C++ Einführung

Bemerkung: Die Beispiele wurden mit dem C-Compiler Dev-Cpp 4.9.9.2 erstellt.

1 Mein erstes Programm

Beispiel 1: Dieses Programm schreibt „Hallo Welt“ auf den Bildschirm:

```
/*********************************************************************
*    Bill Gates / 1A / 1
*    HELLO.CPP  Erstes C Programm ("Hello World")  ***
*   04.09.2006 – Version 1
*********************************************************************/
```

} „**Kommentarkopf**“

```
# include <iostream>                    // Eine Bibliothek für Ausgabe cout und printf()

int main ( )
{
      cout << “Hallo Welt“;             // cout gibt einen Text aus
      printf(" Hello World ");          // printf gibt auch einen Text aus
      return 0;                         // Ende des Hauptprogramms main( )
}
```

1.1 Programmteile:

- **Kommentarkopf** : besteht immer aus:
  - **Name / Klasse / Katalognummer**
  - **Titel**
  - **Erstellungsdatum und Version**

Kommentar-Block :
```
/*   lall
     bla ……
*/
```
Kommentar für eine Zeile : // blablabla (d.h. mit 2 Schrägstrichen)

- **# include <iostream>** **Header-Dateien** werden in den Quellcode eingefügt (inkludiert).

- **int main ( )** Start des Programms mit dem Programm-Aufruf "**main**".
  Das Hauptprogramm "**main**" darf nur einmal vorkommen.

- **Lesbarkeit der Programme:**

```
{ ..........  ;
  ..........  ;

}
```

Alle zusammenhängenden Zeilen werden innerhalb von geschwungenen Klammern zusammengefasst.
Fast jede Zeile muss mit einem ";" abgeschlossen werden.

Die Klammern stehen immer UNTEREINANDER !!!
Zusammenhängende Blöcke werden eingerückt !!!

Aufgaben des Compilers:

- formale Überprüfung des Codes (Syntax-Check)
- Umwandlung des Quellcodes (C-Sourcecode) in eine Maschinensprache, welche vom Prozessor verstanden wird. Daraus entsteht ein ablauffähiges Programm (EXE-Datei).

Übungen:

```
// Übung: Rufe das Programm 3x hintereinander auf
// Übung: Blende '#include <conio2.h>' mit // aus
// Übung: Entferne einen ;  bei cout<<...
// Übung: Entferne ein " bei printf(...
// Übung: Entferne int vor main()
```

und compiliere jeweils!

1.2 Einfache Ein- und Ausgabe:

1.2.1 Ausgabe auf Bildschirm "cout"

(Benötigt **#include <iostream>**)

```
cout << "abc" ;          // weist dem Computer an, anschließende Daten "abc" mit
                         // dem Standard Ausgabegerät zu verknüpfen.
cout << 1 << 0 << 1;     // Es werden die Zahlen 101 hintereinander auf den
                         Bildschirm ausgegeben.
cout << "die Zahl ist " << 101;   // Text und Zahl gemischt
```

1.2.2 Einlesen von der Tastatur „cin"

(Benötigt **#include <iostream>**)

```
cin >> Variable ;        // Eingabe von Variablen oder Buchstaben über die Tastatur.
cin >> Var >> Zahl ;     // Eingabe von 2 Zahlen. (abschliessen jeweils mit <ENTER>)
```

1.2.3 Beispiel

```
#include<iostream>

int main ( )
{   int ErsteVar, ZweiteVar = 3, Summe ;      // Deklaration der Variablen
    cout << „Geben Sie die erste Variable ein" ;
    cin >> ErsteVar ;                         // Eingabe der ersten Zahl
    Summe = ErsteVar + ZweiteVar ;            // Berechnung
    cout << "Ergebnis = " << Summe ;          // Ausgabe auf den Bildschirm
    return 0;
}
```

1.2.4 Ausgabe auf Bildschirm mit "printf( )"

(Benötigt **#include <iostream>**)

printf("Ich bin der Groesste!!! "); // Ausgabe am Bildschirm

int zahl = 12;
printf("Die Zahl lautet %d",zahl); //Ausgabe: Die Zahl lautet 12 (int-Variable *zahl*=12)

1.2.5 Einlesen von der Tastatur mit "scanf( )"

(Benötigt **#include <iostream>**)

scanf("%d", &zahl) ; //Eingabe einer int-Variable *zahl* **(Achtung auf das & !!!)**

scanf("%d,%d", &zahl1, &zahl2) ; //Eingabe von 2 int-Variablen *zahl1* und *zahl2*

1.2.6 Weitere Bildschirm-Befehle

Wichtige Voraussetzungen:

1. „Conio-Paket" muss auf dem Rechner installiert sein **(conio-2.0-1mol.DevPak)**
2. Header-Datei **#include <conio2.h>** verwenden
3. Einbinden der Datei (Bibliothek) "**libconio.a**" bei **jedem** Projekt mit:

   Projekt->Projekt Optionen -> Parameter -> Linker -> Bibliothek/Objekt hinzufügen

- **gotoxy(10,20);** //setzt den Cursor hier an die x-Position 10 (Spalte) und y-Position 20 (Zeile)
- **getch( );** // wartet auf einen Tastendruck um im Programm fortzufahren
- **clrscr( );** // löscht den Bildschirm
- **clreol( );** // löscht von der aktuellen Cursorposition bis zum Zeilenende
- **textcolor(BLUE);** // ändert die Schriftfarbe auf blau (alle verfügbaren Farben siehe conio2.h)
- **textbackground(RED);** // ändert die Hintergrundfarbe des Textes auf rot
- **textattr(16*RED+YELLOW);** // ändert die Hintergrundfarbe auf rot und die Schriftfarbe auf gelb

2 Grundlagen zu C-Programmen

2.1 Reservierte (geschützte) Worte

Diese Namen dürfen **nicht** als Variablen- oder Konstantennamen (Bezeichner) verwendet werden.

asm	auto	break	case	catch	char
class	const	continue	default	delete	do
double	else	enum	extern	float	for
friend	goto	if	inline	int	long
new	operator	private	protected	public	register
return	short	signed	sizeof	static	struct
switch	template	this	throw	try	typedef
union	unsigned	virtual	void	volatile	while

2.2 Bezeichner

Bezeichner sind Namen für Konstanten, Variablen, Funktionen, Strukturen usw.

Regeln:

- Beliebige Länge
- Nur Buchstaben (A-Z, a-z), Ziffern (0-9) und Unterstrich (_)
- Nicht mit Ziffer beginnen
- Es wird zwischen GROSS- und klein-Schreibung unterschieden (d.h abC und AbC sind 2 verschiedene Bezeichner)
- Keine reservierten Wörter

2.3 Anweisungen

Ein Programm besteht aus vielen Anweisungen, die in einer bestimmten Reihenfolge ausgeführt werden. Jede Anweisung stellt einen Teil der Lösung eines Problems (d.h. der Programmieraufgabe) dar.

Jede Anweisung muss mit einem Strichpunkt (**;**) abgeschlossen werden !!!

z.B. **Zahl = 1;**
printf("Ich bin eine Anweisung");

2.4 Kommentare

- Kommentar bis Zeilenende: mit „//“ **z.B.** **// Das ist ein Kommentar**
- Blockweise

```
/*  Das ist ein
        weiterer   Kommentar*/
```

Merke: Kommentare beeinflussen das Programm in seiner Ausführung nicht !

3 Variablen

Man benötigt Variablen, um Werte zu speichern. Eine Variable muss daher einen **Variablennamen** und einen **physikalischen Speicherplatz** besitzen. Die Größe des Speicherplatzes hängt vom Typ der Variablen ab. Den **Variablennamen** kann der Programmierer vergeben. Der Computer legt selbstständig fest, auf welchen Speicherbereich die Daten abgelegt werden.
Man kann sich den Speicher im Computer wie einen Kasten vorstellen mit verschiedenen Laden. Jede Lade hat eine bestimmte Größe (abhängig vom *Typ*) und besitzt eine Beschriftung (*Variablennamen*). Der Inhalt der Lade entspricht dabei dem Wert der Variablen. Damit kann aber Inhalt der Lade entnommen, manipuliert und wieder zurückgelegt werden.

Globale – Lokale Variablen:
Variablen, die vor main() deklariert wurden, heißen **globale Variablen.** Sie können überall im gesamten Programm gelesen und modifiziert werden, daher sollte man sie möglichst meiden. Variablen, die innerhalb einer Funktion (auch in main()) deklariert wurden, heißen **lokale Variablen.** Sie sind nur innerhalb der Funktion ansprechbar. In anderen Funktionen kann es Variable mit dem gleichen Namen geben, die eine ganz andere Aufgabe erfüllen.

Beispiel:

```
int Z;                  // Z ist eine globale Variable
int main ( )
{
        int X;          // X ist eine lokale Variable  (gilt nur innerhalb von main ( ) )
        Z = 10;
        X=0;
}
```

3.1 Datentypen von Variablen:

Der Datentyp einer Variablen legt die Grösse , welche die Variable im Speicher belegt sowie deren Wertebereich fest.

Datentyp	Schlüssel-wort	Speicher-bedarf	Wertebereich	Genauigkeit
Zeichen	**(signed)char**	1 Byte	-128 bis 127	ganzzahlig
Zeichen	**unsigned char**	1 Byte	0 bis 255	ganzzahlig
Integer	**(signed) int**	4 Byte	-2147483648 bis 2147483647	ganzzahlig
Integer	**unsigned int**	4 Byte	0 bis 4294967295	ganzzahlig
Integer	**short**	2 Byte	-32768 bis 32767	ganzzahlig
Integer	**(signed) long**	4 Byte	-2147483648 bis 2147483647	ganzzahlig
Integer	**unsigned long**	4 Byte	0 bis 4294967295	ganzzahlig
Gleitkomma	**float**	4 Byte	$3{,}4\ 10^{-38}$ bis $3{,}4\ 10^{38}$	7 Stellen
Gleitkomma	**double**	8 Byte	$1{,}7\ 10^{-308}$ bis $1{,}7\ 10^{308}$	15 Stellen
Gleitkomma	**long double**	12 Byte	$3{,}4\ 10^{-4932}$ bis $1{,}1\ 10^{4932}$	23 Stellen

Bemerkung:

Der angegebene Speicherbedarf betrifft den Dev-Cpp Compiler. Vorallem der datentyp long double kann variieren zwischen 8, 10, 12 und 14 byte je nach Compiler.
Bei alten 16-bit Systemen (z.b Borland 3.1) ist beträgt für eine int-Variable der Speicherbedarf nur 2 byte !

3.1.1 Variablendeklaration

Variablen müssen durch Angabe des :

Variablennamen und
Variablentyps

deklariert werden.

Achtung: Jede Variable muss deklariert werden!

- Die Deklaration reserviert einen Speicherplatz für eine Variable.
- Die Größe des Speicherplatzes hängt von dem Typ der Variablen ab.

Beispiele:

int Zahl1, Zahl2; Deklarationen von Integervariablen
float Zahl3; Deklarationen von Variablen mit reellen Zahlen
int Zahl4 = 1; Deklaration und Definition (Wertzuweisung)

Wichtige Bemerkungen:

- Dezimalstellen werden mit einem Punkt (.) getrennt, nicht mit einem Beistrich (,)
- Float-Variable sollte man stets mit einem Dezimalpunkt zuweisen:
  Z1=23.0;
- Float-Variable niemals auf Gleichheit prüfen (vergleichen mit = =) -> Rundungsfehler !!!
- Zuweisung von ASCII-Zeichen:

```
char zeich;
zeich = 'D';          // ist gleich mit zeich1 = 68;  (da ASCII-Code von D ist 68)
zeich1='3';           // ist gleich mit zeich1=51;
```

- Die Darstellung einer Zahl als Hexadezimalzahl erfolgt mit dem Vorsatz **" 0x "**
  Z=0x2A3; ist gleich mit **Z=2*256+10*16+3;** ist gleich mit **Z=675; (-> Zahlensysteme)**
- **ACHTUNG !!** Eine führende 0 erzeugt eine Oktalzahl:
  nZ=023; ist gleich mit **nZ=2*8+3** ist gleich mit **nZ=19;**
- Binärzahlen sind in C und C++ nicht erlaubt, wohl aber in C für Mikrocontroller.

