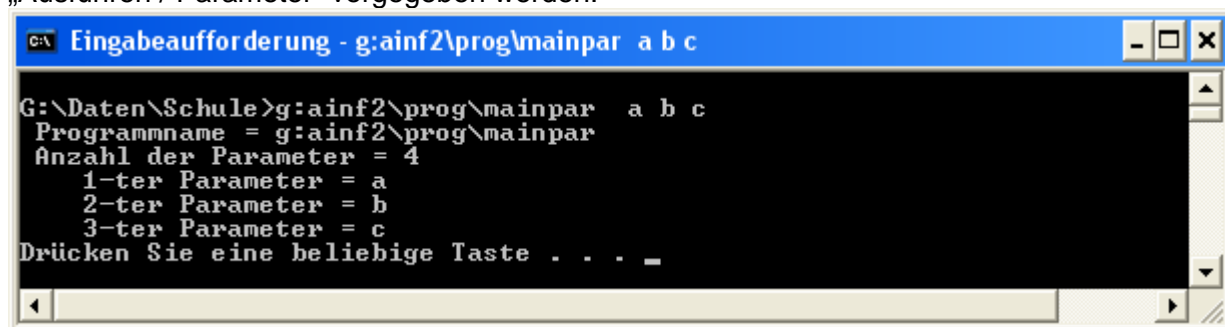
...

der String argv[0] enthält die Pfadangabe des Programmaufrufes (z.B.: ainf2\prog\mainpar)

Beispiel : Parameter von main() (Quellprogrammdatei = mainpar.cpp)

```
int main(int argc, char *argv[])
{   int i;
    cout << " Programmname = " << argv[0] << endl;
    cout << " Anzahl der Parameter = " << argc << endl;
    for(i=1; i<argc; i++)        // alle Aufrufparameter ausgeben
        cout << "          " << i << "-ter Parameter = " << argv[i] << endl ;
    return 0;                    // Rückgabewert (Exitcode)=0 für fehlerfreie Ausführung
}
```

Das übersetzte und ausführbare Programm (mainpar.exe) kann dann im Konsolenfenster (Eingabeaufforderung) mit Angabe der Aufrufparameter aufgerufen werden. (z.B.: mainpar a b c) Die Aufrufparameter können aber auch zum Austesten in der IDE-DevC++ über den Menüpunkt „Ausführen / Parameter“ vorgegeben werden.



Rekursion

Eine Funktion kann rekursiv aufgerufen werden, das heißt, die Funktion ruft sich selbst auf. Es gibt Aufgaben, die sich durch rekursiven Funktionsaufruf am besten lösen lassen. (z.B. schnelles Sortieren mit quicksort, binäres Suchverfahren, „Türme von Hanoi“, ...)

Beispiel : Berechnung der Fakultät $n!$ durch rekursiven Funktionsaufruf

```
// Fakultät :  $n! = 1*2* \dots *n = n*(n-1)!$ 
long fakult(long n)
{   if ( n > 1 ) return ( n*fakult(n-1) );    // rekursiver Aufruf von fakult
    else          return 1;                  // Abbruch bei  $n \leq 1$ 
}
```

Jeder rekursive Aufruf der Funktion fakult erhält einen eigenen Speicherbereich am Stack. Rekursive Funktionsaufrufe werden dadurch beendet, indem ab einer Bedingung ($n \leq 1$) in den jeweiligen vorigen Funktionsaufruf zurückgesprungen wird bis zurück zum ersten und schließlich aus der Funktion. Wichtig für rekursive Funktionen ist daher die korrekte Abbruchbedingung.

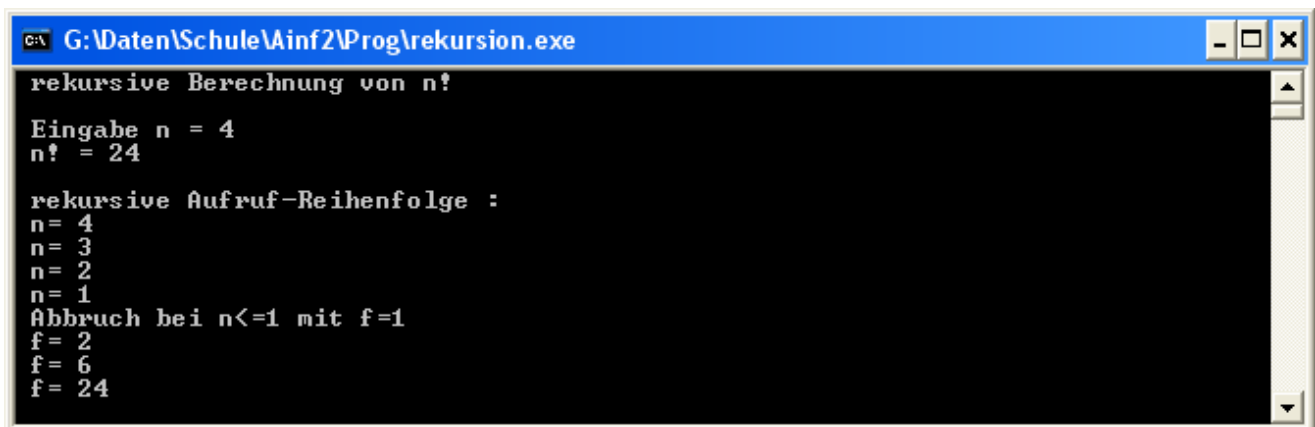
Demoversion zu fakult mit Ausgabe von Zwischergebnissen

```
long fakultdemo(long n)
{   long f;
    cout << " n= " << n << endl;           // Ausgabe des aktuellen Wertes n
    if ( n > 1 )
    {   f= n*fakultdemo(n-1);
        cout << " f= " << f << endl;        // Ausgabe der aktuellen Faktultät f
        return f;
    }
    else
    {   cout << " Abbruch bei  $n \leq 1$  mit  $f=1$  " << endl; // Abbruch bei  $n \leq 1$ 
        return 1;
    }
}

int main()
{   int n,f;

    cout << " Eingabe n = " ; cin >> n;
    f=fakult(n);
    cout << " n! = " << f << endl << endl ;

    cout << " rekursive Aufruf-Reihenfolge : " << endl;
    f=fakultdemo(n);
    cout << endl << endl;
    system("PAUSE");
    return 0;
}
```



```
G:\Daten\Schule\Ainf2\Prog\rekursion.exe
rekursive Berechnung von n!
Eingabe n = 4
n! = 24

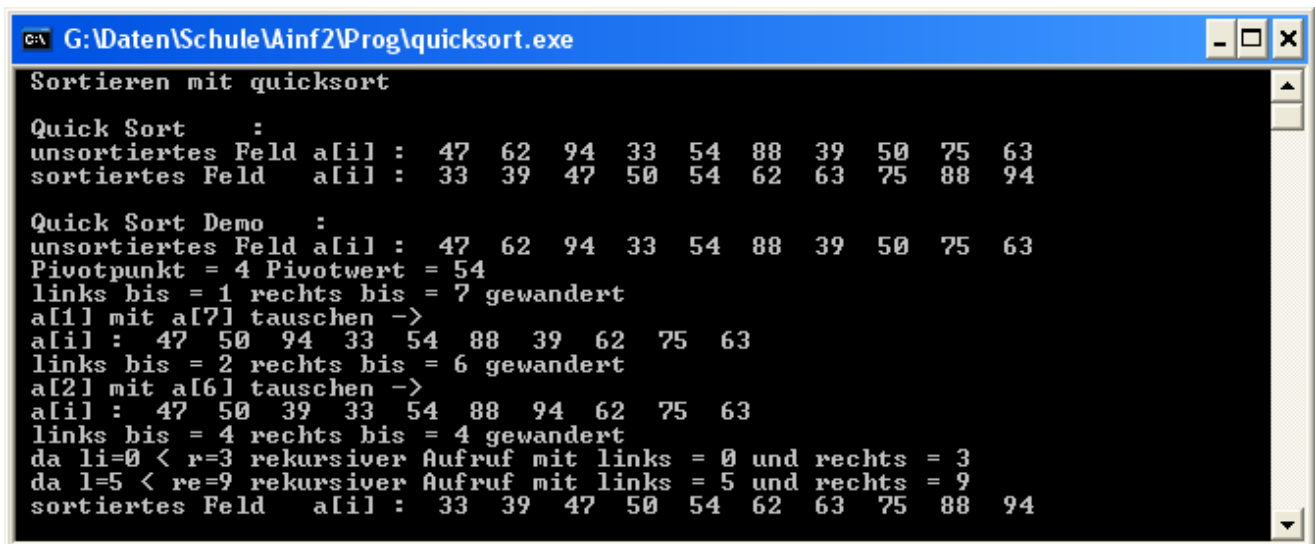
rekursive Aufruf-Reihenfolge :
n= 4
n= 3
n= 2
n= 1
Abbruch bei  $n \leq 1$  mit  $f=1$ 
f= 2
f= 6
f= 24
```

