TU Clausthal

Institut für Prozess- und
Produktionsleittechnik

Skript

Einführung in das Programmieren (für Ingenieure)

Christian Vetter

Arnold-Sommerfeld-Straße 1 • D-38678 Clausthal-Zellerfeld • Tel: 05323/72-7500 • Fax: 05323/72-7599

Arnold-Sommerfeld-Straße 1 • D-38678 Clausthal-Zellerfeld • Tel: 05323/72-7500 • Fax: 05323/72-7599

Inhalt

- Planung und Entwurf mit Struktogrammen
- Grundlagen
- Bedingte Programmierung
- Schleifen
- Felder
- Funktionen
- Dateien
- Datenstrukturen
- Zeigertechnik
- Blick in Fensterprogrammierung (event-orientiert)

Arnold-Sommerfeld-Straße 1 • D-38678 Clausthal-Zellerfeld • Tel: 05323/72-7500 • Fax: 05323/72-7599

Struktogramme

Einführung in das Programmieren (für Ingenieure)

Programmieren in C

Übersicht

- Planung und Entwurf mit Struktogrammen
- Grundlagen
- Bedingte Programmierung
- Schleifen
- Felder
- Funktionen
- Dateien
- Datenstrukturen
- Zeigertechnik
- Blick in Fensterprogrammierung (event-orientiert)

Darstellung von Algorithmen mit Struktogrammen (I)

Algorithmus: allgemeine Berechnungsvorschrift; unabhängig von der Realisierung in einer Programmiersprache

Kontrollstrukturen:

- Sequenz
- Auswahl oder Verzweigung
- Schleife oder Wiederholung
- Aufruf anderer Algorithmen

dienen zur Darstellung von Algorithmen.

Struktogramm: Darstellung eines Algorithmus mit Hilfe von Kontrollstrukturen.

Darstellung von Algorithmen mit Struktogrammen (II)

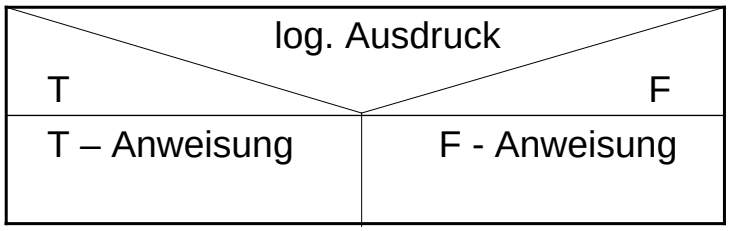
Sequenz: Aneinanderreihung von Anweisungen.

Anweisung 1
Anweisung 2
Anweisung 3

Darstellung von Algorithmen mit Struktogrammen (III)

Auswahl: Anweisungen in Abhängigkeit von Bedingungen ausführen lassen (**Selektion**).

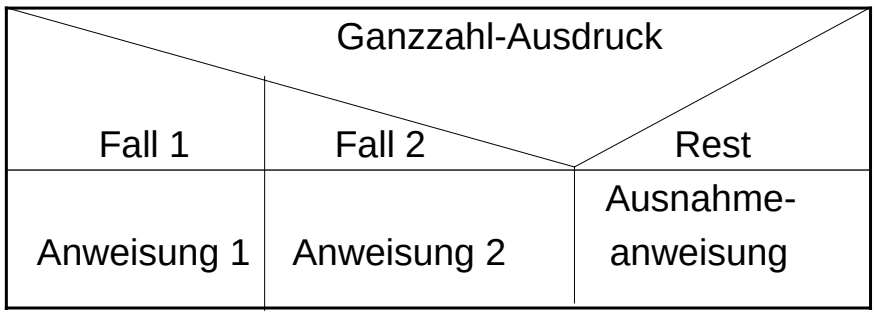
- ein- und zweiseitige Auswahl



Darstellung von Algorithmen mit Struktogrammen (IV)

Auswahl: Anweisungen in Abhängigkeit von Bedingungen ausführen lassen (**Selektion**).