3.1.2 Typenumwandlung („casting“)

```
float Z1, Z2, ergebnis1;
int Zahl1, Zahl2, ergebnis2
...
ergebnis1 = (float)Zahl1 / Zahl2;          // oder : (float)Zahl1 / (float)Zahl2;
ergebnis2 = (int)Z1 % (int)Z2;
```

3.1.3 Konstantendeklaration

Für Zahlenwerte die im gesamten Programm nicht verändert werden dürfen, verwendet man zur Konstantendeklarationen (über eine im Speicher angelegte Variable) **„const“** oder **#define**.

```
const float KONSTANTE1 = 1234.567;
```

Alternativ kann auch #define verwendet werden (als textaler Ersatz):

```
#define KONSTANTE1   1234.567   // Präprozessor-Anweisung
```

ACHTUNG :

- Bei Zeilen mit #define kommt **kein Strichpunkt** am Ende der Anweisung.
- Konstanten schreibt man üblicherweise GROSS!

Beispiel:

```
#include ....
#define  PI  3.141529
int main( )
{       int x;
        x = PI; …..
}
```

4 Rechenoperatoren

=	Zuweisungsoperator	- -	Dekrement der Variablen (-1)
+	Addition	||	Bitweise ODER Verknüpfung
-	Subtraktion	|	Bitweise INKLUSIV ODER
*	Multiplikation	^	Bitweise EXKLUSIV ODER
/	Division	<<	Linksverschiebung
%	Modulo, ganzzahliger Divisionsrest	>>	Rechtsverschiebung
+ =	Additionszuweisung	&=	binär-UND-Zuweisung
- =	Subtraktionszuweisung	|=	binär-INKLUSIV-ODER-Zuweisung
* =	Multiplikationszuweisung	^ =	binär-EXKLUSIV-ODER-Zuweisung
/ =	Divisionszuweisung	<< =	Linksverschiebungszuweisung
% =	Modulo-Zuweisung	>> =	Rechtsverschiebungszuweisung
&	Bitweise UND	~	Einerkomplement. Invertiert die Bits eines Wertes
+ +	Inkrement der Variablen (+1)		

Beispiele zu den Rechenoperationen (Zuweisungen):

Zähler =Zähler +2 ;	Vor der Ausführung der Anweisung enthält der Speicherplatz **"Zähler"** einen bestimmten Wert. Nach der Abarbeitung des Befehls ist dieser Wert um 2 erhöht. Er steht aber noch immer am gleichen Speicherplatz.
Zähler += 2;	wie oben!
Nenner++;	Nenner wird um 1 erhöht
++Nenner;	Nenner wird um 1 erhöht
Erg = ErsteZahl + ZweiteZahl ;	Addiert die Erste und die Zweite Zahl.
Erg = ErsteZahl / Zweite Zahl ;	Dividiert die Erste durch die Zweite Zahl Das Ergebnis ist eine Zahl vom Typ float.
Erg = ErsteZahl % ZweiteZahl ;	Dividiert die Erste durch die Zweite Zahl Das Ergebnis ist der Rest der Division.

5 Steuerzeichen

Zeichen	Bedeutung
\a	Alarm (Bell) Zeichen
\b	Backspace (Funktion der Löschtaste mit dem Pfeil nach links)
\f	Formfeed (neues Blatt - neue Seite)
\n	Newline (neue Zeile)
\r	Carriage return (Cursor zurück zum Zeilenanfang - aber keine neue Zeile).
\t	Waagerechter Tabulatorschritt
\v	Senkrechter Tabulatorschritt
****	Backslash-Funktion
\?	Fragezeichen
\'	halbe Anführungszeichen
\"	Doppelte Anführungszeichen
\0	Null-Zeichen
\ooo	Oktal-Wert
\xhhhh	Hexadezimal-Wert

Innerhalb einer Zeichenkette (Strings) können die Steuer-Zeichen folgendermaßen eingefügt werden.

z.B: **cout << "Hello\n World";**

Außerhalb des Strings sind sie in halbe Anführungszeichen zu setzen: zB: **'\n'**

z.B: **cout << "Alarm\a\t Alarm\a\t Alarm\a" << '\n';**

6 Weitere Ausgabemöglichkeiten mit printf( )

dient der Bildschirm-Ausgabe (Benötigt **#include <iostream>**)

printf ("Die Summe der Zahlen %d und %d beträgt %d " , a , b , c) ;

Formatstring — Argumentliste

Im Formatstring stehen neben dem auszugebenden Text auch sogenannte Format-anweisungen. Diese beginnen mit einem %-Zeichen (abhängig vom Typ der Variablen !).

% d ... (signed) int (dezimal)	**% lf % le % lE % l g** ... double
% u ... unsigned int (dezimal)	**% Lf % Le % LE % Lg** ... long double
% ld ... signed long	**%c** ... char
% lu ... unsigned long	**%s** ... String (Zeichenkette)
%f %e %E %g ... float	**%%** ... Das Prozentzeichen wird ausgegeben

Beispiele:

printf ("Der erste Buchstabe im Alphabet ist %c." , 'A') ;
'A' - Charakterkonstante (steht zwischen einfachen Anführungszeichen) Ausgabe:
Der erste Buchstabe im Alphabet ist A.

printf ("Heute ist %s.", "Montag") ;
"Montag"-Stringkonstante (steht zwischen doppelten Anführungszeichen)
Ausgabe: Heute ist Montag.

printf ("Die Klasse hat %d Schüler." , 31) ;
Ausgabe: Die Klasse hat 31 Schüler.

printf ("Vierzigtausend unsigned %u", 40000);
Ausgabe: Vierzigtausend unsigned 40000

printf ("Vierzigtausend signed %d", 40000);
Ausgabe: Vierzigtausend signed –25536 (40000 liegt nicht im int-Bereich!)

printf ("Nun eine float-Konstante %f" , 3.14) ;
Ausgabe: Nun eine float-Konstante 3.14000

printf ("Andere Schreibweise %g", 3.14);
Ausgabe: Andere Schreibweise 3.14

(nachgestellte Nullen werden entfernt, Dezimalpunkt nur wenn nötig,...)

printf ("Wissenschaftliche Schreibweise %e", 3.14);
Ausgabe: Wissenschaftliche Schreibweise 3.14000e+00

Vollständige Syntax für die Formatanweisungen von printf( )

(Angaben in Eckigen Klammern sind optional und dürfen weggelassen werden, wenn sie nicht benötigt werden)

%[-][+][#][0][Breite][.Stellenzahl]Typ

-	Ausgabe erfolgt linksbündig
+	Zahlenwerte bekommen ein Pluszeichen vorangestellt, sofern sie positiv sind.
0	Die auszugebende Zahl bekommt linksbündig Nullen bis zur maximalen Feldbreite (siehe Breite) vorangestellt.
Breite	Minimale Gesamtlänge des Feldes (Es wird mit Leerzeichen aufgefüllt, steht vor Breite 0 wird mit Nullen aufgefüllt.
Stellenzahl	Anzahl der Nachkommastellen bei float und bei Strings die Anzahl der maximalen Breite.
Leerzeichen	positive Zahlen mit Leerzeichen (blank) beginnen.
#	Zahlen in Oktaldarstellung bekommen eine 0 und Zahlen in Hexadezimalform ein 0x (bzw. 0X) vorangestellt.

Bemerkung: In der Ausgabe entspricht jeder '_' einem Blank!

Beispiele:

int i = 1234;

Format	Ausgabe
%d	1234
%05d	01234
%5d	_1234
%x	4D2
%#x	0x4d2
%-5d	1234_
%2d	1234
%+08d	+0001234
%-8d	1234____

float f=12.34;

Format	Ausgabe
%f	12.340000
%+.1f	+12.3
%06.1f	0012.3
%-6.1f	12.3__

char a=65;

Format	Ausgabe
%c	A
&d	65
%o	101
%#10x	______0x41

float n=0.5;

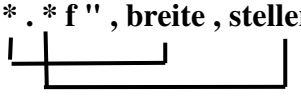
Format	Ausgabe
%e	5.000000e-001
%+f	+0.500000
%g	0.5

"St.Pölten"

Format	Ausgabe
%12s	___St.Pölten
%-12s	St.Pölten___
%4s	St.Pölten
%.4s	St.P

Es gibt auch die Möglichkeit, die Breite und Stellenzahl einer Zahl als Variablen (hier **breite** und **stellenzahl**) anzugeben:

printf (" % * . * f " , breite , stellenzahl , zahl) ;



7 Zufallszahlen

(Benötigt**: #include < iostream > bzw. <time.h>**)

rand( ) ; erzeugt Zufallszahlen und liefert eine Integerzahl zwischen 0 und 32767. Um einen Anfangswert setzen zu können und damit immer neue Kolonnen von Zufallszahlen zu erzeugen, muss zuerst die Funktion **srand(x)** aufgerufen werden, wobei x ein beliebiger Startwert für die Berechnung der Zufallszahl ist !

Beispiel: **srand(10);**

```
Zufallszahl = rand( );            // liefert genau eine Zufallszahl von 0 bis 32767
ZahlKleinerHundert = rand( ) % 101 ;  //Zufallszahl im Bereich von   0 bis 100
```

Achtung: Der Startwert x sollte auch "zufällig" gewählt sein !!!

Das realisiert man z.B. mit den Funktionen:

- **x=time(0);** // **x** gibt die Anzahl von Sekunden an, welche seit einem bestimmten Datum vergangen sind
- **x= clock( );** // **x** gibt die Anzahl von Millisekunden an, welche seit dem Start des Programms vergangen sind

z.B. **srand(time(0));** **oder**

srand(clock( ));

```
x= rand( )%6;              // Zufallszahl von 0 bis 5
x= rand( )%10 + 1;         // Zufallszahl von 1 bis 10
x= rand( )%11 - 5;         // Zufallszahl von -5 bis +5
```

8 Mathematische Funktionen

(Benötigen **#include <math.h>**)

sqrt(x)	Berechnet die Quadratwurzel
sin(x)	Berechnet den Sinus
exp(x)	Berechnet die Exponentialfunktion bzgl. e
log(x)	Berechnet den natürlichen Logarithmus
abs(x)	Berechnet den Betrag (nur für Integerzahlen)
fabs(x)	Berechnet den Betrag (als double)
pow (r, x)	Berechnet r^x
tan (x)	Berechnet den Tangens

Die Argumente der trigonometrischen Funktionen sin, cos, tan und das Ergebnis von acos, asin, atan werden im **Bogenmaß** gemessen (rad = grad * pi/180).