Beispiel : quicksort

Quicksort ist eines der schnellsten Sortierverfahren. Die Sortiermenge wird dabei in zwei Teilmengen (links u. rechts) aufgeteilt und die Elemente werden dann so getauscht, dass alle Elemente der linken Teilmenge kleiner und alle Elemente der rechten Teilmenge größer als der Drehpunkt (Pivotelement) sind. Das Pivotelement wird üblicherweise als mittleres Element der Sortiermenge gewählt. Quicksort wird dann jeweils rekursiv für die linke und rechte Teilmenge aufgerufen bis alle Teilmengen sortiert vorliegen.

```
void quicksort( int a[],int li, int re )
// sortiert das Array a[] ansteigend nach der Methode Quicksort
// Import : int a[]      ... Integer-Array
//          int li       ... Position links von Pivotelement
//          int re       ... Position rechts von Pivotelement
// Return : void        ... ohne Rückgabewert
{
    int l=li,r=re;
    int pivot=(l+r)/2;           // Pivotpunkt
    int pel=a[pivot];           // Wert des Pivotelementes
    do
    {
        while (a[l]<pel) l++;    // wandere mit l nach rechts bis zu pel
        while (a[r]>pel) r--;    // wandere mit r nach links bis zu pel
        if (l < r)              // und tausche wenn linker < rechter Position
        {
            int hilf=a[l];
            a[l]=a[r];
            a[r]=hilf;
        }
        if (l<=r)
        { l++;r--; }
    } while (l <= r);           // wiederhole solange linke <= rechte Position
    if (li < r) quicksort(a,li,r); // rekursiver Aufruf
    if (l < re) quicksort(a,l,re); // rekursiver Aufruf
}

int main()
{
    ...
    cout << " unsortiertes Feld a[i] :"; array_out(a,NMAX);
    quicksort(a,0,NMAX-1);
    cout << " sortiertes Feld   a[i] :";   array_out(a,NMAX);
    ...
}
```



```
Sortieren mit quicksort

Quick Sort :
unsortiertes Feld a[i] : 47 62 94 33 54 88 39 50 75 63
sortiertes Feld  a[i] : 33 39 47 50 54 62 63 75 88 94

Quick Sort Demo :
unsortiertes Feld a[i] : 47 62 94 33 54 88 39 50 75 63
Pivotpunkt = 4 Pivotwert = 54
links bis = 1 rechts bis = 7 gewandert
a[1] mit a[7] tauschen ->
a[i] : 47 50 94 33 54 88 39 62 75 63
links bis = 2 rechts bis = 6 gewandert
a[2] mit a[6] tauschen ->
a[i] : 47 50 39 33 54 88 94 62 75 63
links bis = 4 rechts bis = 4 gewandert
da li=0 < r=3 rekursiver Aufruf mit links = 0 und rechts = 3
da l=5 < re=9 rekursiver Aufruf mit links = 5 und rechts = 9
sortiertes Feld  a[i] : 33 39 47 50 54 62 63 75 88 94
```

Zeichenketten (Strings)

Ein String ist eine Zeichenkette und kann mehrere Zeichen und damit ganze Texte aufnehmen.

Die **C++ Bibliothek** stellt zur Stringverarbeitung die **String-Klasse „string“** zur Verfügung.

C-Strings hingegen werden als **Felder vom Typ char** definiert und können über die Funktionen der Bibliothek „string.h“ verarbeitet werden.

Die C++ Stringklasse sollte bevorzugt verwendet werden, da sie den notwendigen Speicherplatz dynamisch je nach Bedarf der aktuellen Stringlänge automatisch reserviert und die Verarbeitung über Operatoren einfacher und sicherer hinsichtlich Feldgrenzenüberschreitungen erledigt.

C++ Strings

Die C++ Stringklasse wird über das Headerfile <string> eingebunden. Damit steht der Typname „string“ zur Stringdefinition zur Verfügung. Die Stringklasse ermöglicht dann die Stringverarbeitung über Operatoren und Funktionen.

Stringdefinition

```
string s1;                // leerer String
string s2("klasse Strings"); // oder String mit Anfangstext
```

Operatoren der Klasse string:

=	Zuweisung
+	Verkettung
+=	Verkettung und Zuweisung
==	Vergleich auf Gleichheit
<	Vergleich auf alphabetische Reihenfolge
>	Vergleich auf verkehrte alphabetische Reihenfolge
...	

Funktionen der Klasse string:

length()	liefert die Stringlänge
insert(n,s)	String s an Position n einfügen
erase(p,n)	ab der Position p n-Zeichen löschen
find(s)	liefert die Position von Teilstring s im String
replace(p,n,s)	Teilstring s im String ab Position p ersetzen
...	

Die Funktionen der Stringklasse werden als Methoden folgend aufgerufen :

Stringname.Funktionsname(...);

Beispiel mit C++ Stringklasse:

```
#include <string>                // Stringklasse Headerfile

using namespace std;

int main()
{
    string s,s1,s2,;              // String(objekte) definieren
    string name(" klasse Strings "); // String mit Anfangswert initialisieren

    s1 = "abc";                   // Stringzuweisung über Operator =
    s2 = "def";
    s = s1 + s2;                  // Stringverkettung über Operator +

    int len=s.length();          // Stringlänge über Methodenaufruf
    ...
}
```

```

////////////////////////////////////
//  Programmname : stringklasse.cpp
////////////////////////////////////

//  Header-Dateien
#include <stdio.h>
#include <iostream>
#include <iomanip>
#include <string>                                // Stringklasse Headerfile
using namespace std;

//  Hauptprogramm
int main()
{
    int pos,n;
    string s1;                                // String(objekt) definieren
    string s2=s1;                             // String mit String s1 initialisieren
    string name(" klasse Strings ");          // String mit Anfangswert initialisieren
    string text(20,' ');                      // String mit 20 '-'Zeichen initialisieren
    string kopie(text);                       // String mit text initialisieren

    string liste[10]={"Eins","Zwei","Drei","Vier","Fuenf","Sechs","Sieben",
                     "eine","alte","Frau"}; // Array mit 10 Strings

    const string OK_MSG="OK \n";             // Stringkonstante definieren

    system("CLS");
    cout << " C++ Strings " << endl << endl;

    cout << " Einlesen in String s1 : "; cin >> s1; // Strings einlesen
    cout << " Einlesen in String s2 : "; cin >> s2;
    text = s1 + s2;                          // Stringverkettung und Stringzuweisung
    cout << " Strings s1 und s2 verketteten : " << text << endl;
    cout << " neue Stringlaenge = " << text.length() << endl << endl;
    pos=s1.length();
    text.insert(pos,"-insert-");              // String an der Position pos einfügen
    cout << " String an bestimmter Position einfuegen : " << text << endl;
    text.erase(pos+1,n=6);                   // ab Position pos+1 n Zeichen entfernen
    cout << " n-Zeichen ab bestimmter Position entfernen : " << text << endl;

    cout << " String zeichenweise ausgeben : ";
    for (int i=0; i<text.length();i++)
        cout << " " << text[i];             // einzelne Zeichen ausgeben

    text="Die Klasse string bietet gegenueber den C-Strings einige Vorteile \n";
    cout << "Suche mit find - " << endl;
    n = text.find("Die");                    // Teilstring im String suchen
    if (n!=string::npos) cout << " \"Die\" ist an der Position : " << n << endl;
    n = text.find("Die",n+1);
    if (n!=string::npos) cout << " und an der Position : " << n << endl << endl;

    cout << "Ersetzen mit replace - " << endl;
    text.replace(4,13,"Stringklasse");      // Teilstring im String ersetzen

    text.clear();                            // String löschen
    for (int i=0; i<10;i++)
        text += liste[i];                   // Stringarray zu einem String verketteten
    cout << text << endl;

    system("PAUSE");
    return 0;
}

```

C - Strings

C-Strings werden als Array vom Typ `char` definiert und müssen daher bereits bei der Definition mit der maximal notwendigen Stringlänge festgelegt werden. Das jeweils aktuelle Stringende wird intern über das Stringendezeichen (binäre Null = `'\0'` = 0) gekennzeichnet und ist bei der Festlegung der Stringlänge mit zu berücksichtigen.