- Mehrfachauswahl, Fallunterscheidung



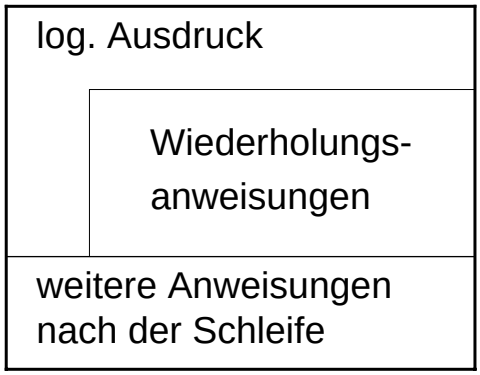
Darstellung von Algorithmen mit Struktogrammen (V)

Schleife: eine oder mehrere Anweisungen werden in Abhängigkeit von einer Bedingung wiederholt durchlaufen (**Iteration**)

- Schleife mit Abfrage **vor** jedem Schleifendurchlauf (**abweisend**)
- Schleife mit Abfrage **nach** jedem Schleifendurchlauf (**nicht abweisend**)
- Schleife mit vorgegebener Durchlaufzahl (Zählschleife).

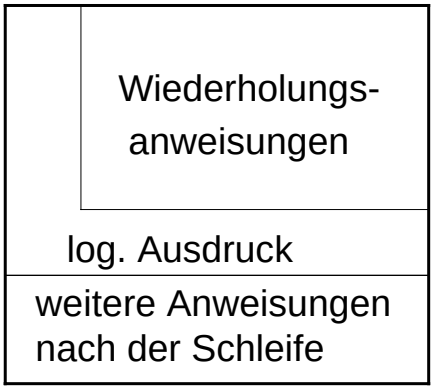
Darstellung von Algorithmen mit Struktogrammen (VI)

Schleife mit Abfrage **vor** jedem Schleifendurchlauf



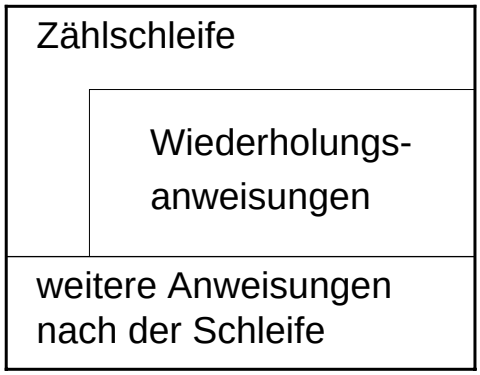
Darstellung von Algorithmen mit Struktogrammen (VII)

Schleife mit Abfrage **nach** jedem Schleifendurchlauf



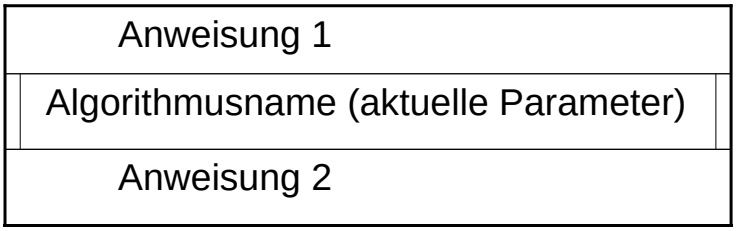
Darstellung von Algorithmen mit Struktogrammen (VIII)

Schleife mit vorgegebener Durchlaufzahl (Zählschleife)



Darstellung von Algorithmen mit Struktogrammen (IX)

Aufruf anderer Algorithmen: erfolgt durch Angabe des Algorithmusnamens gefolgt von der Liste aktueller Parameter. Nach Ausführung des aufgerufenen Algorithmus folgt die Fortsetzung des rufenden Algorithmus hinter der Aufrufstelle.



Darstellung von Algorithmen mit Struktogrammen (X)

Schachtelung von Kontrollstrukturen: Innerhalb von Schleifen können wieder Schleifen oder Auswahlanweisungen stehen. Kontrollstrukturen können in beliebiger Kombination und beliebiger Tiefe geschachtelt werden.
Programme sind Realisierungen von Algorithmen in einer Programmiersprache.