9 Verzweigungen

9.1 Vergleichsoperatoren

<	kleiner
< =	kleiner oder gleich
= =	gleich ACHTUNG: Zwei float-Zahlen können NICHT auf Gleichheit überprüft werden !!!
! =	ungleich
>=	größer oder gleich
>	größer

9.2 if-Anweisung

9.2.1 Einfache „if“ – Anweisung

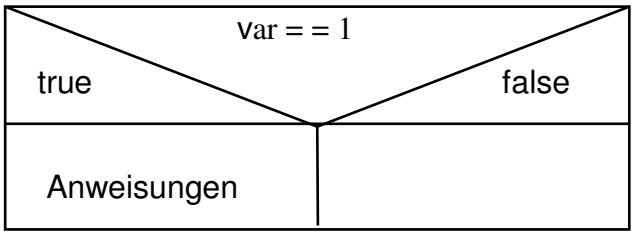
if (5.0 < var) Wenn **5 kleiner var** dann wird die nächste Anweisung oder die nächsten Anweisungen die sich in den { ..... } Klammern befinden, ausgeführt.

if (var = = 1) Wenn **var gleich 1** dann wird die nächste(n) Anweisungen ausgeführt.

if (var) Wie **if (var ==1)** !

if (!var) der **NOT Operator „!“** kehrt den Vergleich um (Wenn „var=0“)

Struktogramm:



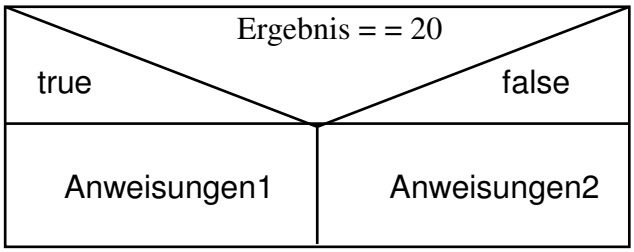
9.2.2 „if - else“ - Anweisung

```
if (Ergebnis = = 20)
{
      Anweisungen 1 ;
}
else
{
      Anweisungen 2 ;
}
```

Wenn der Inhalt der Variablen "**nErgebnis" gleich 20** ist werden die Anweisungen 1 ausgeführt.

Bei **allen anderen** Ergebnissen werden die Anweisungen 2 ausgeführt.

Struktogramm:



9.2.3 „else if“ - Anweisung

```
if (Ergebnis = = 20)
    {
      Anweisungen 1 ;
    }
else if  (Ergebnis > 20)
    {
      Anweisungen 2 ;
    }
else
    {
     Anweisungen 3 ;
    }
```

Wenn der Inhalt der Variablen "**Ergebnis" gleich 20** ist werden die Anweisungen 1 ausgeführt.

Bei "**Ergebnis" größer als 20** werden die Anweisungen 2 ausgeführt.

Bei "**Ergebnis" kleiner als 20** werden die Anweisungen 3 ausgeführt.

Achtung:
if (a = = 5) ist **nicht** gleich **if (a = 5)**

9.2.4 Logische Verknüpfungen

Wenn **mehrere Bedingungen** abzufragen sind, müssen die einzelnen Bedingungen mit logischen Operatoren verknüpft werden.
Beispiele:

if (2.0 <=zahl && zahl <= 3.0) **" zahl "** liegt zwischen 2 und 3

if (nVar == 3 | | nVar == 5) Wird nur dann ausgeführt wenn **"nVar"** den Wert 3 oder 5 besitzt. ("|" Zeichen mit AltGr + < eintippen)

9.2.5 Andere Schreibweise

Statt

```
if (a==2)
{
    b=3;
}
else
    b=4;
```

kann auch der **dreistellige Operator** **? :** *verwendet werden*

```
b = (a = = 2) ?  3 : 4 ;
```

9.3 Switch Anweisung (Mehrfachverzweigung)

```
switch  (var)
{
        case   1 :   Statement 1;
                       break;
        case   2 :   Statement 2;
                       break;
        case   5 :   Statement 3;
                       break;
        case   23 : Statement 4;
                       break;
        default:  Statement 5;
                       break;
}
```

Wenn das Inhalt der Variablen **var** 1 ist, dann wird das Statement 1 bis zu "**break**", abgearbeitet.
Die restlichen Zeilen werden bis zur " **}** " übersprungen.
Gleiches gilt für die Inhalte 2, 5 und 23.
(Ohne "**break**" werden die nächsten Abfragen ebenfalls abgefragt).

Gibt es einen anderen Inhalt als 1, 2, 5 oder 23 dann wird unter "**default**" das Statement 5 ausgeführt.

Beispiel:

```
...
unsigned int Zahl1;
cin >> Zahl1;

switch  (Zahl1)
{
        case   0 :   cout << " Zahl ist 0";
                     cout << " Das ist sehr wenig!!!";
                     break;

        case   1 :   cout << " Zahl ist 1";
                     break;

        case   2 :   cout << " Zahl ist 2";
                     break;

        case   3 :
        case   4 :    cout << " Zahl ist 3 oder 4";
                     break;

        default:
                     cout << " Zahl ist grösser 4";
                     break;
}
```

Ein Zweig der switch-Anweisung kann auch durch mehrere case-Werte ausgewählt werden (hier case 3 und 4)

Bemerkungen:

- Hinter case dürfen nur einfache Typen wie int, long oder char stehen (keine float, double,...)!!!
- Switch wird verwendet bei Mehrfachauswahl von konstanten Werten (z.B. Zahlen oder Zeichen).

10 Schleifen

10.1 Allgemein:

Für Programmschleifen gibt es in C++ 3 verschiedene Konstrukte:

for( ), while( ) und **do-while( ).**

Mit **break** können Schleifen an jeder **beliebigen Stelle verlassen** (= abgebrochen) werden.

Mit **continue** wird erzwungen, dass die Schleife wieder bei der **ersten Anweisung** beginnt.

10.2 Die while( ) Schleife

```
Zaehler = 1 ;
while ( Zaehler < 10 )
{
    Zaehler = Zaehler + 1 ;
    cout << " * " ;
}
```

Nach dem Schlüsselwort **while** folgt in der Klammer die „*Laufbedingung*".
Wenn die **Laufbedingung vor dem Aufruf** der Schleife nicht erfüllt ist, wird sie **kein einziges Mal** durchlaufen, sondern gleich übersprungen.

Deshalb heißt diese Schleife auch **"abweisende Schleife".**

10.3 Die do-while() Schleife

```
Zaehler = 55;
do
{
      Zaehler = Zaehler – 1 ;
      cout << " * " ;
} while (Zaehler > 10) ;
```

Die Abbruchbedingung wird am **Ende** des Schleifendurchlaufes geprüft.

Dadurch wird diese Schleife **mindestens 1 mal** durchlaufen, auch wenn die Laufbedingung von Anfang an nicht erfüllt ist.

10.4 Weitere Schreibweisen

Bei Verminderung um 1 ist folgendes gleichbedeutend:
Zaehler = Zaehler – 1 , Zaehler – = 1 oder Zaehler – – !!!

Bei Erhöhung um 1 ist folgendes zulässig:
Zaehler = Zaehler + 1 , Zaehler + = 1 oder Zaehler + + .

Jeder C++ Ausdruck und damit auch die Schleifenvariablen haben selbst einen Wert, mit dem auch **weitergerechnet** werden kann:

Beispiel:

```
{
    int Zaehler = 10 ;
    while (Zaehler – –)
    {     Ergebnis = Zaehler * 2 ;
          cout << "*" << Ergebnis ;
    }
}
```

Die While- Schleife wird solange durchlaufen, bis **Zaehler gleich 0** ist.
Der Wert 0 bedeutet falsch und damit Abbruch der Schleife.

Die Anweisung **while(1)** erzeugt eine Endlos-Schleife, die nur mit **break** verlassen werden kann!

10.5 for - Schleife

Sie wird dann eingesetzt, wenn die Anzahl der Durchläufe bekannt ist.
Dabei kann die Anzahl der Durchläufe entweder als eine Konstante oder eine Variable definiert sein.
Die Syntax der **for() Schleife** lautet:

for (Anfangsbedingung ; Laufbedingung ; Bearbeitung der Laufvariablen)

```
{       Anweisung1 ;
        Anweisung2 ; ....
}
```

- Beispiel:

```
for (Zaehler = 0 ; Zaehler < 15 ; Zaehler + + )
{
    gotoxy( 2, 2 + Zaehler);
    cout  <<  " * " ;
}
```

Diese Schleife wird 15 mal durchlaufen.

Beachte besonders die Anfangs- und die Laufbedingung:
- Soll mit 0 oder 1 begonnen werden?
- Steht in der Laufbedingung < oder <= ?

Wird die Schleife mit 0 begonnen und soll sie n-Mal durchlaufen werden, so muss die Laufbedingung <n lauten!

Die Anweisung **for(;;)** erzeugt eine Endlos-Schleife, die nur mit **break** verlassen werden kann!

- Beispiel: Es werden solange positive Zahlen addiert, bis die Zahl 0 eingegeben wird:

```
int main(void)
{
    int summe=0, Zahl;
    for(;;)                                  // Endlos-schleife
    {       cin >> Zahl;
            if (Zahl)
            {       if (Zahl <0) continue;   // Berechnung übergehen !
                    summe+=Zahl;
            }
            else
                    break;                   // Schleife verlassen
    }
printf(" Die Summe lautet %d",summe);
printf("Auf Wiedersehen!");
}
```

- Beispiel: Schleifenvariable wird um 3 erhöht nach jedem Schleifendurchlauf

```
for (i = 0 ; i < 15 ; i = i+3 )
{  printf(„Hallo“);
}
```

Frage: Wie oft wird Hallo ausgegeben ?

11 Beschreibung von Software (Ablauf von Programmen)

11.1 Struktogramme

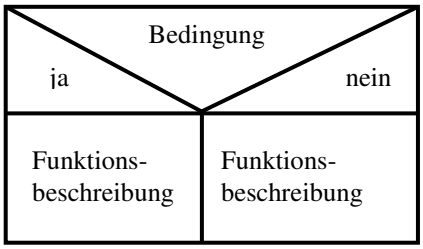
11.1.1 Elementarblock

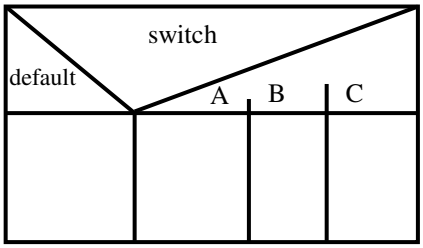
Einzelne Anweisung:

11.1.2 Modul- oder Funktionsblock

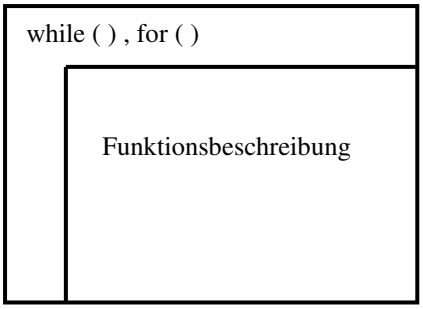
Ein Modulblock beschreibt einen öfter benötigten Algorithmus. Sie sind eine Zusammenfassung mehrerer einzelner Anweisungen.