Vereinbarung :

direkte Stringdefinition :

char Stringname[Anzahl];

Bsp.: `char s[20];`

vereinbart einen String mit dem Namen `s` für 20 Zeichen und damit maximal 19 nutzbaren Zeichen, da ein Zeichen für das Stringendezeichen reserviert werden muß.

oder über einen eigenen Typnamen :

typedef char Stringtypname[Anzahl];
Stringtypname Stringname;

Bsp.: `typedef char string80[81];`
`string80 s1,s2;`

Achtung !, es erfolgt wie bei allen Feldern keine Prüfung auf die Indexgrenzen !

Stringzuweisung :

C-Strings können nicht direkt mit dem Zuweisungsoperator (=) zugewiesen werden, sondern dazu muß die Bibliotheksfunktion **`strcpy`** (string copy) aus **`<string.h>`** verwendet werden. In C++ hingegen kann die Zuweisung durch das Überladen des Operators = über eine Stringklasse ausgeführt werden.

Bsp.: `strcpy(s,"Hallo");` // kopiert s <= "hallo"
`strcpy(s2,s1);` // kopiert s2 <= s1

Einlesen und Ausgeben :

Einlesen :

```
cin >> s1;
scanf("%s",s1);
gets(s1);
```

Ausgabe :

```
cout << s2;
printf("%s",s2);
puts(s2);
```

Bei `scanf` / `printf` kann über den Formatstring `%s` ein String eingelesen / ausgegeben werden. Die Funktion **`gets(s)`** aus **`<stdio.h>`** ermöglicht das Einlesen von Strings auch mit Leerzeichen.

Zugriff auf Einzelzeichen :

Über den Feldindex kann auf die einzelnen Zeichen eines Strings zugegriffen werden.

Bsp.: `s[0]= 'a'` // erstes Zeichen von s = 'a'
`s[1]= 'b'` // zweites Zeichen
`s[2]= '\0'` // Stringendezeichen zuweisen
// oder auch über `s[2]=0` die binäre Null als Int-Wert zuweisen
`printf("%c",s[0]);` // erstes Zeichen ausgeben

String - Endezeichen `'\0'` (zero-terminated string)

Das letzte Zeichen eines C-Strings ist das Steuerzeichen mit dem Ordinalwert 0 (= `'\0'`) .

Alle Bibliotheksfunktionen verarbeiten Strings korrekt über dieses Endezeichen.

Bei eigenen Stringroutinen muss natürlich auch das Stringendezeichen gesetzt werden !

```

Bsp.: Stringlänge über Bibl.funktion strlen bestimmen
      n=strlen(s) // Bibliotheksfunktion stringlength

Bsp.: Stringlänge über eigene Routine bestimmen
      n=0; while(s[n]) n++; // bis Stringende (s[n]==0) gehen

Bsp.: Zeichen eines Strings einzeln ausgeben
      for (i=0;i<strlen(s);i++)
      { putchar(s[i]); }
      // oder über die Bedingung s[i]!=0 (= bis Stringende )
      for (i=0;s[i];i++)
      { putchar(s[i]); }

```

Strings als Parameter :

Für Stringparameter gelten prinzipiell die **gleichen Regeln wie für Feldparameter**.
Die C- Bibliotheksfunktionen verwenden für Strings vorwiegend Zeigerparameter.

```

Bsp.: char *strcpy(char *dest, const char *src);
      // Aufruf :
      strcpy(s,"abc123");

```

Durch den **Modifizierer const** bei einem Zeigerparameter kann die Funktion den Wert der Variablen (des Strings) nicht verändern und damit Felder nach dem Prinzip Eingabeparameter übergeben.

```

Bsp.: Funktion zur Bestimmung der Anzahl bestimmter Zeichen im String
      int zeichensuche( char *s , char c )
      { int i,z=0;
        for (i=0;s[i];i++) // alle Zeichen bis Stringende (s[i]!=0)
          if ( s[i]==c ) z++; // zähle die gefundenen Zeichen
        return z; // liefere die Anzahl zurück
      }
      // Aufruf :
      printf("Anzahl von Zeichen 'A' im String %s = %d ",s1,zeichensuche(s1,'A'));

```

weitere String-Bibliotheksfunktionen aus string.h :

Stringverkettung : **char *strcat(char *dest, const char *src);**

```

Bsp.: char s1="Dennis",s2="M. Ritchie";
      strcat(s1,s2); // ergibt für s1 = "Dennis M. Ritchie"
                     // hoffentlich wurde s1 groß genug definiert!

```

Stringvergleich : **int strcmp(const char *s1, const char*s2);**

die Funktion liefert folgenden Rückgabewert :

```

0      bei s1=s2
<0     s1 ist alphabetisch vor s2
>0     s1 ist alphabetisch nach s2

```

```

Bsp.: i=strcmp("abc","abc"); // ergibt i= 0 da s1==s2
      i=strcmp("abc","abb"); // ergibt i>0 da s1>s2
      i=strcmp("abc","bcd"); // ergibt i<0 da s1<s2
      i=strcmp("abc","a");   // ergibt i>0 da s1>s2

```


Suche nach Zeichen : `char *strchr(const char *s, char c);`

sucht im String s nach dem Zeichen c und gibt die Adresse des gefundenen Zeichen zurück.

```
Bsp.: char* pc;                                // char-Zeiger definieren
      pc = strchr(s, 'a' );                    // Zeichensuche
      if (pc) printf("Position= %d", pc-s);    // Position = Adresse-Stringanfang
      else   printf("Zeichen nicht vorhanden ");
```

Suche nach String : `char *strstr(const char *s1, const char s2);`

sucht im String s1 nach dem Teilstring s2 und gibt die Adresse des gefundenen Teilstrings zurück.

```
Bsp.: char* pc;                                // char-Zeiger definieren
      pc = strstr(s, text);                    // sucht text in s
      if (pc) printf("Position= %d", pc-s);    // Position= Adresse-Stringanfang
```

Teilstring bis zu Trennstring bestimmen : `char *strtok(char *s1, const char s2);`

erzeugt den Teilstring bis zu einem Trennstring und gibt die Adresse des Teilstrings zurück.

```
Bsp.: pc = strtok(s, "," );                    // erster Teilstring bis ","
      while (pc)
      { printf("%s : ", pc);
        pc = strtok(NULL, "," );              // weitere Teilstrings bis ","
      }
```

Umwandlung auf numerische Variable und umgekehrt : `sscanf, sprintf` aus `<stdio.h>`

`sscanf` konvertiert einen String in eine numerische Variable

```
Bsp.: sscanf (s, "%6.2f", &x);                // String s in float Var. x umwandeln
// bei fehlerfreier Umwandlung ist der Returnwert 0 .
```

`sprintf` konvertiert eine numerische Variable in einen String

```
Bsp.: sprintf(s, "%6.2f", x);                  // float x in den String s umwandeln
```

Die Umwandlung kann auch über die to- Funktionen aus `<stdlib.h>` erfolgen :

`itoa` (int auf string), `ltoa` (long auf string), ...

`atoi` (string auf int), `atol` (string auf long), `atof` (string auf float), ...