Struktogramm-Editor-Programm

- Zu finden unter
  http://www.strukted.de/

Grundlagen

Einführung in das Programmieren (für Ingenieure)

Programmieren in C

Der C Zeichensatz:

1. Alphanumerische Zeichen:
   26 Großbuchstaben, 26 Kleinbuchstaben
   (lateinisches Alphabet)
   --- Groß- und Kleinschreibung werden unterschieden! ---
   10 Ziffern 0 1 2 3 4 5 6 7 8 9
2. Sonderzeichen: ()[]{}<>+-*/%^~&|_=!?#\,.;:“‘
3. Steuerzeichen:
   Leerzeichen, Zeilenende-Zeichen, horizontaler Tabulator, vertikaler Tabulator, Seitenvorschub

4. Zeichen, die leicht verwechselt werden:
   - Ziffer 0 und Buchstabe O
   - Ziffer 1 und Buchstaben l, I,J
5. Format von C Programmen: formatfrei

Elementare Datentypen

Datentyp	stellt dar	Zusätze
char	Zeichen, ganze Zahl	signed
int	ganze Zahl	short, long, unsigned,
float	Fließkommazahl in einfacher Genauigkeit	
double	Fließkommazahl in doppelter Genauigkeit	long

Zahlbereiche für Datentypen in Systemdateien limits.h, float.h

```
#define SHRT_MIN    (-32768)  /* minimum (signed) short value */
#define SHRT_MAX      32767  /* maximum (signed) short value */
#define INT_MIN  (-2147483648) /* minimum (signed) int value */
#define INT_MAX   2147483647   /* maximum (signed) int value */


#define FLT_MIN      1.175494351e-38F      /* min positive value */
#define FLT_MAX     3.402823466e+38F      /* max value */
#define DBL_MIN      2.2250738585072014e-308
                                          /* min positive value */
#define DBL_MAX    1.7976931348623158e+308 /* max value */
```

Bezeichner und Namen (I)

Bezeichner(identifier): dienen zur Identifizierung von Objekten innerhalb eines C-Programms.
Datenobjekt: Speicherbereich, der aus einem einzelnen Byte (8 Bit) oder einer zusammenhängenden Folge von Bytes besteht.

sizeof-Operator: dient zur Ermittlung der Größe des Speicherbereiches eines Typs.

z.B.: sizeof(float) ergibt bei heutigen Rechnern 4

Speicherbereich der Datentypen in Byte

Datentyp	Speicherbereich	Sammelbegriff
char	1	
short int	2	Ganzzahlen-
int	4	Datentyp
unsigned int	4	-“-
long int	8	-“-
float	4	Fließkomma-
double	8	Datentyp

Bezeichner und Namen (II)

Bezeichner: Bestehen aus einer Folge alphanumerischer Zeichen, wobei das *erste Zeichen ein Buchstabe* ist. Dabei zählt der Unterstrich (_) als Buchstabe.

Spezielle Bezeichner:

- Reservierte Worte: Schlüsselworte (siehe Tabelle), nicht als Bezeichner von Objekten verwendbar.
- Namen: Bezeichner von Variablen, Funktionen und Marken. Signifikant sind 31 Zeichen bei internen Namen.

Groß- und Kleinschreibung werden unterschieden!

Beispiele: gültige Namen: A, _a, a2, vektor, kraft, bereit
ungültige Namen: do, 1a, a-b

Reservierte Schlüsselwörter

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

Konstanten (I)

Jede Konstante hat einen Datentyp.

Konstanten vom Ganzzahlen-Datentyp: Ziffernfolge **ohne** Dezimalpunkt. Zahlsystem als Lese- und Schreibschnittstelle: *dezimal*, *oktal* (0) und *hexadezimal* (0x)

Beispiel: 25(dezimal) = 031(oktal) = 0x19(hexadezimal)

Konstanten vom Fließkomma-Datentyp: Ziffernfolge **mit** Dezimalpunkt; Darstellung *mit* oder *ohne* Exponent (Zehnerpotenz).

Beispiel: 127.4 = 0.1274e+3
0.00432 = 0.432e-2

Zeichenkonstanten (char-Konstanten): Darstellung als ganze Zahl gemäß **ASCII-Zeichensatz**.

Beispielprogramm zur Ausgabe der ASCII-Tabelle.

Konstanten (II)

Zeichenkettenkonstante (string-Konstante): eine Zeichenkette mit n Zeichen benötigt Speicherbereich der Länge n+1 wegen des *abschließenden Nullzeichens* (`\0`).

Beispiel: Zeichenkette ``abcd`` wird abgelegt als `a``b``c``d``\0`. Die Länge dieser Zeichenkette beträgt 4. Der Speicherbereich ist 5 Bytes groß.