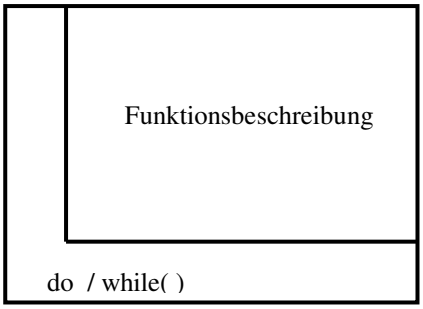
a) Verzweigungen:



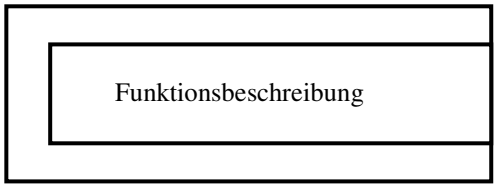


b) Schleifen:

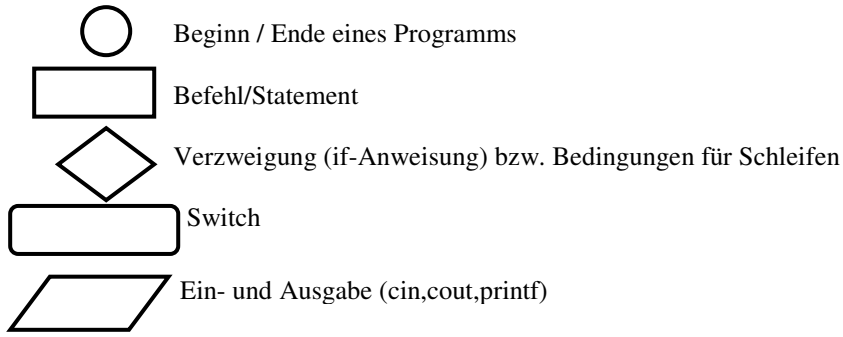




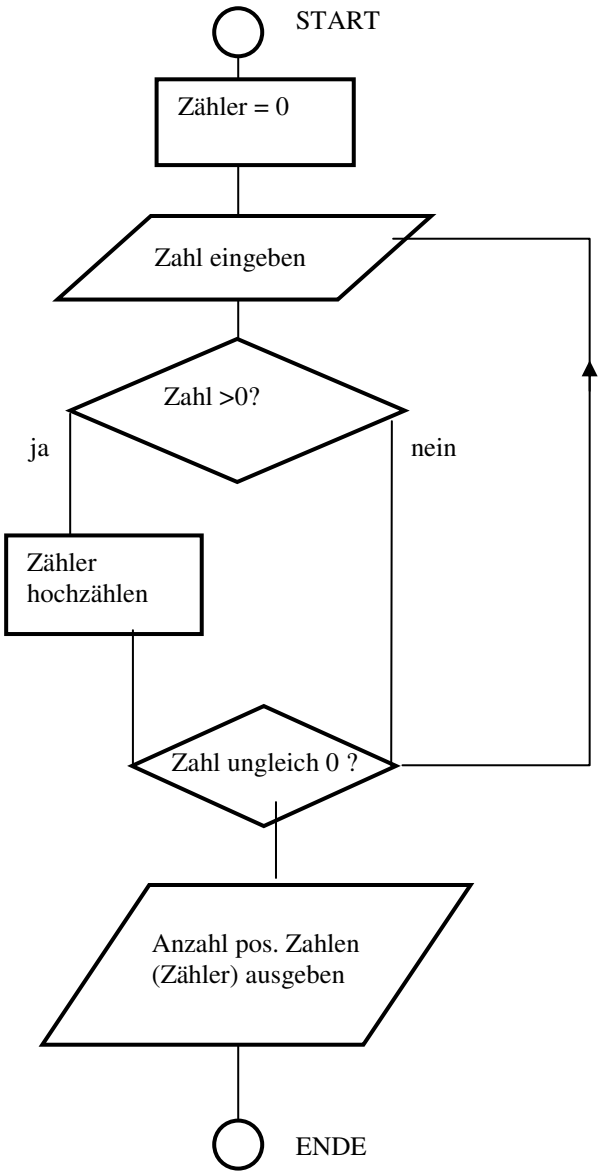
c) Unterprogramm-/Funktionsaufruf:



11.2 Flußdiagramme



Beispiel: Anzahl positiver Zahlen zählen und ausgeben



12 Felder (Array)

12.1 Eindimensionale Felder

Ein Feld (auch Array oder Vektor) ist eine Zusammenfassung von Variablen gleichen Typs, die im Speicher unmittelbar hintereinander stehen.

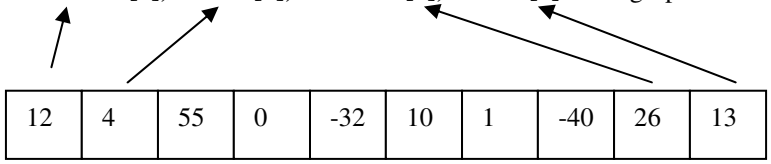
Bei der **Definition** eines Feldes sind folgende Informationen anzugeben:
- Feldtyp (d.h. der Typ der verwendeten Variablen)
- Feldname (Name, mit dem die gesamte Gruppe gekennzeichnet und angesprochen werden kann)
- Anzahl der Elemente

Beispiel:

```
int Zahlen[10];          /* deklariert ein Feld vom Datentyp int mit Namen Zahlen und
                            10 Elementen */
```

Die 10 Integervariablen können über die Namen

Zahlen[0], Zahlen[1], ... Zahlen[8], Zahlen[9] angesprochen werden.



Allgemein: **Zahlen [i]**

Feldname Index

```
Zahlen[0] = 12; Zahlen[1] = 4;          // Wertzuweisungen
Zahlen[4] = - 32; Zahlen[9] = 13;
```

Die Zählung der Feldelemente **beginnt immer** mit dem **Index 0** und **endet** bei **Anzahl –1 !!!**

Der Programmierer ist selbst verantwortlich, dass keine Feldelemente mit einem höheren als dem in der Deklaration angegebenen Index angesprochen werden. Eine solche Grenzüberschreitung wird vom Compiler nicht verhindert!!!

```
Zahlen [10] = 8;        // fehlerhaften Zuweisung: Speicherbereich oberhalb des letzten Feld-
                        // elementes Zahlen [9] wird überschrieben
```

Bei der Deklaration von Feldern sollen **definierte Grenzen** gewählt werden (mit #define)

```
z.B:  #define  MAX  5
      float Feld[MAX];      // Feld mit 5 Elementen
```

Initialisierung (Anfangswerte):

```
int Zahlen [10] = {0} ;                  // Das gesamte Feld Zahlen[ ] wird mit "0" aufgefüllt

for (i=0; i<10 ;i++) {Zahlen[i] = 0;}    // andere Variante

int Zahlen[10] = {0,1,2,3,4,5,6,7,8,9};  // Zuweisung der Werte "0“ bis "9“
```

Beispiel:

Ein Programm soll eine Folge von 10 ganzen Zahlen einlesen und jene Zahlen aus der Folge entfernen, welche grösser bzw. kleiner als die beiden eingegebenen oberen und unteren Grenzen.
Anschliessend wird die neue Zahlen-Folge wieder ausgegeben.

```
...

#define ANZAHL 10


void main(void)
{
    int i, j, o_grenze, u_grenze;
    int zahl[ANZAHL];

    clrscr();
    printf("\nGeben Sie die obere Grenze ein!");      // Eingabe der Grenzen
    cin >> o_grenze;
    printf("\nGeben Sie die untere Grenze ein!\n");
    cin >> u_grenze;

    for (i=0; i<ANZAHL; i++)                          // Eingabe der Zahlenfolge
    {
        cout << i+1 <<".Zahl:\n";
        cin >> zahl[i];
    }

    for (i=0, j=0; i<ANZAHL; i++)                     // Überprüfung, ob innerhalb der Grenzen
    {
        if (zahl[i] > u_grenze && zahl[i] < o_grenze)
        {
            zahl[j] = zahl[i];                        // auszugebende Zahlen ins Feld
    schreiben
            j++;
        }
    }
    printf(" \nAlle Zahlen, welche innerhalb der Grenzen %d und %d liegen,
    lauten:\n", u_grenze, o_grenze);
    for (i=0; i<j; i++)                               // Ausgabe der Zahlenfolge
    {
        cout << zahl[i] << "\n";
    }

    printf("Auf Wiedersehen!");
}
```

12.2 Mehrdimensionale Felder

Um ein mehrdimensionales Feld zu deklarieren, sind mehrere Größenangaben nötig.

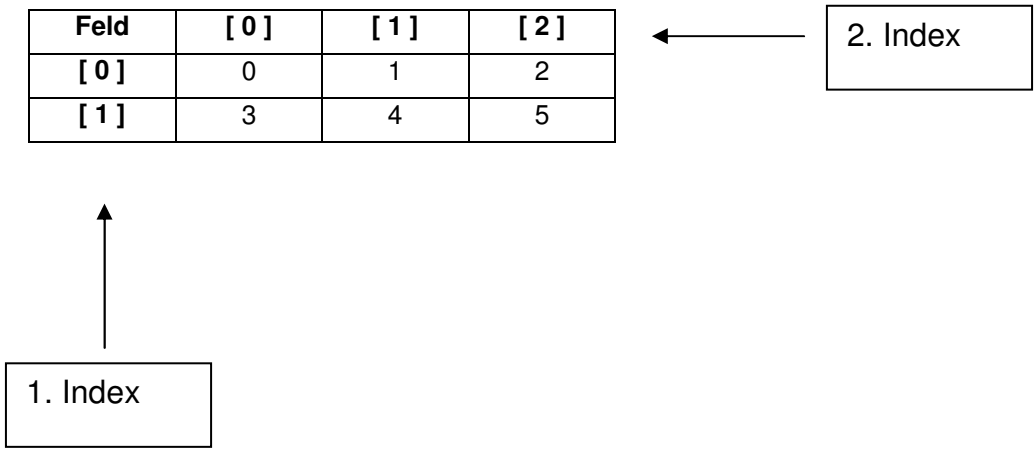
```
int Feld [2] [3] ;
```

Hier wird ein zweidimensionales lnteger-Array deklariert.
Folgende lnitialisierung macht deutlich, dass es sich dabei um ein zweielementiges Feld, dessen Elemente selbst Integerarrays mit jeweils 3 Elementen sind, handelt:

```
int Feld [2] [3] = {{0,1,2} , {3,4,5}};
```

Jedes Element muss ünber **zwei** Indizes angesprochen werden.
Man kann sich ein solches zweidimensionales Feld auch als Tabelle (Matrix) vorstellen:

Feld	[0]	[1]	[2]
[0]	0	1	2
[1]	3	4	5



Beispiel:

Es soll für 4 Wochen jeden Tag die Temperatur in eine Tabelle eingelesen werden.

```
...
#define WOCHE 4

void main(void)
{
    int i,j;
    int temperatur[WOCHE] [7];

    for (i=0; i < WOCHE; i++)                    // je Woche
    {
            cout << i+1 <<".Woche:\n";
            for (j=0; j < 7; j++)                // je Tag
            {
                    cout << j+1 <<".Tag:\n";
                    cin >> temperatur[i][j];     // Eingabe der Temperatur
            }
    }

}
```

12.3 Weitere Übungsbeispiele zu Feldern

Beispiel A:

Schreibe die Zahlen von 1 bis 10 in ein Feld und gib dieses dann aus.

```
void main(void)
{
    int feld[10], i;

    for (i=0; i < 10; i++)                    // Erzeugen der Zahlenfolge
    {
            feld[i] = i + 1;                  // Zuweisung (1. Feld beginnt mit Index 0 !!!)
            printf("\n Element %d = %d", i+1, feld[i]);  // Ausgabe: Element 1 = 1 ....
    }
}
```

Beispiel B:

Lese solange ganze Zahlen von der Tatstatur ein, bis 0 eingegeben wurde (Maximal 40 Zahlen). Die grösste positive Zahl soll ausgegeben werden.

```
....
#define MAX_ANZAHL 40

void main(void)
{
    int feld[MAX_ANZAHL], i =0, max = 0;

    do                                        // Schleife für Eingabe
    {       cout << i+1 << ".Zahl:\n";
            cin  >> feld[i];
            if (feld[i] > max)
            {       max = feld[i];            // momentane grösste Zahl abspeichern
            }
            i++;
    } while ((feld[i] != 0) && (i < MAX_ANZAHL));

    if (max)
    {       printf(" \n Die grösste Zahl = %d", max);
    }
    else
    {       printf(" \n Es wurde keine positiven Zahlen eingeben !!!");
    }
}
```

13 Unterprogramme (Funktionen)

Jedes C++-Programm besteht aus Funktionen. Die wichtigste Funktion ist das Hauptprogramm main( ).
Diese Funktion wird innerhalb eines C-Programms als erste abgearbeitet und ruft dann weitere Funktionen auf.
Wenn gleiche bzw. sehr ähnliche Programmstücke öfter gebracht werden, sollte man für diesen Code nur einmal eine **Funktion** schreiben.

Vorteil: - immer wieder verwendbar
- übersichtliches Programm
- Aufteilung der Aufgabe zwischen mehreren Personen
- leichteres Auffinden von Fehlern

Vordefinierte Funktionen sind z.B. printf ( ), gotoxy ( ), getche ( ), ...