String Listen :

Über ein Array von Strings kann eine Stringliste definiert werden.

```
Bsp.: typedef char name[21];                    // String-Typdefinition
      name liste[10];                          // Liste für 10 Strings

oder: char liste[10][21];                      // direkte Definition der Stringliste
      ...                                     // als zweidimensionales Array
      strcpy(liste[0], "Ritchie");

Bsp.: Funktion sort für alphabetisches Sortieren einer Stringliste
void sort(name *l, int n)
{ int i, j;
  name s;
  for ( i=0; i<n; i++)                        // Bubble Sort der ersten n-Elemente
  { for ( j=i+1; j<n; j++)
    { if (strcmp(l[i], l[j])>0)                // String Vergleich
      { strcpy(s, l[i]);                      // wenn nicht ansteigend, dann tauschen
        strcpy(l[i], l[j]);
        strcpy(l[j], s);
      }
    }
  }
}
```

```

////////////////////////////////////
//  Programmname : cstrings.cpp
////////////////////////////////////

//  Header-Dateien
#include <stdio.h>
#include <iostream>
#include <iomanip>
#include <string.h>                                // C-String Bibliothek
using namespace std;

//  globale C-String Definitionen
typedef char string80[81];                          // eigenen C-Stringtyp definieren
string80 s;                                          // C-String vom Typ string80
char text[500];                                    // C-String für max. 499 Zeichen
char name[21]="C-Strings" ;                        // C-String initialisieren
const char OK_MSG[3]="OK";                          // C-Stringkonstante

//  Hauptprogramm
int main()
{
    char s1[41],s2[41];                            // C-String als char-Array
    char trenn[5];                                  // C-Strings mit max. 4 Zeichen
    int pos,l,i;

    system("CLS");
    printf("\n %s ",name);                          // String ausgeben
    printf("\n\n Einlesen in String s1 : ");
    scanf("%s",s1);                                  // String einlesen
    printf("\n Einlesen in String s2 : ");
    scanf("%s",s2);

    strcpy(text,s1);                                // C-String kopieren mit strcpy
    printf("\n\n Strings kopieren mit strcpy : %s ",text);
    strcat(text,s2);                                // C-String verketteten mit strcat
    printf("\n Strings verketteten mit strcat : %s ",text);
    printf("\n neue Stringlaenge = %d ",strlen(text) );

    printf("\n\n Zeichen einzeln ausgeben : ");
    for (int i=0;i<strlen(text);i++)                // String zeichenweise ausgeben
    {
        putchar(text[i]);
    }

    printf("\n\n einzelne Zeichen aendern : ");
    text[pos]='*';
    text[pos+1]='*';
    printf("%s \n",text);

    printf ("\n\n String \"1234cd\" auf Zahl  umwandeln : ");
    strcpy(s,"1234cd");
    sscanf(s,"%4d",&i);                             // String s auf Format(%4d) umwandeln
    printf("%4d \n\n",i);

    strcpy(text,"C-Strings werden als char-Arrays definiert. \n");
    strcpy(s,"Strings");
    i=strcmp(text,s);                                // Stringvergleich von text mit s
    if (i==0) printf("\n  Strings sind gleich ");
    else      printf("\n  Strings ungleich - Returnwert = %4d ",i);

    system("PAUSE");
    return 0;
}

```

Strukturen (structs)

Eine Struktur ist ein Datentyp, der mehrere Variable unterschiedlicher Typen aufnehmen kann. Eine Struktur kann damit alle Variable, die sinngemäß zusammengehören, in einen Verbund zusammenfassen und so eine logische Datenorganisation erzielen. Komponenten einer Struktur können beliebige Variable, Felder und auch Strukturen sein.

Vereinbarung :

über Typnamen : **typedef struct structTypname**
 { Typ Komponente1;
 ...;
 };

 structTypname structVariable;

oder direkt : **struct structTypname** (typedef kann bei structs auch entfallen)
 { Typ Komponente1;
 ...;
 } structVariable;

```
Bsp.: typedef struct Person                   // Strukturtyp Person
      {   string name;                        // Komponente name als String
          int alter;                           // Komponente alter als int
      };
      Person p;                                // Strukturvariable p
      Person kartei[10];                      // Feld von Strukturen
```

```
Bsp.: gemeinsame Typ- und Strukturvariablen Definition
      struct Person                           // Strukturtyp Person
      {   string name;
          int alter;
      } p;                                      // Strukturvariable p
```

```
Bsp.: Strukturvariable ohne Typnamen
      struct
      {   string name;
          int alter;
      } p;                                      // Strukturvariable p
```

Die Struktur Person hat beispielsweise als Komponenten die Variablen `name` und `alter`, um die Daten einer Person aufzunehmen. Damit ergibt sich eine sinnvolle Datenorganisation, die eine Vielzahl von Einzeldaten logisch zusammenfassen und übersichtlicher verarbeiten lassen.

Zugriff auf die Komponenten über den Auswahloperator :

auf die Komponenten einer Struktur wird über den **Auswahloperator (.)** zugegriffen.

Auswahl einer Komponente (Qualifikation) : **structVariable.Komponente;**

```
Bsp.: p.alter = 20;
      cin >> p.name;
      cout << p.name;
```

Mit `p.alter` wird auf die Komponente `alter` der Strukturvariablen `p` zugegriffen. Der Typ von `p.alter` entspricht damit `int`, der Typ von `p.name` ist `string`, der von `p` ist hingegen `person`.

Zuweisung :

Die Zuweisung von Strukturvariablen gleichen Typs kann über den Zuweisungsoperator erfolgen. Damit werden die Werte aller Komponenten zugewiesen.

```
Bsp.: kartei[0] = p; // Strukturvariable zuweisen
```

Die Zuweisung der Komponenten kann natürlich auch einzeln erfolgen.

```
Bsp.: kartei[0].name = p.name;
      kartei[0].alter = p.alter;
```

Eingeschachtelte Strukturen :

Die Komponente einer Struktur kann auch eine weitere Struktur sein.

```
Bsp.: struct Adresse // Struktur Adresse
      { string name;
        int plz;
        string ort;
      };

      struct Person // Struktur Person
      { Adresse adr; // Struktur Adresse als Komponente
        int alter; // Komponente alter
      };

      struct Person p; // Strukturvariable p
      struct Adresse a; // Strukturvariable a
```

Der Zugriff auf die Komponenten einer eingeschachtelten Struktur ist über eine Mehrfach-Auswahl möglich.

```
Bsp.: p.adr.name="Stroustrup";
      p.adr.plz = 3100;
      p.adr = a;
```

Mit `p.adr.name` wird auf die Komponente `name` der Komponente `adr` zugegriffen. Der Typ von `p.adr.plz` ist `int`, der Typ von `p.adr` ist `Adresse`.

Strukturen als Parameter :

Strukturen als Parameter werden wie einfache Variable übergeben.

```
Bsp.: Funktion zum Ausgeben aller Komponenten von Person
      void pers_aus ( person p ) // mit person p als Parameter
      { cout << p.name << ", ";
        cout << p.alter;
      }

      pers_aus(p); // Aufruf mit Strukturvariablen p
```

Ausgabeparameter können als Referenz- oder Zeigerparameter ausgeführt werden. Auch für Eingabeparameter ist die Übergabe per Adresse bei größeren Strukturen ratsam und effizienter, da bei der Wertübergabe alle Komponenten auf den Stack kopiert werden. Bei einem Zeiger auf eine Struktur kann der Zugriff auf die Komponenten über den Pfeiloperator (`->`) erfolgen.

```
Bsp.: Funktion pers_ein mit Zeigerparameter
      void pers_ein ( person *p ) // mit Zeigerparameter *P
      { cout << " Name = " ; cin >> p->name; // auch (*p).name
        cout << " Alter = " ; cin >> p->alter; // auch (*p).alter
      };
```

Der Pfeiloperator bei `p->name` ersetzt dabei den Inhalts- u. den Auswahloperator `(*p).name`.

```

////////////////////////////////////
//  Programmname : struct.cpp      ( Personenkartei )
////////////////////////////////////

//  Header-Dateien
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <iostream>
using namespace std;

//  globale Struktur-Definition
struct person                               // Struktur Person
{  string name;                             // Komponente Name
  string adr;                               // Adresse
  int alter;                                // Alter
};

const int N=4;                             // Karteigröße
person kartei[N];                           // Strukturfeld kartei
int sort_name[N] ;                         // sortiertes Indexfeld

//  Funktions Prototypen
void pers_aus ( person p );                 // Ausgeben einer Person
void pers_namen_sort ( person *k, int index[], int n ); // Sortieren nach Namen

//  Hauptprogramm
int main()
{  int i;
  person p;

  system("CLS");
  cout << " Kartei erstellen ... " << endl;
  for (i=0;i<N;i++)                       // Kartei füllen
  {  p.name = n[i]; p.adr = a[i]; p.alter = i+20;
    kartei[i]=p;
  }

  pers_namen_sort(kartei,sort_name,N);      // nach Namen alph. sortieren

  cout << endl << " Kartei sortiert : ";
  for (i=0;i<N;i++)
  {  cout << endl << i+1 << ". Person : ";
    pers_aus(kartei[sort_name[i]]);        // Ausgabe über Indexfeld
  }
  ...
}

//  Funktions Definitionen
void pers_namen_sort(person *k, int index[], int n)
/*  sortiert die ersten n-Elemente der Kartei k
    und gibt die sortierte Reihenfolge über ein Schlüsselfeld (index) aus
*/
{  int i,j,h;
  for ( i=0;i<n;i++) index[i]=i;           // Indexfeld initialisieren
  for ( i=0;i<n;i++)                       // Bubble Sort der Schlüssel
  {  for ( j=i+1;j<n;j++)
    {  if (k[index[i]].name > k[index[j]].name)
      {  h=index[i];index[i]=index[j];index[j]=h; // Schlüssel tauschen
      }
    }
  }
}

```

Unions

Unions haben Ähnlichkeit zu Strukturen, legen jedoch die Komponenten übereinander immer an der gleichen Speicheradresse an.