Deklarationen

Alle Variablen eines Programms müssen vor ihrer Benutzung deklariert werden, d.h. vereinbart werden. Deklarationen bestehen aus Datentyp und einer Liste von Variablen, z.B.

```
int lower, upper, step;
```

oder gleichwertig:

```
int lower; /* Anfangswert */
int upper; /* Endwert */
int step;  /* Schrittweite */
```

Initialisierung von Variablen in Deklarationen, z.B.

```
int i=0;
float eps=1.0e-5;
```

Die Initialisierung, d.h. Anfangswertzuweisung, erfolgt bereits zur *Übersetzungszeit* und hilft so *Laufzeit* einzusparen ;-)
Zu beachten ist auch der Aspekt einer klaren und sicheren Initialisierung vor einer möglichen Nutzung.

Arithmetische Operatoren

`+, -, *, /, %`

Die Division unterscheidet sich für Ganz- und Fließkommazahlen!
Modulo Operator: x%y bedeutet Rest bei der Division x durch y. **Nicht** anwendbar auf die Datentypen *float* und *double*.
Beispiel: 31%7 ergibt 3.

Bei Verknüpfungen von Operanden **unterschiedlichen** Datentyps findet eine *Typumwandlung* statt vom „*niedrigeren*“ zum „*höheren*“ Datentyp
(int < float < double)
entsprechend der Tabelle:

Verknüpfungstabelle

+, -, *, /	int	float	double
int	int	float	double
float	float	float	double
double	double	double	double

Es gibt in C keinen Operator für die Exponentiation.
Hierzu kann die Funktion **pow** in der mathematischen Bibliothek **libm.a** (#include <math.h>) verwendet werden:
pow(x,y) berechnet „x hoch y“.

Zuweisung

Auch bei der Zuweisung findet eine *Typumwandlung* statt, wenn sich der Datentyp der Variablen auf der linken Seite vom Datentyp des Ausdrucks auf der rechten Seite unterscheidet.

variable = *ausdruck*;

Der Datentyp des Ausdrucks wird in den Datentyp der Variablen gewandelt.
Explizite Angabe des Zieldatentyps: **cast**.

variable = (Datentyp *variable*) *ausdruck*

Inkrement- und Dekrement-operatoren(I)

Präfix-	**Post**fix-Notation	Bedeutung
`++i`	`i++`	`i=i+1` (inkrementieren)
`--i`	`i--`	`i=i-1` (dekrementieren)

Der Unterschied zwischen Präfix- und Postfix-Notation wird deutlich, wenn eine weitere Operation beteiligt ist:
Beispiel 1:

```
x = ++i; bedeutet  i = i+1; x = i;
x = i++; bedeutet  x = i;   i = i+1;
y = --i; bedeutet  i = i-1; y = i;
y = i--; bedeutet  y = i;   i = i-1;
```

Inkrement- und Dekrement-operatoren (II)

Beispiel 2:

`++i*k` bedeutet: erst i inkrementieren, dann i mit k multiplizieren.

`i++*k` bedeutet: erst i mit k multiplizieren, dann i inkrementieren.

`--j*k` bedeutet: erst j dekrementieren, dann j mit k multiplizieren.

`j--*k` bedeutet: erst j mit k multiplizieren, dann j dekrementieren.

Kurzform von Zuweisungsausdrücken

```
i = i+k;  entspricht  i += k;
i = i-k;  entspricht  i -= k;
i = i*k;  entspricht  i *= k;
i = i/k;  entspricht  i /= k;
i = i%k;  entspricht  i %= k;
```

Wenn die Variable auf der linken Seite einer Zuweisung auf der rechten Seite **nochmals** auftritt, kann eine verkürzte Form der Zuweisung verwendet werden.

Erste Programmbeispiele (I)

Kommentare sind zwischen **/*** und ***/** eingeschlossen, können aber nicht geschachtelt werden. Kommentare, Einrückungen und Zeilenumbrüche erhöhen die Lesbarkeit von Programmen.

C Programme bestehen aus Funktionen. Jedes C Programm muss eine Funktion namens main() enthalten. Die Funktion **main()** bildet das Hauptprogamm. Alle anderen Funktionen sind mit Unterprogrammen (Fortran) oder Prozeduren (Pascal) vergleichbar. Funktionen werden durch Paare geschweifter Klammern in Blöcke eingeteilt.