Jede Funktion hat eine nachfolgende Klammer, in der eventuell *Übergabeparameter* stehen:

z.B. gotoxy (10,2);

Wenn keine Parameter zu übergeben sind, so bleibt die Klammer leer !

z.B. c = getch ( );

Bei diesem Beispiel gibt die Funktion getch ( ) einen Wert zurück und speichert ihn in der Variable *c* (Typ ist char) !

Jede Funktion kann, aber muss keinen *Rückgabewert* (Return-wert) haben.

Funktionsnamen werden üblicherweise klein geschrieben bzw. fangen nur mit einem Grossbuchstaben an !

13.1 Deklaration und Definition einer Funktion:

Jede Funktion besteht aus 3 Teilen:

1) Funktionsprototyp: Deklaration der Funktion mit der Parameterliste
   Er steht immer vor main ( ) !
   Er kann auch entfallen, wenn die Funktion vor dem Hauptprogramm definiert wird.
   -> Praxis: immer Prototypen verwendet, da die Funktionen entweder nach dem main ( ) definiert werden oder überhaupt in einer eigenen Datei stehen !!!

2) Funktions-Code: das eigentliche Programm (Definition der Funktion)

3) Funktionsaufruf: Anwendung der Funktion (kann im main ( ) oder von einer anderen Funktion kommen)

13.2 Unterprogramm ohne Übergabeparameter

Beispiel 1:

```
#include <stdio.h>

void function1 ( );                    // Funktions-Prototyp

int main ( )
{  int zaehler;
   for (zaehler = 0; zaehler < 5; zaehler++)
   {    function1 ( );                 // Aufruf der Funktion
   }
  return 0;
}

void function1 ( )                     // Definition der Funktion
{
        printf (" \nIch bin eine Funktion !");
}
```

=> Ergebnis: 5 mal wird *„Ich bin eine Funktion!"* ausgegeben !"

Beispiel 2: Berechnung der Quadratzahl

```
void quad ( );
float zahl, ergebnis;                  // globale Variable

int main ( )
{       cout << "Gib eine Zahl ein !\n";
        cin >> zahl;
        quad ( );
        cout << ,, Die Quadratzahl lautet: ,, << ergebnis ;
        return 0;
}
void quad ( )
{       ergebnis = zahl * zahl;        // Berechnung
}
```

Globale Variablen: Diese Variablen gelten ab dem Zeitpunkt der Deklaration für alle Funktionen (Hauptprogramm und Unterprogramme) und werden immer **vor** main ( ) deklariert !!!

Sie kann jederzeit von jeder Funktion geändert werden und sollte möglichst immer vermieden werden !!!!!

13.3 Unterprogramm mit Übergabeparameter (Ein- und Ausgabeparameter)

Beim Unterprogrammaufruf können auch Werte von Variablen (Parameter) übergeben werden.

Man unterscheidet 2 Möglichkeiten der Parameterübergabe:

a) „**call by value**“:
Parameter werden auf einen speziellen Speicherplatz kopiert (*stack*) und für das Unterprogramm verwendet. Nach dem Verlassen des Unterprogramms wird dieser wieder freigegeben (= gelöscht).
Die ursprüngliche Variable wurde NICHT verändert, d.h. es wurden nur Eingabeparameter verwendet !!!

b) „**call by reference**“:
Will man Variablen durch ein Unterprogramm verändern, so verwendet man Ausgabeparameter für die Übergabe.
Dies wird durch das **&** - Zeichen (den ***Adressoperator***) erreicht, welcher der Variablen vorangestellt wird. Dadurch wird der Funktion nicht der Wert der Variablen, sondern nur ihre (Anfangs-) Adresse (oder „**Zeiger**“) übergeben.
Die ursprüngliche Variable wird durch den Unterprogrammaufruf auch im aufrufenden Programm (z.B. main()) verändert !!!!!

Beispiel 1:

Es wird die Folge von n Zahlen erzeugt, in der jede Zahl den doppelten Wert der vorhergehenden hat (beginnend mit 1). Ausserdem soll die Summe berechnet werden.

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

void berechnung ( int, int &);                    // Funktions-Prototyp

int main ( )
{  int anzahl, summe;
   cout << "\nGeben Sie die Anzahl ein:";
   cin >> anzahl;
   berechnung (anzahl, summe);                    // Aufruf der Funktion
   cout << "\n Die Summe lautet:" << summe;
   return 0;
}

void berechnung ( int anz, int &sum)              // Definition der Funktion
{    int i, zahl;
     zahl = 1; sum = 1;
     cout << zahl1;
     for (i = 0; i < anz; i++)
     {    zahl = zahl*2;
          printf ("  %d",zahl);
          sum += zahl;
      }
}
```

Beachte: kein Adressoperator bei Unterprogrammaufruf !!!

Beispiel 2a: Übergabe als Wert (*call by value*)

Die Funktion *triple( )* multipliziert den übergebenen Wert mit dem Faktor 3 ohne die Variable im Hauptprogramm zu ändern.

```
#include <iostream>

void triple ( int);                    // Funktions-Prototyp

int main ( )
{   int zahl = 2;

    triple (zahl);
    cout << "\n Die neue Zahl lautet:" << zahl;    // zahl = 2   d.h. die Variable wurde im
    return 0;                                     // Hauptprogramm nicht verändert !
}

void triple (int z)
{   z *= 3;
    cout << "\n Die neue Zahl lautet:" << z;       // zahl = 6
}
```

Beispiel 2b: Übergabe als Verweis (*call by reference*)

Die Funktion *triple( )* multipliziert den übergebenen Wert mit dem Faktor 3 und verändert damit auch die Variable im Hauptprogramm.

```
#include <iostream>

void triple ( int &);                  // Funktions-Prototyp inkl. Adress-Operator

int main ( )
{   int zahl = 2;

    triple (zahl);
    cout << "\n Die neue Zahl lautet:" << zahl;    // zahl = 6   d.h. die Variable wurde im
    return 0;                                     // Hauptprogramm verändert !!!
}

void triple (int &z)
{   z *= 3;
    cout << "\n Die neue Zahl lautet:" << z;       // zahl = 6
}
```

Beispiel 3:

Die Funktion *tausch( )* vertauscht die beiden übergebenen Werte.

```
#include <iostream>

void tausch ( float &, float &);                    // Funktions-Prototyp

int main ( )
{   float zahl1 = 2.22, zahl2 = 5.55;

    tausch (zahl1,zahl2);
    cout << "\n Zahl1 =:" << zahl1 << endl;        // zahl1 = 5.55
    cout << "Zahl2 =:" << zahl2 << endl;          // zahl2 = 2.22
    return 0;
}

void tausch (float &a, float &b)
{   float temp;
    temp = a;
    a = b;
    b = temp;
}
```

Zusammenfassung:

Übergabe als Wert (call by value)	**Übergabe als Verweis (call by reference)**
float z; Der Parameter z ist eine lokale Variable. z ist ein Duplikat der Eingabeparameters (Argumentes). Der Eingabeparameter kann nicht verändert werden. Der als Wert übergebene Parameter kann eine Konstante oder eine Variable sein.	**float &z;** Der Parameter z ist ein lokaler Verweis. z ist ein nur ein Synonym (anderer Name) des Eingabeparameters. Der Eingabeparameter kann verändert werden. Der als Wert übergebene Parameter muss eine Variable sein.

Die Ein- bzw. Ausgabeparameter eines Unterprogramms werden auch **Argumente** genannt.

13.4 Funktionen (Rückgabewert)

Falls ein Unterprogramm selbst ein Resultat zurück liefert (*Rückgabewert*), so spricht man von einer Funktion im engeren Sinn.

Das Resultat der Funktion wird durch die Anweisung

return *Wert;*

an das aufrufende Programm zurückgegeben und kann dort einer Variablen vom selben Typ zugewiesen werden.

Die return-Anweisung erfüllt 2 Aufgaben:

- Sie beendet die Ausführung der Funktion.
- Sie liefert einen Wert an das aufrufende Programm zurück.

Natürlich kann die Funktion auch neben einem Rückgabewert Ein- bzw. Ausgabeparameter haben.

Der Syntax für eine Funktion *function( )* lautet:

Rückgabetyp **function** (*Übergabeparameter*) { ... }

Beispiel:

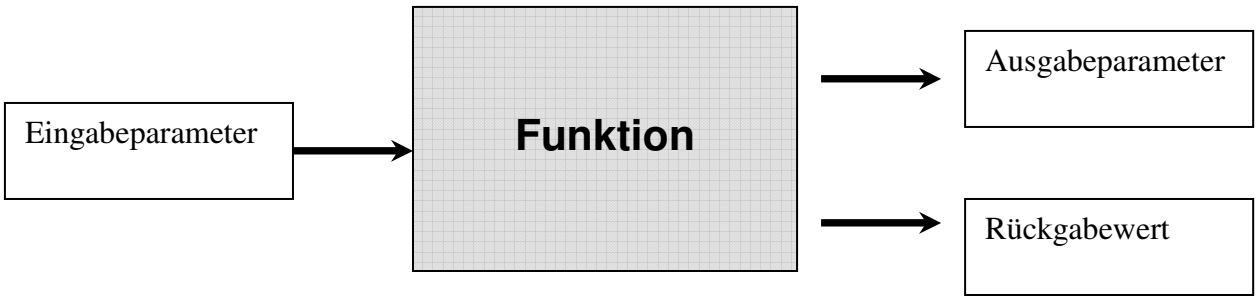
```
float cube( float);              // Funktions-Prototyp

main( )
{ float n,result;
.....
cin >> n;
result = cube(n);                // Funktionsaufruf
cout >> result;
....
}

float cube( float x)
{
return x*x*x;                    // liefert den kubische Wert zurück
}
```

Bemerkungen:

- Es können nur einfache Datentypen bzw. Zeiger als Rückgabewert (z.B. int, float,char, double, ...)verwendet werden, niemals jedoch Felder !!!
- Der Datentyp des Rückgabewertes muss immer mit dem Typ der zugewiesenen Variablen im aufrufenden Programms übereinstimmen !

13.5 Zusammenfassung:

Schema einer Funktion: (Schnittstellen zum aufrufenden Programm)



Die Wertübergabe in bzw. aus der Funktion lässt sich mit den Parametern bewerkstelligen.

Eingabeparameter lassen Werte nur in die Funktion hinein übergeben, jedoch keine Werte, die in der Funktion entstanden sind, über diese Schnittstelle heraus übergeben (call by value).

Ausgabeparameter übergeben hingegen Werte, die in der Funktion verändert bzw. entstanden sind, über diese Schnittstelle an das aufrufende Programm (call by reference).

Zusätzlich kann die Ausgabe eines Funktionsergebniswertes über den Rückgabewert erfolgen.

Lokale - globale Variablen: Gültigkeit

Lokalen Variablen werden innerhalb eines Blockes (z.B. innerhalb von main( ) bzw. einer Funktion) deklariert und gelten nur innerhalb dieses Blocks.
Daher sind auch die Übergabeparameter lokale Variablen ! Sie können daher auch den selben Namen haben, wie im aufrufenden Programm. Dies sollte jedoch aus Übersichtsgründen unterlassen werden.

Globale Variablen werden außerhalb eines Blocks deklariert, wie z.B. vor main( ). Sie gelten ab dann überall im Programm, d.h. sowohl im Hauptprogramm als auch in allen nachfolgenden Unterprogrammen. Durch die Verwendung von globale Variablen kann theoretisch die Verwendung von Übergabeparametern entfallen.
Grundätzlich sollten jedoch NIE globale Variablen verwendet werden (unübersichtlich, schwer nachvollziehbar wie die Variable verändert wird, ...) !!!

Hauptprogramm int main( ):

Bei Verwendung eines Compilers entspechend der neuen ANSI/ISO Standards (z.B. DevCpp, ...) ist das Hauptprogramm selbst eine Funktion, welche einen Returnwert an das Betriebssystem zurückliefert.