Die typmäßig größte Komponente bestimmt damit auch die Speichergröße der Union.

Unions sind dann vorteilhaft einzusetzen, wenn jeweils nur eine Komponente wahlweise verwendet wird oder ein Type-Overlaying (Zugriff auf den gleichen Speicherbereich mit unterschiedlichen Datentypen) ausgeführt wird.

Vereinbarung : ***union unionTypname***
 { Typ Komponente1; ...
 } unionVariable ;

Bsp.: Typ Overlaying mit unions

(Zugriff auf den gleichen Speicherplatz mit verschiedenen Datentypen)

```
typedef unsigned char byte;           // Typdefinition für Byte

typedef union intbyte                 // Definition der union
{   int    i;
    byte  b[2];
};

union intbyte x;                      // union Variable x

// x kann sowohl als int als auch als Byte verarbeitet werden.

x.i = 255;                           // Komponente i einen int-Wert zuweisen
cout << x.b[0]                       // lower Byte von i ausgeben
cout << x.b[1]                       // higher Byte von i ausgeben
```

Bitfelder

Bitfelder sind Strukturen, die den Zugriff auf einzelne Bits erlauben.

Die Komponenten eines Bitfeldes müssen vom Typ `int` oder `unsigned int` sein.

Bsp.: Bitfeld

```
typedef unsigned short word;          // Typdefinition für word

struct bit                            // Bitfeld Definition
{   word  b0 : 1;                    // Komponente für 1 Bit
    word  b1 : 1;
    word  b2 : 1;
    word  b3 : 1;
    word  b4 : 1;
    word  b5 : 1;
    word  b6 : 1;
    word  b7 : 1;
    word  b8_15 : 8;                 // Komponente für 8 Bits
};

bit b={1,1,1,1,0,0,0,0,255};         // Bitfeld Variable b
```

Zugriff auf einzelne Bits : `b.b0 = 1; // Bit 0`
 `b.b1 = 0; // Bit 1`
 `...`
 `b.b8_15=0xFF; // higher Byte (Bit8 - Bit15)`

```

////////////////////////////////////
//  Programmname : union.cpp    ( Unions und Bitfelder )
////////////////////////////////////

//  Header-Dateien
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

//  globale Definitionen
typedef unsigned char byte;           // Typdefintion byte
typedef unsigned char word;          // Typdefintion word

typedef struct bit                    // Struktur als bitfeld
{                                     // einzelne Bits festlegen
    word b0 : 1;
    word b1 : 1;
    word b2 : 1;
    word b3 : 1;
    word b4 : 1;
    word b5 : 1;
    word b6 : 1;
    word b7 : 1;
};

union intbyte                        // Union für Type-Overlying
{                                     // int-Komponente
    int i;                           // Byte-Komponente
    byte x[2];                       // Bitfeld-Komponente
    bit b;
};

intbyte u;                           // Union Variable
bit b={1,1,1,1,1,0,0,0,0};          // Bitfeld Variable

//  Hauptprogramm
int main()
{
    system("CLS");
    printf(" Unions und Bitfelder \n");

    u.i=0xF;                          // Komponente i zuweisen

    printf("\n  Union Komponente i    = %d ", u.i); // Komponente i ausgeben
    printf("\n  Uniongroesse [Bytes] = %d ", sizeof(u));
    printf("\n  Union als Byte ausgeben : ");
    printf("\n  1.Byte   x[0]          = %d ",u.x[0]); // als Byte ausgeben
    printf("\n  2.Byte   x[1]          = %d ",u.x[1]);

    printf("\n  Komponente als Bitfeld ausgeben : "); // als Bits ausgeben
    printf("\n  Bits b.b7,...,b.b0    = %d%d%d%d%d%d%d%d ",
        u.b.b7,u.b.b6,u.b.b5,u.b.b4,u.b.b3,u.b.b2,u.b.b1,u.b.b0);

    cout << endl << endl;
    system("PAUSE");
    return 0;
}

```

Aufzählungstyp enum

Ein Aufzählungstyp (enumeration) enthält Elemente mit Namen, wie z.B. für Wochentage oder Farben. Intern werden die Elemente mit Zahlenwerten beginnend mit Null durchnummeriert.

Vereinbarung : **enum Typname**
 { Element1, Element2, ...
 };

Beispiel : Farbe

```
enum Farbe                                // Aufzählungstyp Farbe
{  ROT,GRUEN,BLAU,GELB                     // Elemente
};

Farbe f1,f2;                             // Variable zu Farbe
f1 = ROT;                                  // Wertzuweisung
f2 = GELB;

cout << endl << " Farbe 1 = " << f1;      // gibt den Zahlenwert 0 aus
if ( f2==GELB )
    cout << endl << " Farbe 2 ist Gelb ";
```

Die automatisch zugewiesenen Zahlenwerte können bei Bedarf auch anders belegt werden.

Beispiel :

```
enum Farbe                                // Aufzählungstyp Farbe
{  ROT=1,GRUEN=2,BLAU=6,GELB=8             // Zahlenwerte festlegen
};
```

Dynamische Variable

Variable können auch während der Programmausführung dynamisch angelegt werden. Damit können z.B. Felder mit der im Programmablauf aktuell benötigten Größe erstellt werden.

Dynamische Speicherplatzreservierung in C++ :

allgemein : Typ *Zeiger;
 Zeiger = new Typ; **// für einfache Variable**
oder : Zeiger = new Typ[n];**// für dynamisches Array**

Freigabe : **delete Zeiger;** **// für einfache Variable**
 oder : delete[] Zeiger; **// für dynamisches Array**

Zur dynamischen Speicherplatzreservierung wird zuerst ein Zeiger mit dem Zeigertyp der gewünschten Variable definiert und dann mit new ein Speicherplatz angefordert. Die Typangabe nach new gibt dabei die Größe des notwendigen Speicherplatzes an, der Zeiger der von new zurückgegeben wird enthält die (Basis-) Adresse des allokierten Speicherplatzes.

Beispiel : dynamisches Array

```
int n=100;                                // Variable n für Feldgröße

int *pa;                                  // Zeiger für dynamische Variable
pa = new int[n];                          // dynamisches int-Array mit n-Elementen

for (i=0;i<n;i++)
{  pa[i]=i+1;                             // Zugriff auf die Feldelemente
}
delete[] pa;                              // dynamisches Feld freigeben
```


Dateiverarbeitung

Um Daten dauerhaft zu speichern, können sie auf eine Datei (File) ausgegeben werden.

Die Dateiverarbeitung erfolgt in C über die „f-functions“ der Bibliothek „stdio.h“ und in C++ über die Methoden der „fstream“- Klassenbibliothek.

Es besteht prinzipiell die Möglichkeit, Dateien entweder im Text- oder im Binärformat zu erstellen.

Textdateien speichern Daten im ASCII-Code und sind mit einem Texteditor les- u. bearbeitbar.

Binärdateien hingegen speichern die Daten im Binärcode des jeweiligen Variablentyps und sind bei großen Datenmengen, wie z.B. bei Grafikdateien, kompakter und effizienter.

Der **Dateizugriff** kann auf zwei Arten erfolgen, entweder sequentiell oder wahlfrei.