Erste Programmbeispiele (II)

```
{ /* Anfang eines Blockes */
} /* Ende eines Blockes */
```

Funktionsrumpf: äußerer Block einer Funktion.
Innerhalb eines Blocks stehen Anweisungen. Sie werden durch ein Semikolon (**;**) abgeschlossen. Ein Semikolon hinter der schließenden Klammer eines Blocks entspricht einer *leeren* Anweisung.
Alle in einem C Programm verwendeten Variablen müssen *explizit deklariert* werden. Durch Verschreiben können keine neuen Variablen eingeführt werden.

Ein- und Ausgabe (C)

scanf(*format*, arg1, arg2,…);
Eingabe der Variablen arg1, arg2,… *von der Tastatur* entsprechend dem Formatausdruck format.
Dabei müssen die Argumente arg1, arg2,… *Zeiger* sein.
Dies wird bei einfachen Variablen durch das Voranstellen eines *Adressoperators* (**&**) vor dem Variablennamen erreicht.

printf(*format*, arg1, arg2,…);
Ausgabe der Variablen arg1, arg2,… *auf dem Bildschirm* entsprechend dem Formatausdruck format.

***format**:* zwischen ``und`` eingeschlossener Textausdruck mit:
- Formatspezifikationen, die den zu übertragenden Variablen entsprechen (siehe Tabelle)
- Texte, die auszugeben sind.
- Steuerzeichen, z.B. **\t** horizontaler Tabulator, **\n** neue Zeile

Formatspezifikationen bei Ein- und Ausgabe (I)

Datentyp	Eingabeformat	Ausgabeformat
int	%d	%d oder %wd
float	%f	%f oder %w.pf
		%e oder %w.pe
double	%lf	%f oder %w.pf
		%e oder %w.pe
char	%c	%c
char[ ]	%s	%s oder %ws

Formatspezifikationen bei der Ein- und Ausgabe (II)

char[ ]: Feld vom Datentyp char zur Aufnahme von Zeichenketten.

w: Feldbreite

p: Anzahl der Stellen hinter dem Dezimalpunkt

Beachte:

- Eingabe des Datentyps **double** mit **%lf**, nicht mit %f.
- %e oder %w.pe bewirkt Ausgabe mit Exponent (Zehnerpotenz).

Ein- und Ausgabe (C++)

Voraussetzung für Ein- und Ausgabefunktionen
ist die Einbindung der Bibliothek

```
#include <iostream>
```

Ausgabe von Daten **auf** die Konsole mit impliziter Umwandlung in Text:

```
std::cout << “Bitte geben Sie Laenge, Breite und Hoehe\
eines Quaders ein:\n“;
```

Eingabe von Text **von** der Tastatur mit impliziter Umwandlung in
den richtigen Datentyp:

```
float laenge, breite, hoehe;
std::cin >> laenge >> breite >> hoehe ;
```

Präprozessordirektiven (I)

Ihre Auswertung erfolgt durch den *Präprozessor* vor der Übersetzung des C Programms durch den C Compiler.

Eigenschaften von Präprozessordirektiven:

- stehen am Programmanfang
- werden durch **#** eingeleitet
- werden nicht mit einem **;** abgeschlossen
- ein **** am Zeilenende zeigt ihre Fortsetzung in der Folgezeile an.

Präprozessordirektiven (II)

Beispiel 1: Definition einer symbolischen Konstanten (in Großbuchstaben):

#define PI 3.141593

Dort, wo die symbolische Konstante **PI** innerhalb des Programms auftritt, wird sie durch den Zahlenwert aus der **#define** Direktive ersetzt.

Beispiel 2: Einfügen einer Deklarationsdatei (Header Datei)

a) **#include <stdio.h>** stdio.h enthält die Deklarationen der Standardfunktionen zur Ein- und Ausgabe, z.B. *scanf* und *printf*.
Die zwischen **<** und **>** angegebene Datei wird in einem Systemverzeichnis gesucht.

Präprozessordirektiven (III)

Beispiel 2: Einfügen einer Deklarationsdatei (Header Datei)

b) **#include ``myheader.h``**

Dabei sei myheader.h eine benutzer-eigene Deklarationsdatei. Eine zwischen `` und `` angegebene Datei wird im aktuellen Verzeichnis gesucht.