```
int main( )
{       int i;
        .....   //Befehle

        return 0;
}
```

d.h. **jedes** Hauptprogramm endet mit einem return-Befehl !

Die return-Anweisung:

Die **return**-Anweisung **beendet an dieser Stelle die Funktion** und kehrt zum nächsten Befehl des aufrufenden Programms zurück.

Sie sollte daher immer am Ende einer Funktion stehen bzw. unter einer Bedingung ausgeführt werden!

Beispiel:

```
float max (float x, float y)          // berechnet das Maximum von x und y
{       if (x > y)      return x;     // return erzeugt das Ergebnis und bricht ab
        else            return y;
        clrscr( );                   // dieser Befehl wird NIE ausgeführt !
}
```

Eine void-Funktion (d.h. eine Funktion mit dem return-Typ void) kann auch mit einer return-Anweisung abgebrochen werden!

Beispiel:

```
void tausch (int &x, int &y)
{       int hilf;
        if (x == y)             // falls x  ist gleich gross wie y dann Funktion
abrechen
                return;
        else
        {       hilf = x;
                x = y;
                y = hilf;
        }
}
```

13.6 Inline - Funktionen (ab C++)

Inline Funktionen sind vorteilhaft bei kurzen, rasch auszuführenden Funktionen.
Inline Funktionen werden durch den Compiler nicht in Funktionsaufrufe übersetzt, sondern werden **an allen Stellen der Aufrufe eingefügt**. (Die Funktion wird als Makro umgesetzt.)

```
float ausgabe(float);

int main ( )
{       float x,zahl; ....
        x = ausgabe(zahl);              // -> x = zahl*zahl*zahl;
...}

inline float ausgabe(float z)           // Inline Definition
        {
            return z*z*z;
        }
```

Vorteil: schnellere Ausführzeit
Nachteil: längeres Programm

13.7 Überladene Funktionen (ab C++)

Überladene Funktionen sind Funktionen mit **gleichen Funktionsnamen**, jedoch unterschiedlicher Signatur (unterschiedliche Anzahl oder Typen der Parameter).

Der Compiler kann an der Aufrufstelle die richtige Funktion aus dem Vergleich der Typen der aktuellen mit den formalen Parameter herausfinden und aufrufen.

Überladene Funktionen können z.B. für gleichartige Operationen verwendet werden, die mit unterschiedlichen Datentypen erledigt werden sollen.

```
float quadrat(float);
long quadrat(long);

int main( )
{ ....
     return 0;
}

float quadrat( float x)                 // Funktion quadrat für float-Werte
{ return x*x; }

long quadrat( long x)                   // 2.Funktion quadrat für long-Werte
{ return x*x; }
```

13.8 Bool'sche Funktionen

Funktionen, welche nur die Werte 0 und 1 an das aufrufende Programm zurückgeben, nennt man auch "*Bool'sche Funktionen*".
Das können z.B. Funktionen sein, welche einen Sachverhalt überprüfen (z.B. ist das eingegebene Zeichen eine Ziffer ?).

Je nach Ergebnis wird von der Funktion als Wert

1 (TRUE) oder
0 (FALSE) zurückgegeben.

Der Variablen-Typ des Rückgabewertes kann ***int*** bzw. ***bool*** sein.

Beispiel: Funktion untersucht, ob eingegebene Zahl positiv ist

```
bool IsPositive(float);

int main( )
{     float zahl;
      bool ret;

      cin >> zahl;
      ret = IsPositive(zahl);
      if (ret)
        cout << "\nZahl ist positiv!";
      else
        cout << "\nZahl ist negativ!";

      return 0;
}

bool IsPositive ( float x)
{ if (x >=0)
      return TRUE;
  else
      return FALSE;
}
```

oder noch einfacher ohne Hilfsvariable ***ret*** :

```
int main( )
{     float zahl;

      cin >> zahl;
      if (IsPositive(zahl))          // Es kann der return-Wert sofort für die
        cout << "\nZahl ist positiv!"; // Bedingung verwendet werden!!
else   .....
}
```

13.9 Rekursive Unterprogramme und Funktionen

Rekursive Funktionen rufen sich selbst auf .

Beispiel: Berechnung der Fakultät durch Rekursion

Fakultät: n! = 1*2*3*...*n = n*(n-1)! = n*(n-1)(n-2)!

```
int main( )
{       ...
        cout <<  fakult(3);
}

long fakult(long n)
{       if (n > 1)                              // Ausstiegsbedingung
                return n*fakult(n-1);
        else
                return 1;
}
```

bei n=3:

fakult(3)	3*fakult(3-1)=3*fakult(2)
fakult(2)	2*fakult(2-1)=2*fakult(1)
fakult(1)	1

d.h. Ergebnis berechnet sich von unten nach oben :

fakult(n) = 1*2*3 = 6

ACHTUNG:

Man kommt sehr schnell an die Kapazitätsgrenzen oder sehr lange Berechnungszeiten ! Man muss sich immer sehr genau die Ausstiegsbedingung (d.h. die Bedingung, welche zum Beenden des Funktionsaufrufes führt) überlegen.

13.10 Felder als Übergabeparameter

Felder können auch als Parameter einer Funktion bzw. Unterprogramms übergeben werden.
Dabei ist zu beachten, dass Felder intern immer als Referenz , d.h. über die Anfangsadresse des Feldes übergeben werden.

Es gibt 3 Ausführungsmöglichkeiten:

a) **Feld-Parameter über Feldklammern Angabe**:

```
int sum(int [ ], int);                  // Prototyp
int main( )
{       int feld[ ] = { 5,10,15,20}
        int size;
        size = sizeof(feld)/sizeof(int);   // Bestimmung der Anzahl der Feldelemente
        cout << sum(feld,size);
}

int sum(int a[ ], int s)
{       int sum = 0, i;
        for (i=0;i<s;i++)
           sum += a[i];
        return sum;
}
```

Wenn ein Feld an eine Funktion übergeben wird, so entspricht der Wert des Feldnamens **feld** in Wirklichkeit der Speicheradresse des ersten Feldes (**&feld[0]**) !
Die Übergabe eines Feldes gleicht daher der Übergabe einer Variablen als Verweis, obwohl die Feld-Variable als Wert übergeben wird.
Die Feldelemente können daher durch die Funktion verändert werden!!!
Beachte, dass die Grösse eines übergebenen Feldes der Funktion bekanntgegeben werden muss, denn in C++ kann die Funktion selbst die Grösse des übergebenen Feldes nicht erkennen !

sizeof(x) bestimmt die Anzahl der Bytes von x .
d.h. **sizeof(feld)** gibt die Anzahl der Elemente mal bytes(Typ) an !
hier: 4*4 = 16 Bytes bzw. 4*2 = 8 Bytes (Rechnerabhängig)

b) **Feld-Parameter über *Feldtyp* Angabe:**

```
#define MAX 5
typedef int TypFeld[MAX];                    // Typdefinition von TypFeld

void init(TypFeld, int);                     // Prototyp
int main( )
{       TypFeld feld1;
        init(feld1,5);
}

void init(TypFeld a, int s)
{       int i;
        for (i=0;i<MAX;i++)
                a[i] = s;
}
```

Mit der **typedef**-Anweisung wird für eine Variable ein eigener Typ definiert.
Es wird dabei ein neuer Name für einen bestehenden Typ (hier int Feld[ ]) vereinbart.
Beachte, dass die Anzahl der Elemente eines Feldes nicht Bestandteil seines Typs sind.

c) **Feld-Parameter über *Zeigerparameter*:**

```
void init(float *, float, int);              // Prototyp
int main( )
{       float feld2[10], value;
        cin >>value;
        init(feld2,value,10);
}

void init(float *a, float s, int n)
{       int i;
        for (i=0;i<n;i++)
                a[i] = s;
}
```

Der *** - Operator** bezeichnet einen sogenannten ***"Zeiger"*** auf eine Variable. Ein Zeiger ist eine **Adresse** der Variablen, d.h es wird nicht der Inhalt, sondern der Speicherplatz der Variablen bestimmt (**float *a** ist ein Zeiger der float-Variablen **a**).

Merke: Der Funktionsaufruf ist in allen 3 Fällen gleich !!!

14 Zeichenketten (Strings)

Ein String ist ein *Feld vom Typ char* und damit eine Zeichenkette. Das Ende eines Strings wird jedoch nicht durch die Längenangabe ermittelt, sondern besteht aus dem *Endezeichen* '\0' (binäre Null 0x00).
ACHTUNG: nicht verwechseln mit der Ziffer '0' (0x40) !!!

14.1 Deklaration:

```
char text[10];
```

vereinbart einen String mit dem Namen text bestehend aus 10 Zeichen. Es können daher nur 9 Zeichen genutzt werden, da das letzte Zeichen für das String-Ende reserviert wird.

Wie bei Feldern muss jedoch immer die Feldgrösse angegeben werden !

14.2 Initialisierung:

```
text[0] = 'G';
text[1] = 'a';
text[2] = 'n';
text[3] = 'd';
text[4] = 'a';
text[5] = 'l';
text[6] = 'f';
text[7] = '\0';                    // auch möglich: text[7] = 0;
```

d.h. es wurde der String text mit "Gandalf" angelegt.

oder einfacher:

```
char text[ ] = {'G','a','n','d','a','l','f',0};
```

bzw.:

```
char text[ ] = "Gandalf";
```

d.h. hier wird automatisch als letztes Zeichen '\0' angefügt und die Grösse des strings muss nicht explizit festgelegt werden.

14.3 Ein- und Ausgabe:

Stream-Funktionen:

```
char name[5];
cin >> name;          // auch Eingabe von mehr als 4 Zeichen möglich !
                      // z.B. "Legolas"
cout << name;         //Ausgabe des strings bis zum nächsten Ende-Zeichen
                      // hier: Legolas
```

z.B: **cin >> name;** //Eingabe: *"Hallo Legolas!"*
cout<< name; //Ausgabe: "Hallo"

d.h.: Der string endet mit <RETURN> oder Leerzeichen !!!

formatierte Ein- bzw. Ausgabe:

```
scanf("%s",name);     // formatierte Eingabe

printf("%s",name);    // formatierte Ausgabe
```

Auch hier endet der string mit <RETURN> oder Leerzeichen !

Bibliotheksfunktionen: (benötigt <stdio.h>)

```
gets(name);           // Eingabe, wird mit <RETURN> abgeschlossen

puts(name);           // Ausgabe
```

Die Funktion **gets( )** ermöglicht das Einlesen von Strings auch mit Leerzeichen .

Beispiel:

```
char text[1];
gets(text);           // Eingabe: "Hallo Freund!"

puts(text);           // Ausgabe: "Hallo Freund!"
```

!!! ACHTUNG:

```
text[13] = '!';       // Ende-Zeichen wird überschrieben !!!
puts(text);           // Ausgabe erfolgt solange, bis ein '\0' - Zeichen
                      // gefunden wurde !!!
```

14.4 Stringverarbeitung:

```
char text1[10], text2[10];

text1 = "Gandalf";      // Fehlermeldung  !!!
text2 = text1;          // Fehlermeldung  !!!
```

Eine direkte Zuweisung ist bei Strings nicht möglich !

Dazu verwendet man Funktion ***strcpy( )*** Bibliothekt <string.h> :

```
strcpy( text2,text1);          // kopiert text1 -> text2
strcpy( text2,"Hallo");        // beachte: text2 ist 6 byte lang (inkl. '\0')
```

Weitere Funktionen von <string.h> :

sprintf( ) : kopiert bzw. konvertiert eine Variable in einen String

```
float x=14.12;
sprintf(text1,"%6.2f",x);          // wandelt float-Zahl x in einen String um
sprintf(text1,"Ciao");             // kopiert "Ciao" -> text1
```

strlen( ) : Ermittelt die Länge eines Strings (ohne Endezeichen bzw. Nullterminator)

```
int x;
char text1[ ] = "Guten Morgen";
x = strlen(text1);                 // x = 12
```

strcmp( ) : Vergleicht einen String mit einem anderen String.

Bei Gleichheit wird 0 zurückgeliefert.