Bei **sequentiellm Zugriff** werden die Daten beginnend mit dem ersten Element nacheinander bis zum Dateiende (EOF = End Of File) abgearbeitet.

Bei **wahlfreiem Zugriff** (random access) wird auf einzelne Datensätze an bestimmten Positionen der Datei zugegriffen, wie es bei großen Datenbanken notwendig ist.

Die Übertragung der Daten zur Datei wird über einen internen **Datenpuffer** abgewickelt, der als Zwischenspeicher dient und eine blockweise Übertragung zum Laufwerk ermöglicht.

Die **Programmausführung** für einen Dateizugriff erfolgt in **3 Schritten** :

- 1) Datei öffnen : Verbindung zur Datei herstellen
- 2) Dateizugriff : lesen, schreiben, anhängen
- 3) Datei schließen : Verbindung zur Datei beenden

Dateiverarbeitung in C :

Datei öffnen : die Bibliotheksfunktion **fopen** (aus <stdio.h>) öffnet eine Datei und gibt einen Zeiger auf den Typ „FILE“ zurück, über den die weitere Abwicklung des Dateizugriffes erfolgt.

FILE* fopen (char *path, char *mode);

FILE* Dateizeiger (File Handle)

FILE ist eine Struktur in <stdio.h> und enthält Informationen für die I/O-Routinen, wie Pufferadresse, Puffergröße, Position des Schreib-/Lesezeigers, ...

Im Fehlerfall wird die Zeigerkonstante "NULL" zurückgeliefert.

path Pfadname der Datei

bei DOS-Pfadangaben ist der "\" im C-String durch "\\" anzugeben ist!

mode Zugriffsmodus (Modus-String)

"r" read (lesen) Datei zum Lesen öffnen

"w" write (schreiben) Datei neu anlegen und schreiben

"a" append (anhängen) zum Anhängen und Schreiben öffnen

"r+" lesen u. schreiben zum Lesen und Schreiben öffnen

"w+" lesen u. schreiben Datei neu anlegen und schreiben

"t" Textformat Konvertierung des Zeilenendezeichens (\n)

"b" Binärformat

kombinierte Modusangabe möglich, wie "rt", "rb", "w+b", ...

vordefinierte Standard-Filepointer :

stdin	Standardeingabe	(Tastatur)
stdout	Standardausgabe	(Bildschirm)
stderr	Fehlerausgabe	(Bildschirm)

Beispiel : Datei öffnen

```
FILE *datei; // File Pointer definieren
datei = fopen("C:\\daten.txt", "w"); // Datei zum Schreiben öffnen
if (datei==NULL) // Fehler beim Öffnen?
{ fprintf(stderr, "Datei kann nicht geöffnet werden");
  exit(1); // dann Programmabbruch
}
```

Datei schließen :

die Bibliotheksfunktion **fclose** schließt eine Datei.

int fclose(FILE *stream);

Beispiel : Datei schließen

```
fclose(datei); // Datei schließen
```

Dateizugriff auf Textdateien

Textdateien werden üblicherweise sequentiell vom Dateianfang bis zum Dateiende abgearbeitet. Die C-Bibliotheksfunktionen von `<stdio.h>` erlauben zeichenweises, zeilenweises, formatiertes und auch blockweises Schreiben und Lesen.

zeichenweise Lesen : int fgetc (FILE *stream);

fgetc liest ein Zeichen vom Datenstrom (stream) und gibt das Zeichen als Rückgabewert aus, bei Dateiende wird EOF (End Of File) zurückgegeben.

zeichenweise Schreiben : int fputc (int c, FILE *stream);

fputc schreibt ein Zeichen auf den Datenstrom und gibt im Fehlerfall als Rückgabewert EOF aus.

Beispiel : Textdatei schreiben und lesen

```
FILE *datei; // File Pointer definieren
char zeichen;

datei=fopen(dateiname, "w"); // Datei zum Schreiben öffnen
if(datei==NULL) return 1; // Fehlerbehandlung
do
{ zeichen=getchar(); // Zeichen von Tastatur einlesen
  fputc(zeichen, datei); // Zeichen auf Datei schreiben
} while (zeichen!='. '); // wiederholen bis '.'
fclose(datei); // Datei schließen

datei=fopen(dateiname, "r"); // Datei zum Lesen öffnen
if(datei==NULL) return 1; // Fehlerbehandlung
while ((zeichen=fgetc(datei))!=EOF) // bis Dateiende zeichenweise lesen
{ putchar(zeichen); // Zeichen am Bildschirm ausgeben
}

fclose(datei); // Datei schließen
```

Beispiel : Zeichen und Zeilen einer Textdatei zählen

```
#define LF 10 // Ersatztext für Steuerzeichen LF

char c;
int nc,nl; // Anzahl der Zeichen u. Zeilen
FILE *datei; // Filepointer

datei=fopen(dateiname,"r"); // Datei zum Lesen öffnen
if(datei==NULL) exit(1); // Fehlerbehandlung

nc=nl=0; // mit 0 zu zählen beginnen
while ((c=fgetc(datei))!=EOF) // bis Dateiende zeichenweise lesen
{
    nc++; // Zeichen mitzählen
    if(c==LF) // beim Zeichen Line Feed (LF)
    {
        nl++; // Zeilen mitzählen
    }
}
fclose(datei); // Datei schließen
```

zeilenweise Lesen : char *fgets(char *s, int n, FILE *stream);

fgets liest bis Zeilenende maximal n-Zeichen von Datei in den String s und gibt bei Dateiende den Rückgabewert NULL aus.

zeilenweise Schreiben : int fputs(const char *s, FILE *stream);

fputs schreibt den String s auf Datei und gibt im Fehlerfall den Rückgabewert EOF aus.

Beispiel : Textdatei zeilenweise lesen und schreiben

```
FILE *datei;
char zeile[81]; // Stringvariable

datei=fopen(dateiname,"r"); // Datei zum Lesen öffnen
if(datei==NULL) exit(1); // Fehlerbehandlung
while(fgets(zeile,80,datei)) // zeilenweise von Datei lesen
{
    printf("%s",zeile); // und am Bildschirm ausgeben
}
fclose(datei);

datei=fopen(dateiname,"a"); // Datei zum Anhängen öffnen
if(datei==NULL) exit(1); // Fehlerbehandlung
printf("\n Zeilen auf Datei schreiben : \n");
for (int l=1;l<=5;l++)
{
    printf("%d:",l);
    fgets(zeile,80,stdin); // Zeile über Tastatur einlesen
    fputs(zeile,datei); // Zeile auf Datei schreiben
}
fclose(datei);
```

Beispiel : Textdatei nach einem bestimmten Text durchsuchen

```
datei=fopen(dateiname,"r"); // Datei zum Lesen öffnen
if(datei==NULL) return 1; // Fehlerbehandlung
while(fgets(zeile,80,datei)) // zeilenweise von Datei lesen
{
    if(strstr(zeile,suchmuster)!= 0) // Stringmuster suchen
    {
        printf("%d:%s",zeile); // und am Bildschirm ausgeben
    }
}
fclose(datei); // Datei schließen
```

```

////////////////////////////////////
//   Programmname : fputs.cpp
////////////////////////////////////

//   Header-Dateien
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
using namespace std;

//   Funktionen
int datei(char *dateiname) // neue Datei anlegen
{
    FILE *datei; // File Descriptor
    datei=fopen(dateiname,"w"); // Datei erstellen
    if(datei==NULL) // Fehlerbehandlung
    {
        return EXIT_FAILURE;
    }
    fclose(datei); // Datei schließen
    return 0;
}

int zeileschreiben(char *dateiname, char *zeile ) // Zeile auf Datei schreiben
{
    FILE *datei; // File Descriptor
    datei=fopen(dateiname,"a"); // Datei zum Anhängen öffnen
    if(datei==NULL) return 1; // Fehlerbehandlung
    fputs(zeile,datei); // zeilenweise auf Datei schreiben
    fclose(datei); // Datei schließen
    return 0;
}

int zeilelesen(char *dateiname, char *zeile, int l) // die Zeile l von Datei lesen
{
    char s[80]="";
    FILE *datei; // File Descriptor definieren
    datei=fopen(dateiname,"r"); // Datei zum Lesen öffnen
    if(datei==NULL) return 1; // Fehlerbehandlung
    for( int i=1;i<=l;i++) // bis zur Zeile l gehen
    {
        if (fgets(s,80,datei)==0) return 2; // falls Dateiende dann Abbruch
    }
    strcpy(zeile,s); // Zeile ausgeben
    fclose(datei); // Datei schließen
    return 0;
}

//   Hauptprogramm
int main()
{
    char dateiname[41]="text.txt"; // Dateiname
    char zeile[80]; // String für eine Zeile
    int err,l;

    printf("\n Dateiname = %s ",dateiname);
    if (datei(dateiname)) exit(1); // neue Datei anlegen

    printf("\n 5 Zeilen auf Datei schreiben : \n");
    for (l=1;l<=5;l++)
    {
        fgets(zeile,80,stdin); // Zeile über Tastatur einlesen
        zeileschreiben(dateiname,zeile); // und auf Datei ausgeben
    }
    printf("\n welche Zeile lesen : "); scanf("%d",&l);
    err=zeilelesen(dateiname,zeile,l); // eine Zeile lesen
    if (err==2) printf("\n Dateiende erreicht \n");
    else puts(zeile);

    printf("\n\n"); system("PAUSE");
}