Aufruf mathematischer Standardfunktionen (I)

In der Bibliothek **libm.a,** die in einem Systemverzeichnis liegt, sind mathematische Standardfunktionen zusammengefasst, wie
- *trigonometrische* Funktionen: **sin**, **cos, tan**,.., **asin**, **acos**, **atan**,..
- *Bessel*funktionen
- *hyperbolische* Funktionen: **sinh**,**cosh**,**tanh**,**asinh**,**acosh**,**atanh**
- verschiedene Funktionen wie: **sqrt, fabs, exp, pow, log, …**

Die Deklarationsdatei zu dieser Bibliothek heißt **math.h** und liegt in einem Systemverzeichnis.

#include <math.h> Präprozessordirektive zum Einfügen von **math.h** in ein C Programm

Bedingte Programmierung

Bedingte Anweisungen

Durch bedingte Anweisungen kann der Ablauf eines Programms von Bedingungen abhängig gemacht werden, die erfüllt bzw. **wahr** sind oder nicht erfüllt bzw. **falsch** sind. Viele Bedingungen lassen sich mit logischen Ausdrücken formulieren. Eine besonders wichtige Gruppe logischer Ausdrücke sind die Vergleichsausdrücke.

Vergleichsausdrücke

Logische Ausdrücke, die durch die Verknüpfung zweier Operanden vom arithmetischen Datentyp (Integer-Datentyp oder Floating-Datentyp) mit einem Vergleichsoperator entstehen.

Vergleichsoperator	Bedeutung
>	größer
>=	größer oder gleich
<	kleiner
<=	kleiner oder gleich
==	gleich
!=	ungleich

Beispiele: `0<=x, x!=100, y==1`

Zusammengesetzte Vergleichsausdrücke

Sie entstehen durch die Verknüpfung von einfachen Vergleichsausdrücken mit Hilfe logischer Operatoren.

logischer Operator	Bedeutung
!	Negation
&&	logisches *und*
||	logisches *oder*

Beispiel: `0<=x && x<=1` (mathematisch: 0<= x <=1)

Bezeichnungen

anweisung: einzelne Anweisung oder eine Gruppe von Anweisungen, die durch geschweifte Klammern zu einem **Block** zusammengefasst sind.

ausdruck: Ausdruck vom arithmetischen Datentyp. Logische Ausdrücke gehören dazu, da sie vom arithmetischen Datentyp (int) sind.

ausdruck **!=** 0 bedeutet *wahr*.
ausdruck **==** 0 bedeutet *falsch*.

In anderen Programmiersprachen (Fortran und Pascal) besitzen logische Ausdrücke einen eigenen Datentyp (LOGICAL bzw. BOOLEAN).

Die if-else Anweisung (I)

Sie hat die allgemeine Form:

```
if(ausdruck)
  anweisung_1;
else
  anweisung_2;
```

Falls *ausdruck* != 0, d.h. wahr ist, folgt anweisung_1.

Falls *ausdruck* == 0, d.h. falsch ist, folgt anweisung_2.

Der **else**-Teil in dieser Konstruktion kann fehlen.

Bei einer Folge ineinander geschachtelter **if**-Anweisungen wird eine **else**-Anweisung immer der nächststehenden **if**-Anweisung ohne **else** zugerechnet, z.B.

Die if-else Anweisung (II)

```
if(n>0)
  if(a>b)
     z = a;
  else
     z = b;
```

Hier gehört **else** zum *inneren* **if**.

Soll der **else** Teil dagegen zum ersten **if** gehören, ist dies durch geschweifte Klammern zum Ausdruck zu bringen.

```
if (n>0) {
  if(a>b)
     z = a;
  }
else
  z = b;
```

Hier gehört **else** zum *äußeren* **if**.

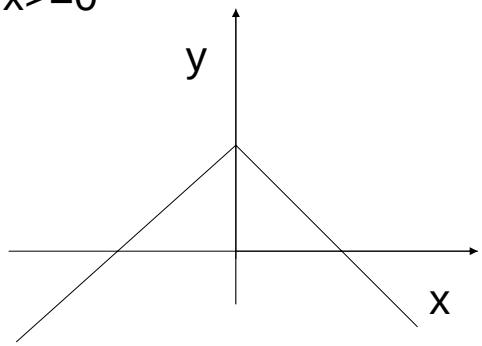
Die if-else Anweisung (III)

Beispiel: Auswertung einer zweiteiligen Funktion

y = f(x) = x+1 für x<0

y = f(x) = -x+1 für x>=0

```
if(x<0)
  y = x+1;
else
  y = -x+1;
```



Die „else if“-Anweisung (I)

Folgende Konstruktion ermöglicht in Abhängigkeit von mehreren Ausdrücken *ausdruck_1*, *ausdruck_2*,…, *ausdruck_n* mehr als zwei Wege für die Programmfortsetzung.

if(*ausdruck_1*)
  anweisung_1;
else if (*ausdruck_2*)
  anweisung_2;
…..
else if (*ausdruck_n*)
  anweisung_n;
else
  anweisung;

Gilt *ausdruck_i* **!=** 0 für einen Index i, so folgt *anweisung_i*.