```
x = strcmp(text1,text2);
if(!x)                             // falls beide Strings gleich sind
{  .....  }
```

strcat( ) : Hängt einen String an einen anderen String an.

```
char text1[ ] = "Frodo", text2[ ] = " Beutlin";
strcat(text1,text2);       // ergibt für text1 = "Frodo Beutlin"
```

15 Strukturen (structs)

Eine Struktur ist ein Datentyp, der mehrere Variablen unterschiedlichen Typs aufnehmen kann.
In der Praxis fasst eine Struktur alle Variablen, welche sinngemäss zusammengehören, in einen Verbund zusammen (z.B. Daten einer Person mit Geburtsdatum, Alter, Adresse, Telefon, ...).
Somit kann eine vernünftige Organisation von Daten erreicht werden.
Jede Variable ist eine Komponente der Struktur, und sie kann eine einfache Variable (von Typ int, float, char, ...), Felder, Strings, Zeiger und auch eine andere Struktur sein.

15.1 Deklaration:

a) mit Typdefinition (typedef) :

```
typedef struct Schueler        // Struktur-Typ
{ char name[20];               // Komponente1 ist ein string
  int alter;                   // Komponente2 ist eine ganze Zahl
 int noten[10];                // Komponente3 ist ein Zahlen-Feld
  . . .
};

Schueler meier,mueller;        // Variablendeklaration
```

➔ d.h. ***meier*** und ***mueller*** sind Variablen vom Typ ***Schueler*** mit den Eigenschaften ***name***,***alter***, ***noten***, ...

b) gemeinsame Typdefinition und Variablendeklaration:

```
struct Schueler                // Struktur-Typ
{ char name[20];               // Komponente1
  int alter;                   // Komponente2
 int noten[10];                // Komponente3
  . . .
} meier,mueller;               // Variablendeklaration
```

➔ d.h. auch hier sind *meier* und ***mueller*** Variablen vom Typ ***Schueler***.

c) Variablendeklaration ohne Typdefinition:

```
struct
{ char name[20];          // Komponente1
  int alter;              // Komponente2
 int noten[10];           // Komponente3
  . . .
} meier,mueller;          // Variablendeklaration
```

➔ d.h. auch hier sind ***meier*** und ***mueller*** Variablen mit derselben Struktur, jedoch gibt es keinen eigenen Variablentyp, welcher diese Struktur beschreibt.

Hier beschreibt die Struktur ***Schueler*** die Eigenschaften eines Schülers. Damit ergibt sich eine sinnvolle Datenorganisation, die eine Vielzahl von Einzeldaten logisch zusammenfasst.

+ Vorteil: übersichtliches Programmieren

15.2 Zugriff auf die Komponenten:

Auf die einzelnen Komponenten einer Struktur-Variable wird über den Auswahloperator " . " (Punkt) zugegriffen.

z.B.
```
x = mueller.alter;
printf("\nName des Schülers ist %s", meier.name);
cout << mueller.note[3];

char wort[20];
strcpy(wort,mueller.name);        // mueller.name ist ein string !
```

Mit ***mueller.alter*** wird auf die Komponente ***alter*** der Strukturvariablen ***mueller*** zugegriffen. Der Typ von ***mueller.alter*** entspricht einem *int* .

Beispiel: Feld von Strukturen:
```
Schueler klasse1[25];          // Klasse von 25 Schülern
klasse1[3].alter = 20;         // Alter des 4.Schülers
```

klasse1 ist ein Feld von 25 Strukturen vom Typ ***Schueler***!
klasse1[3] entspricht dem 4.Schüler !

15.3 Zuweisung von Strukturvariablen:

Die Zuweisung von Strukturvariablen gleichen Typs ist möglich, d.h. die Werte aller Komponenten werden direkt auf einmal zugewiesen!

```
Schueler huber;          // Deklaration
huber = meier;           // Zuweisung
```

d.h alle Eigenschaften von meier besitzt nun auch huber!

```
klasse1[10] = mueller;
```

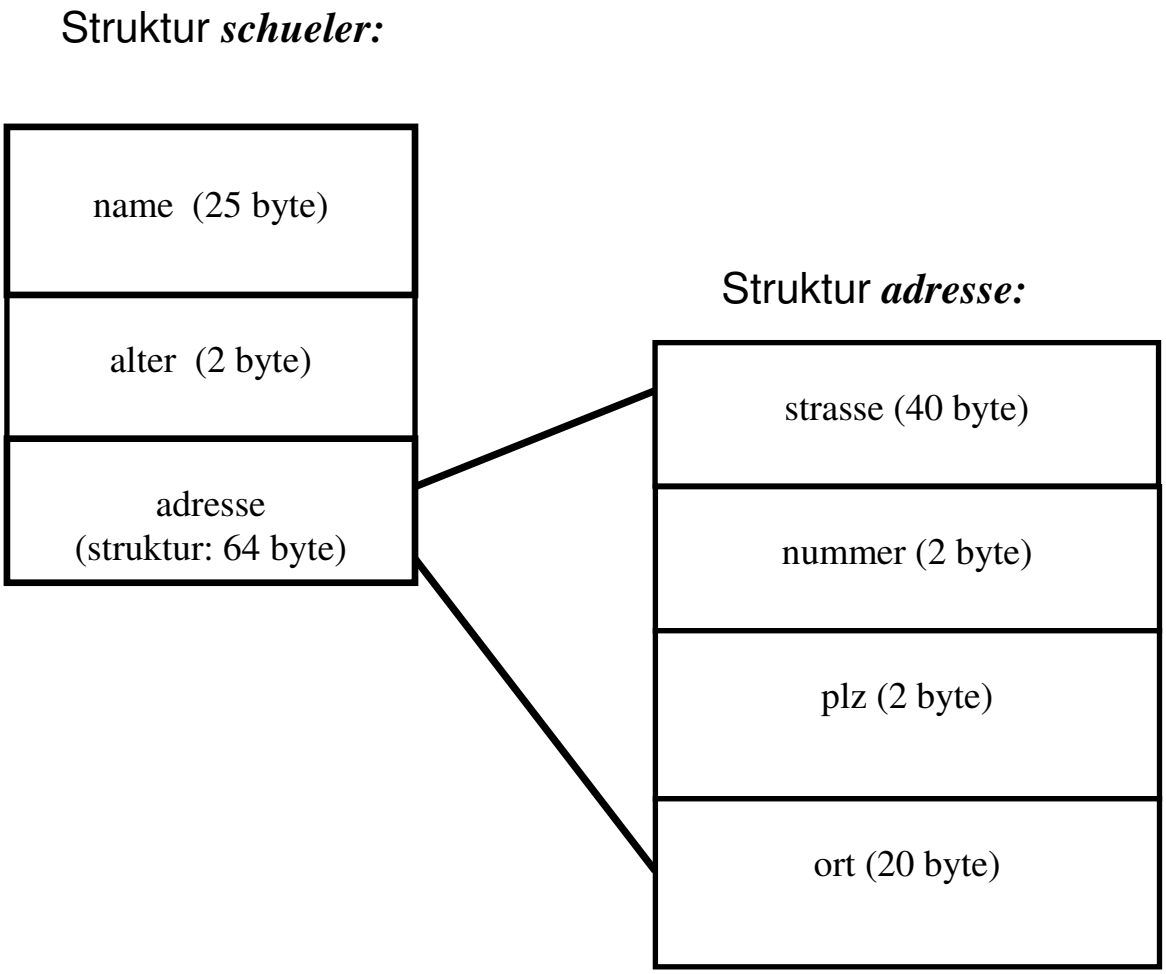
Natürlich können auch die Komponenten einzeln zugewiesen werden:

```
strcpy(huber.name,meier.name);
huber.alter = meier.alter;
. . .
```

15.4 Verschachtelte Strukturen:

Als Komponenten einer Struktur können auch wieder Strukturen verwendet werden (verschachtelte Strukturen).

```
typedef struct adresse
{       char strasse[40];
        int nummer;
        int plz;
        char ort[20];
};
struct schueler
{       char name[25];
        int alter;
        adresse adr;          // Struktur Adresse als Komponente
} meier;

adresse a;                    // Strukturvariable a
```



Der Zugriff auf die Komponenten einer verschachtelten Struktur kann über eine Mehrfachauswahl erfolgen:

```
meier.adr.plz =3100;
strcpy(meier.adr.stasse,"Waldstrasse");
a = meier.adr;
```

Mit ***meier.adr.plz*** wird auf die Komponente ***plz*** der Komponente ***adr*** von ***meier*** zugegriffen. Der Typ von ***meier.adr*** ist ***adresse*** !

15.5 Strukturen als Parameter:

Strukturen können als Ein- bzw. Ausgabegabeparameter mittels **Adress- oder Zeigerparameter** übergeben werden.

Beispiel eingabe(...) liest die Daten eines Schülers ein:

- **Variante 1:** call by reference mit Adressparameter &

```
void eingabe_sch(Schueler &s)
{       cout<<"\nName = ";
        gets(s.name);                   // Name des Schülers eingeben
        cout<< "\nAlter = ";
        cin>> s.alter;                  // Alter des Schülers eingeben
}

int main( )
{ ...
Schueler mueller;
eingabe_sch(mueller);                   // Funktionsaufruf
}
```

Die Aufruf und Übergabe einer Struktur erfolgt hier wie bei einfachen Variablen.

- **Variante 2:** call by reference mit Zeigerparameter *

```
void eingabe_sch(Schueler *s)
{       cout<<"\nName = ";
        gets(s->name);                  // Name des Schülers eingeben
        cout<< "\nAlter = ";
        cin>> s->alter;                 // Alter des Schülers eingeben
}

int main( )
{...
Schueler mueller;
eingabe_sch(&mueller);                  // Aufruf mit &-Operator !!!
```

ACHTUNG: Wird ein Zeiger auf eine Struktur (Zeigerparameter) einer Funktion übergeben, so erfolgt der Zugriff auf die einzelnen Komponenten mittels **Pfeiloperator (->) .**
Beim Zugriff auf die einzelnen Komponenten (z.B. ***mueller->name***) kann auch statt des Pfeiloperators ***(*mueller).name*** verwendet werden.

15.6 Unions:

Bei Strukturen wird für jede Komponente ein eigener Speicherplatz verwendet.
Unions sind ähnlich wie Strukturen, jedoch werden alle Komponenten an derselben Speicheradresse angelegt (element overlaying).
Daher bestimmt die typmässig grösste Komponente die gesamte Speichergrösse der Union.

Beispiel: Zugriff auf den gleichen Speicherplatz mit verschiedenen Datentypen