```

formatiertes Lesen u. Schreiben :

lesen : int fscanf (FILE *stream, const char *format, ...);

schreiben : int fprintf (FILE *stream, const char *format, ...);

Über die Funktionen fscanf u. fprintf kann durch Angabe eines Formatstrings (wie bei scanf u. printf) formatiert von Datei gelesen oder auf Datei geschrieben werden.

Der Rückgabewert liefert die Anzahl der fehlerfrei gelesenen oder geschriebenen Elemente, bei Dateiende oder im Fehlerfall wird EOF zurückgegeben.

Beispiel : CSV-Datei lesen

```
////////////////////////////////////
//   Programmname : csv.cpp
//   CSV (comma seporate value) - durch Komma getrennte Werte
////////////////////////////////////

//-----
//   Header-Dateien
#include <stdio.h>
#include <stdlib.h>

//-----
//   Hauptprogramm
int main(int argc, char *argv[])
/* Konsolen-Aufruf mit Parameter : csv dateiname
   zum Austesten können die Parameter auch in der DevCpp-IDE
   unter Menüpunkt "Ausführen -> Parameter" übergeben werden.
*/
{
    FILE *datei;
    int i;
    float x;
    char text[40];

    printf("\n CSV-Datei einlesen \n");
    if ( argc<=1 )                // Aufruf-Parameter fehlt?
    { printf("\n Aufruf : csv [Datei] \n");
      return 1;                  // dann Abbruch
    }

    datei = fopen(argv[1],"r");    // Dateiname über Aufrufparameter holen
    if( datei==NULL)
    { printf("Fehler beim Oeffnen ...\n");
      return 1;
    }

    while((fscanf(datei,"%d,%f,%s\n",&i,&x,text))!=EOF)    // formatiert lesen
        fprintf(stdout," i=%d  x=%f  Text=%s \n",i,x,text); // und ausgeben

    system("PAUSE");
    return 0;
}
```

Der Inhalt der CSV-Testdatei wird über einen Texteditor erstellt:

```
1,0.1,eins
2,0.2,zwei
3,0.3,drei
4,0.4,vier
5,0.5,fuenf
```

Dateizugriff auf Binärdateien

Über die Funktionen „fread“ und „fwrite“ können ganze Datenblöcke im binären Format von Datei gelesen oder auf Datei geschrieben werden.

Lesen : `int fread (void *buffer, size_t size, size_t num, FILE *stream);`
Schreiben : `int fwrite (const void *buffer, size_t size, size_t count, FILE *stream);`

Bedeutung der Parameter :

buffer	Zeiger auf den Datenpuffer (der Typ void* kann auf beliebige Datentypen zeigen)
size	Größe eines Datenelementes in Bytes (wird mit sizeof () ermittelt)
n	Anzahl der Datenelemente
stream	File Pointer

insgesamt werden n-Datenlemente (n*size Bytes) gelesen oder geschrieben,
der Rückgabewert liefert die Anzahl der tatsächlich gelesenen oder geschriebenen Datenelemente.

wahlfreier Zugriff (random access) :

Die Funktion „fseek“ setzt den Dateizeiger an eine bestimmte Position und erlaubt damit den Zugriff auf Datensätze an bestimmten Positionen.

Positionieren : `int fseek (FILE *stream, long offset, int origin)`

Bedeutung der Parameter :

stream	File Pointer
offset	Position in Bytes als Offset (Differenz) zum Ursprung (origin)
origin	Ursprung
	0 = SEEK_SET (Dateianfang)
	1 = SEEK_END (Dateiende)
	2 = SEEK_CUR (aktuelle Position)

Der Rückgabewert liefert 0 bei fehlerfreier Ausführung, im Fehlerfall ungleich 0.

weitere Bibliotheksfunktionen aus <stdio.h> :

```
int feof( FILE *stream);           // liefert true bei Dateiende
int ferror( FILE *stream);         // liefert true bei einem Fehler
long ftell(FILE *stream);          // Position des Dateizeigers
void rewind(FILE *stream);         // Dateizeiger auf Anfang setzen
int fflush(FILE *stream);          // Ausgabepuffer leeren

void setbuf(FILE *stream, char *buf); // Dateipuffer festlegen
                                   // buf=0 -> ungepuffert

int setvbuf( FILE *stream, char *buffer, int mode, size_t size );
                                   // Dateipuffergröße u. -modus festlegen

int remove( const char *fname );   // Datei löschen
int rename(const char *oldname, const char *newname); // Datei umbenennen

FILE *tmpfile( void );             // temporäre Datei öffnen
```

Beispiel : Dateizeiger auf Dateianfang und -ende setzen

```

fp=fopen(dateiname,"r");
fseek(fp,0,SEEK_SET);           // auf Dateianfang setzen
fseek(fp,0,SEEK_END);           // auf Dateiende setzen

```

Beispiel : Dateizeiger auf bestimmte Position setzen

```

int error;
error = fseek(fp,i*sizeof(int),0); // Dateizeiger auf i-tes Datenelement
if (error) return -1;              // wenn Positionierfehler dann Abbruch

```

Beispiel : Dateilänge über fseek u. ftell bestimmen

```

long filelen(FILE *fp)
// Dateilänge in Bytes ermitteln
// Return : long ... Dateigröße in Bytes
// Dateizeiger auf Dateiende setzen
{ if (fseek(fp,0,SEEK_END))
  return -1;
  return ftell(fp); // und aktuelle Dateiposition ermitteln
}

```

Beispiel : Array als Datenblock schreiben und lesen

```

int a[10]; // Array als Datenblock
f1=fopen(datei1,"w");
f2=fopen(datei2,"r");
fwrite(a,sizeof(int),10,f1); // 10 Elemente auf Datei schreiben
fread(a,sizeof(int),10,f2);  // 10 Elemente von Datei lesen

```

Beispiel : bestimmte Datenblöcke lesen und schreiben

```

int binaerschreiben(FILE *fp, int a[], int pos, int n)
// Integerdaten auf Datei binaer schreiben
// Import : fp ... Filepointer
//          a[] ... Datenfeld
//          pos ... Position des Dateizeigers
//          n ... Anzahl der Datenelemente
// Return : int ... Anzahl tatsächlich geschriebener Elemente
{
  if (fseek(fp,pos*sizeof(int),0)) // Dateizeiger auf pos setzen
    return -1;
  n=fwrite(a,sizeof(int),n,fp); // n-Elemente vom Typ int schreiben
  return n; // Anzahl geschriebener El. ausgeben
}

```

int binaerlesen(FILE *fp, int a[], int pos, int n)

```

// Integerdaten von Datei binaer lesen
// Import : fp ... Filepointer
//          pos ... Position des Dateizeigers
//          n ... Anzahl der Datenelemente
// Export : a[] ... Datenfeld
// Return : int ... Anzahl tatsächlich gelesener Elemente
{
  if (fseek(fp,pos*sizeof(int),0)) // Dateizeiger positionieren
    return -1;
  n=fread(a,sizeof(int),n,fp); // n-Elemente vom Typ int lesen
  return n; // Anzahl gelesener El. ausgeben
}