Gilt *ausdruck_i* **==** 0 für alle i folgt *anweisung* hinter else.

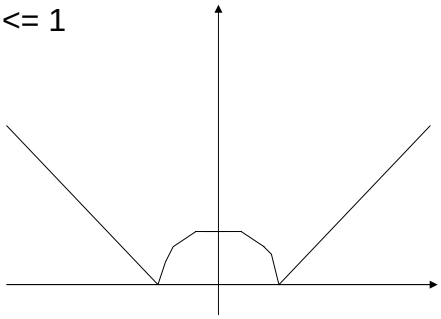
Der else-Teil ist wieder optional, d.h. er kann vorhanden sein oder auch fehlen.

Die „else if“-Anweisung (II)

Beispiel 1: Auswertung einer dreiteiligen Funktion

y = g(x) = -1-x für x < -1
y = g(x) = 1-x² für -1 <= x <= 1
y = g(x) = -1+x für 1 < x

```
if(x<-1)
  y = -1-x;
else if(-1<=x && x<=1)
  y = 1-x*x;
else
  y = -1+x;
```



Beispiel 2: Bedeutung von Zensuren und
Zuordnung von rechnerischen Werten auf zulässige Noten (2,3)

```
if(x<-1) // Schreibalternative 1
   y = -1-x;
else if(-1<=x && x<=1)
   y = 1-x*x;
else
   y = -1+x;


if(x<-1) // Schreibalternative 2
   y = -1-x;
else
   if(-1<=x && x<=1)
      y = 1-x*x;
   else
      y = -1+x;
```

Konditionalausdrücke (bedingte Bewertung)

Die if-else Anweisung in der Form

```
if(bedingung)
  var = wert1;
else
  var = wert2;
```

kann mit dem dreizähligen Operator **?:** als *Konditionalausdruck* geschrieben werden:

```
var = (bedingung) ? wert1 : wert2;
```

Die switch-Anweisung (I)

Sie dient zur Formulierung von Mehrwegentscheidungen, ähnlich wie eine **if** Konstruktion mit **else if** Anweisungen. Allgemeine Form der **switch** Anweisung:

```
switch(ausdruck) {
    case konst_1: anweisung_1; break;
    case konst_2: anweisung_2; break;
    ….
    case konst_n: anweisung_n; break;
    default: anweisung;
  }
```

Hierbei bezeichnen:

ausdruck: Ausdruck vom Datentyp **int** oder **char**.

konst_1, *konst_2*,…,*konst_n*: verschiedene Konstanten vom Datentyp **int** oder **char**.

Die switch-Anweisung (II)

Gilt *ausdruck* == *konst_i* für einen Index i mit i=1,2,…,n, wird *anweisung_i* ausgeführt. Eine **break** Anweisung am Ende von *anweisung_i* führt zum Verlassen der **switch** Anweisung. Auf diese Weise werden nur die Anweisungen eines Falles abgearbeitet.
Gilt *ausdruck* != *konst_i* für alle Indizes i, werden die Anweisungen hinter der Marke **default** ausgeführt.
Alle **case** Konstanten *konst_1*, *konst_2*, …, *konst_n* müssen verschieden sein.

```
#include <iostream>
using namespace std;

int main() // Demo einer bösen Falle
{  char a;
   int x,y,z;

   x=21; y=17; z=4; // Initialisierung und Kontrolle
   cout << "x=" << x << ", y=" << y << ", z=" << z << endl;
   cout << "Das sollte programmiert werden:\n";
   if (x == y+z) // Das ist wirklich ein Vergleich.
       cout << "x ist tatsaechlich gleich y+z" << "\n\n";

   x=16; // nur aus didaktischen Gründen
   cout << "x=" << x << ", y=" << y << ", z=" << z << endl;
   cout << "Das kann passieren:\n";
   if (x = y+z) // versehentlich Zuweisung!!
      // ^         aber (Wert!=0) --> "Bedingung" erfüllt!
      cout << "x=" << x << ", y=" << y << ", z=" << z << endl
         << “Was ist hier passiert?“ << endl;

   cout << endl << "Ende? "; cin >> a;
   return 0;
}
```