```
typedef union Zahl          // Definition der Union Zahl ( int oder byte)
{     int i;
      char b[2];            // 2 bytes bzw. char
};

      union Zahl x;         // Deklaration von x als eine union-Variable
      ...
      x.i =33;
      cout<< x.b[0];        // lower byte von x ausgeben
      cout<< x.b[1];        // higher byte von x ausgeben
```

d.h. Variable x kann entweder vom Typ int oder vom Typ char[2] sein !!!

Verwendung: - wahlweise Verwendung von Komponenten
- Zugriff auf den gleichen Speicherplatz mit verschiedenen Datentypen (Type-Overlaying)

16 Dateiverarbeitung

Dateien (Files) dienen zur permanenten Speicherung von Daten. Dateien können Daten entweder im Text- oder im Binärformat speichern.
Textdateien speichern die Daten aller Variablen (auch int, float,...) im ASCII Code und können damit auch mit jedem Texteditor bearbeitet werden (benötigen allerdings mehr Speicher: jede Ziffer = 1byte!)..
Binärdateien hingegen speichern die Daten im jeweiligen Binärcode des Variablentyps. Binärdateien sind damit kompakter und bei großen Datenmengen (Grafikdateien, ...) effizienter.

Der **Dateizugriff** kann entweder sequentiell oder wahlfrei erfolgen.
Beim sequentiellen Zugriff werden die Daten beginnend mit dem ersten Element hintereinander bis zum Dateiende (EOF = End Of File) abgearbeitet.
Beim wahlfreien Zugriff (random access) wird auf einzelne Datensätze an bestimmten Positionen der Datei zugegriffen.
Die Übertragung der Daten vom Programm zur Datei wird über einen Datenpuffer abgewickelt, der als Zwischenspeicher dient und eine blockweise Übertragung zum Laufwerk ermöglicht.

Im Programm erfolgt der Dateizugriff in 3 Schritten :

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

16.1 Datei öffnen

die Bibliotheksfunktion **fopen** (in <stdio.h>) öffnet eine Datei und gibt einen Zeiger auf den Typ FILE zurück:

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

FILE* Datei Descriptor (File Pointer) :
FILE ist eine Struktur in <stdio.h> mit den Informationen für die I/0-Routinen, wie Pufferadresse, Puffergröße, Position des Schreib-/Lesezeigers, etc.
Im Fehlerfall wird die Zeigerkonstante "NULL" zurückgeliefert.
path Pfadname der Datei :
zu beachten ist, dass bei DOS-Pfadangaben der „\“ im C-String durch „\\“ anzugeben ist.
mode Zugriffsmodus (Modus String) :

„r“	read (lesen)	Datei muß existieren
„w“	write (schreiben)	Dateien werden überschrieben
„a“	append(anhängen)	schreiben an das Dateiende
„t“	Textformat	
„b"	Binärformat	

Beispiel zu Datei öffnen **:**

```
FILE *datei;                                    //File-Pointer definieren
datei = fopen("C:\\daten.dat", "w");            //Datei öffnen für Schreiben
if      (datei==NULL)
{       fprintf(stderr,"Datei kann nicht geöffnet werden");
        exit(1);                                //bei Fehler -> Abbruch
}                                               // stderr: Fehlerausgabe Bildschirm
```

16.2 Datei schliessen:

die Bibliotheksfunktion ***fclose( )*** schließt eine Datei

```
fclose( FILE *stream);
```

Beispiel**:** **fclose (datei);**

16.3 Textdatei Zugriff:

Textdateien werden üblicherweise seqentiell abgearbeitet. Die C-Bibliotheksfunktionen (in <stdio.h>) erlauben zeichenweises, zeilenweises, formatiertes und auch blockweises Schreiben und Lesen.

16.3.1 Zeichenweises Lesen u. Schreiben :

Lesen: **char fgetc(FILE *stream);**

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

fgetc liest ein Zeichen vom Datei-Stream und gibt das Zeichen als Rückgabewert aus, bei Dateiende bzw. Fehler den Wert EOF.
fputc schreibt ein Zeichen auf den Datei-Stream und gibt im Fehlerfall als Rückgabewert EOF aus.
Danach wird jeweils der Filepointer auf die nächste Stelle verschoben!

Beispiel: **Zeichen einer Datei wird gelesen**

```
char z;
FILE *datei;
datei = fopen("test.txt","r");
z = fgetc(datei);
cout << z;                      // gelesenes Zeichen wird ausgegeben
fclose(datei);
```

Beispiel: **Zeichen wird in eine Datei geschrieben**

```
char z, dname[20];
FILE *datei;

gets(dname);                    // einlesen eines Dateinamens
datei = fopen(dname,"w");       // öffnen der Datei
z=getche( );                    // Zeichen einlesen
fputc(z,datei);                 // Zeichen in Datei schreiben
cout << z;
fclose(datei);
```

Zeilenende von Textdateien:

Für das Zeilenende sind folgende zwei Steuerzeichen wichtig:

```
#define CR 13      // Carriage Return (entspricht '\r' : zurück an Zeilenanfang)
#define LF 10      // Linefeed (entspricht '\n' : nächste Zeile)
```

Bei **<ENTER>** wird mit der Funktion ***getch( )*** bzw. ***getche( )*** nur das CR abgespeichert. In (MSDOS-) Textdateien muss daher das LF geschrieben werden (statt des CR), damit in die nächste Zeile gegangen wird.

```
z=getche( );

if (z == CR)
{  z = LF;
   cout << z;          // Ausgabe nächste Zeile am Bildschirm
}
fputc(z,datei);        // Zeichen in Datei schreiben
```

16.3.2 Zeilenweises Lesen u. Schreiben :

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

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

fgets( ) liest n-1 Zeichen oder bis Zeilenende von Datei in den String s, wobei das Endezeichen '\0' automatisch an den String s angehängt wird.

Falls schon das Dateiende erreicht wurde bzw. im Fehlerfall wird als Rückgabewert **0** ausgegeben.

fputs( ) schreibt den String s auf die Datei und gibt im Fehlerfall den Rückgabewert **EOF** aus.

Beispiel: **Textdatei zeilenweise lesen**

```
char s[81];                          //Hilfsvariable s
int z;
for( z=1;z<=20;z++)                  // von Zeile 1 bis 20
{     if (fgets(s, 80, datei)==0)    // zeilenweise von Datei lesen
      { break;                       // Dateiende -> Ausstieg
      }
      printf("%s",s);                // String s auf Bildschirm ausgeben
}
```

Beispiel: **Textdatei zeilenweise schreiben**

```
....
gets(s);                             // Zeile bzw. string einlesen
fputs(s, datei);                     // string in Datei schreiben
....
```

16.3.3 Formatiertes Lesen u. Schreiben

Lesen: **int fscanf(FILE *stream, char *format, ...);**
Schreiben: **int fprintf(FILE *stream, char *format, ...);**

Mit ***fscanf( )*** und ***fprintf( )*** kann durch Angabe des Formatstrings - wie bei ***printf( )*** - (z.B. **%d, %f, %s**, ...) **formatiert** von einer Datei gelesen und auf eine Datei geschrieben werden.
Der Rückgabewert liefert die Anzahl der fehlerfrei gelesenen / geschriebenen Elemente oder **EOF** bei Dateiende / im Fehlerfall.

Beispiele:

```
char s[20];
int zahl;
fscanf(datei, "%s", s);              // String bzw. Wort wird aus Datei gelesen
fscanf(datei, "%d", &zahl);          // int-Zahl wird aus Datei gelesen
fprintf(datei,"%d", zahl);           // int-Zahl wird in Datei geschrieben
```

Beachte: **&-Operator** bei ***fscanf( )***, da Strings mit Adresse angesprochen werden !!!

***Beispiel*: Nach einem bestimmten Wort suchen**

```
int wort_search(char *filename, char *wort)
{       FILE *datei;
        int anz=0,x;
        char string[81];

        gets(filename);
        datei = fopen(filename,"r");          // öffnen der Datei
        if (datei ==NULL) return 0;           // Fehler beim Lesen
        while(1)
        {       x=fscanf(datei,"%s",string);
                if(x ==EOF)                   // Dateiende
                        break;
                if(strstr(string,wort))       // Suche nach Wort
                        anz++;
        }
        fclose(filename);
        return anz;
}
```

Die Funktion **strstr(char *string1,char *string2)** aus der Bibliothek <string.h> sucht im String **string1** nach dem Teilstring **string2** und gibt bei Erfolg 1 zurück **!**
(Man kann hier natürlich auch die Funktion **strcmp( )** verwenden!)

***Beispiel*: Zahlen zählen**

```
{  ...   int zahl;
         datei = fopen(dname,"r");
         while(1)
         {  x=fscanf(datei,"%d",&zahl);
          if(x == EOF)
                break;
          if(x == 0)          // wenn keine Zahl !!!
                break;
          anz++;
  ....   }
```

Datei mit Zahlen:

4711 22 23 24 \n
1 2 3 4 5 \n
222 xyz 30 31 EOF

fscanf(..) gibt 0 zurück wenn bei "xyz" angelangt !

➔ Immer auf Format der Variablen achten !!!

16.4 Wahlfreier Zugriff (random access)

Die Funktion **fseek( )** setzt den ***Dateizeiger*** an eine neue Position.

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

Parameter: **stream** : File Pointer
offset : Position in Bytes als Offset (Differenz) zu der Position origin
origin : Anfangs-Position:

0 = SEEK_SET	Dateianfang
1 = SEEK_ CUR	aktuelle Position
2 = SEEK_ END	Dateiende

Rückgabewert : 0 bei fehlerfreier Ausführung, im Fehlerfall ungleich 0

Es kann damit an bestimmten Positionen zugegriffen werden auf:
- **binären Dateien** : auf einzelne Datensätze (z.B. Strukturen)
- **Textdateien** : auf einzelne Bytes

***Beispiel Textdatei*:**

```
FILE *ptr;
long pos=0;                          // erstes Byte
if (fseek(ptr, pos, SEEK_SET))       // Positionieren auf pos-tes Element
    return 0;                        // Abbruch bei Fehler
```

***Beispiel Binärdatei*:**

z.B. in einer Datei steht ein ganzer Datensatz von Autos:

```
. . .
FILE *ptr;
int pos=3;                           // 3. Element
car Auto;                            // Struktur car
int len=sizeof(car);

if (fseek(ptr,pos*len,SEEK_SET))     // Positionieren auf 3. Element Auto
    return 0;                        // Abbruch bei Fehler
```

16.5 Binäres Lesen u. Schreiben:

Um an einer bestimmten Stelle in eine Datei zu schreiben bzw. davon zu lesen benutzt man folgende Funktionen:

Lesen : **int fread (void *zeig, int size, int n, FILE *stream)**
Schreiben: **int fwrite (void *zeig, int size, int n, FILE *stream)**

Parameter: **zeig :** Zeiger auf den Speicherbereich der Daten (Variable)
size : Größe der Datenelemente in Bytes
(kann mit sizeof (typ/variable) bestimmt werden)
n : Anzahl der Datenelemente
stream : File Pointer

Daher werden insgesamt **n * size** Bytes gelesen / geschrieben.
Bei Textdateien (jedes Zeichen besteht ja nur aus einzelnen Bytes), ist **size** immer 1 !!!

Rückgabewert: liefert die Anzahl der gelesenen geschriebenen Datenelemente.

Beispiel : Schreiben eines Wortes in Textdatei:

```
FILE *ptr;                          // File Pointer
int len;
char wort[20];
ptr=fopen("test.txt","r+w");        // Datei öffnen für Schreiben
if (ptr ==NULL)
        return 0;                   // Abbruch bei Fehler
gets(wort);
fseek(ptr, 10, SEEK_CUR);           // Positionieren des File-Pointers

n=fwrite(wort, 1, 5, ptr);          // von Wort werden 5 Zeichen geschrieben

if (n==0)
{       fclose(ptr);
        return 0;                   // Abbruch bei Schreibfehler
}
 fclose(ptr);
```

Binär-Dateien:

Binär-Dateien speichern die Daten in ihrer internen Form. Datensätze können als Elemente einer Struktur angelegt werden.

Beispiel : Lesen eines Datenelementes in Binärdatei:

```
FILE *ptr;                          // File Pointer
car Auto;                           // Struktur car
int len=sizeof(car);

ptr=fopen("test.txt","w+b");        // Datei öffnen für Schreiben
if (ptr ==NULL)
      return 0;                     // Abbruch bei Fehler

fseek(ptr, 0, SEEK_CUR);            // Positionieren des File-Pointers

n=fread(&Auto, len, 1, ptr);        // die Daten eines Autos werden gelesen

if (n==0)
      return 0;                     // Abbruch bei Schreibfehler

fclose(ptr);
```