```

Beispiel : Personenkartei mit wahlfreiem Dateizugriff

```
////////////////////////////////////
//  Programmname : kartei.cpp
//  Karteisystem
////////////////////////////////////

//----- Struktur Person -----//
struct person                                // Struktur Person als Datensatz
{ char name[41];                               // Komponente Name
  char adr[41];                                // Adresse
  char ort[41];                                // Ort
  int plz;                                     // Postleitzahl
};

//----- Funktions Definitionen -----
int kartei_zugriff(char *file, person &b, int pos, int mode)
////////////////////////////////////
// Kartei Zugriff
// Import : person &b ... Struktur person
//          pos      ... Dateizeiger Position
//          mode     ... 0=lesen, 1=zurückschreiben,
//                      2=überschreiben, 3=anhängen
// Export : person &b ... Struktur person
// Return : int      ... 0=fehlerfrei
////////////////////////////////////
{ FILE *d;                                     // File Pointer
  int n;
  int l= sizeof(person);                       // Strukturgröße

  switch (mode)                                // Öffnen für
  { case 0 : d=fopen(file,"rb");break;          // lesen
    case 1 : d=fopen(file,"r+b");break;        // zurückschreiben
    case 2 : d=fopen(file,"wb");break;         // überschreiben
    case 3 : d=fopen(file,"ab");break;         // anhängen
    default : return 1;                        // sonst Abbruch
  }
  if (d==NULL) return 2;                       // Fehler bei Öffnen?
  switch (mode)                                // Positionieren
  { case 0 :
    case 1 :
    case 2 : n=fseek(d,pos*l,SEEK_SET);break;
    case 3 : n=fseek(d,0,SEEK_END);break;
  }
  if (n)                                        // Fehler bei Positionieren?
  { fclose(d); return 3;
  }
  switch (mode)                                // Schreiben/Lesen
  { case 0 : n=fread(&b,l,1,d);break;
    case 1 :
    case 2 :
    case 3 : n=fwrite(&b,l,1,d);break;
  }
  if (!n)                                      // Schreib-/Lesefehler?
  { fclose(d); return 4;
  }
  fclose(d);
  return 0;                                    // fehlerfrei
}

...
```


Dateiverarbeitung in C++ :

In C++ kann der Dateizugriff über die Methoden der <fstream> - Klassenbibliothek erfolgen. Zuerst benötigt man eine Variable der Klasse „fstream“ (File-Stream = Datenstrom), (oder auch „ofstream“ für nur schreiben und „ifstream“ für nur lesen) und dann kann über die Methode „open“ die Datei geöffnet werden.

Beispiel : Dateizugriff über fstream-Klasse

```
// Header-Dateien
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    fstream f1; // Variable zur Klasse fstream
    f1.open("test.dat", ios::out); // Datei schreibend öffnen
    if (!f1) // Fehlerabfrage
    { cerr << " Fehler beim öffnen ";
      exit(-1);
    }

    f1 << " Streams Streams Streams " << endl; // Text auf Datei schreiben
    f1 << " noch eine Zeile " << endl;
    f1.close(); // Datei schließen
    ...
}
```

Auf die Variablen der Streamklassen können auch die Ein- u. Ausgabeoperatoren „<<“ und „>>“ angewendet werden. Mit der Methode „close“ wird der Datenstrom dann geschlossen. Über die Methoden „get()“ und „getline()“ kann zeichenweise und zeilenweise gelesen werden.

```
cout << endl << " Datei zeichenweise lesen : " << endl;
f2.open("test.dat", ios::in); // Datei lesend öffnen
while(!f2.eof()) // bis zum Dateiende
{ f2.get(c); // zeichenweise lesen
  cout << c;
}
f2.close(); // Datei schließen

cout << endl << " Datei zeilenweise lesen : " << endl;
f3.open("test.dat", ios::in); // Datei lesend öffnen
while(!f3.eof()) // bis zum Dateiende
{ getline(f3,s); // zeilenweise lesen
  cout << s << endl;
}
f3.close(); // Datei schließen
}
```

Beispiel : von Quelldatei auf Zieldatei kopieren

```
char c;
ifstream f1; // Variable zur Klasse ifstream
ofstream f2; // Variable zur Klasse ofstream

f1.open("test.dat"); // Quelldatei öffnen
f2.open("testcopy.dat"); // Zieldatei öffnen

while(f1.get(c)) // lesen bis Dateiende
{ f2.put(c); // schreiben
}
f1.close(); // Dateien schließen
f2.close();
```

Die Operatoren „<<“ und „>>“ arbeiten mit fstream-Variablen wie mit cin u. cout. Es können dabei auch **numerische Variable in den Datenstrom** gebracht werden. Als Trennzeichen muss dabei entweder das Leerzeichen, das Tabulator- oder Zeilenvorschubzeichen verwendet werden.

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//  Zahlen auf Datei streamen
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//-----
//  Header-Dateien
#include <iostream>
#include <fstream>
using namespace std;

//-----
//  Hauptprogramm
int main()
{
    int i,j;
    cout << endl << " Integer streamen " << endl;

    fstream f;                                // Variable zur Klasse fstream

    f.open("data.dat",ios::in |                 // Datei für Lesen,
           ios::out |                           // Schreiben
           ios::trunc );                       // u. neu Erstellen öffnen

    if (!f)
    { cerr << " Fehler beim öffnen " << endl;
      exit(-1);
    }

    cout << endl << " Zahlen auf Datei schreiben ... " << endl;
    for (i=0;i<10;i++)
    { f << i << ' ';                            // Variable i auf Datei schreiben
      }                                           // Leerzeichen ist zwingend !
    f << endl;

    cout << endl << " Dateiinhalt : " << endl;
    f.seekg(0);                                // Dateizeiger auf Anfang setzen
    while(1)
    { f >> j;                                    // von Datei in Variable j einlesen
      if (f.eof()) break;                       // Abbruch bei Dateende
      cout << j << ',';
    }
    cout << endl;
    f.close();

    cout << endl << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Die Modusangabe beim Öffnen kann auch über die bitweise Oder Verknüpfung kombiniert werden.

Modusangaben und Bedeutung :

ios::in	zum Lesen öffnen
ios::out	zum Schreiben öffnen
ios::app	zum Anhängen öffnen
ios::trunc	den Inhalt der Datei zuvor löschen
ios::binary	binärer Zugriff (keine Umwandlung der Zeilenendezeichen)
ios::ate	Dateizeiger an das Dateende setzen

Über die Methoden „write“ und „read“ können Datenblöcke binär geschrieben und gelesen werden. Die Positionierung des Dateizeigers erfolgt im ifstream über „seekg“ und im ofstream über „seekp“

Beispiel : Array als Datenblock schreiben und lesen

```
////////////////////////////////////
//  Binärdatei Zugriff über Bytestreams
////////////////////////////////////

//-----
//  Header-Dateien
#include <iostream>
#include <sstream>                // für ostringstream
#include <fstream>
using namespace std;

//-----
//  Hauptprogramm
int main()
{
    int a[10]={1,2,3,4,5,6,7,8,9,10};
    int b[10];

    cout << endl << " Bytestreams  " << endl;
    fstream f;                                // Variable zur Klasse fstream
    f.open("bindata.dat",ios::out | ios::binary ); // Öffnen für binäres Schreiben
    if (!f)
    {
        cerr << " Fehler beim Öffnen " << endl;
        exit(-1);
    }

    cout << endl << " Array auf Datei schreiben ... " << endl;
    f.write((char*)a,10*sizeof(int));           // Array auf Datei schreiben
    f.close();

    f.open("bindata.dat",ios::in | ios::binary ); // Öffnen für binäres Lesen
    cout << endl << " Array von Datei lesen :  " ;
    f.read((char*)b,10*sizeof(int));           // Array von Datei lesen

    for (int i=0;i<10;i++)
    {
        cout << b[i] << ", " ;
    }
    cout << endl << endl << " wahlfreier Zugriff  ";
    int pos,i;
    do
    {
        cout << endl << " Position = " ;
        cin >> pos;
        f.seekg(pos*sizeof(int));                // Dateizeiger setzen
        f.read((char*)&i,sizeof(int));           // Integerwert von Datei lesen
        if (f.fail()) break;                     // Abbruch bei Fehler
        cout << " Wert= " << i;
    } while (pos>=0);
    f.close();

    cout << endl << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```