Schleifen

Schleifen

Schleifenkonstruktionen dienen zur wiederholten Ausführung einer einzelnen Anweisung oder einer zu einem Block zusammengefassten Gruppe von Anweisungen.
Arten von Schleifen:

- **while** Schleife (abweisend)
- **do-while** Schleife (nicht abweisend)
- **for** Schleife

Die **while** Schleife hat die allgemeine Form:

```
while(ausdruck)
      anweisung
```

Falls *ausdruck* != 0 gilt, wird *anweisung* ausgeführt.
Falls *ausdruck* ==0 gilt, wird *anweisung* nicht ausgeführt.
Die **while** Schleife heißt daher auch „abweisende“ Schleife.

Die do-while Schleife

Sie hat die allgemeine Form:

```
do
    anweisung
while(ausdruck);
```

Im Gegensatz zur **while** Schleife wird *anweisung* bei der **do-while** Schleife wenigstens einmal durchlaufen, unabhängig vom Wert von *ausdruck*. Eine **do-while** Schleife ist daher auch eine „nichtabweisende“ Schleife.
Falls *ausdruck* != 0 gilt, wird *anweisung* wiederholt.
Falls *ausdruck* ==0 gilt, wird *anweisung* nicht wiederholt.
Das Semikolon (**;**) hinter der **while** Anweisung darf nicht fehlen. Sonst folgt ein Syntaxfehler beim Übersetzen.

Die for Schleife (I)

Sie hat die allgemeine Form:

```
for(ausdruck_1; ausdruck_2; ausdruck_3)
  anweisung
```

mit den Bezeichnungen:

ausdruck_1: Initialisierungsausdruck
- wird nur einmal ausgewertet.
- bewirkt Anfangswertzuweisung an die Schleifenvariable

ausdruck_2: Schleifenbedingung
Gilt *ausdruck_2* **!=** 0 wird *anweisung* ausgeführt.
Gilt *ausdruck_2* **==**0 wird *anweisung* nicht ausgeführt.

ausdruck_3: Reinitialisierungsausdruck
- wird im Anschluss an *anweisung* ausgewertet.
- bewirkt in der Regel eine Abänderung der Schleifenvariablen.

Die for Schleife (II)

Eine **for** Schleife ist äquivalent zu folgender **while** Schleife:

```
ausdruck_1;
while(ausdruck_2) {
  anweisung
  ausdruck_3;
}
```

Mit Hilfe des Komma Operators (**,**) können in einer **for** Anweisung statt eines einzelnen Ausdrucks mehrere Ausdrücke angegeben werden, z.B.
für *ausdruck_1* zwei Initialisierungsausdrücke oder
für *ausdruck_3* zwei Reinitialisierungsausdrücke.

Sonderfälle (I)

In einer **for** Schleife kann jeder der Ausdrücke *ausdruck_1*, *ausdruck_2*, *ausdruck_3* fehlen.
„Unendliche“ Schleife als

for Schleife	**while** Schleife
for(; ;) {	**while(1)** {
….	….
}	}

Die Schleifenbedingung wird immer als „wahr“ angesehen.

Die **break** Anweisung bewirkt, daß die innerste umgebende **for**, **while** oder **do-while** Schleife verlassen wird, unabhängig von der Schleifenbedingung.

Sonderfälle (II)

Die **continue** Anweisung bewirkt, dass der aktuelle Schleifendurchlauf abgebrochen und die Schleife mit der nächsten Iteration fortgesetzt wird. Es kommt aber nicht wie bei **break** zum Abbruch der gesamten Schleife.
Mit der **goto** Anweisung kann innerhalb einer Funktion zu einer Anweisung gesprungen werden, die durch eine Marke gekennzeichnet ist.

```
for(…)
  for(…) {
     if(schwerer_fehler) goto error;
  }
error:  fehler_behandlungs_anweisung
```

Bemerkung: In vielen Fällen kann die **goto** Anweisung beim Programmieren in C vermieden werden.

Felder

Felder

Zu jedem *elementaren* Datentyp „**t**“ kann man den Datentyp „**Feld von t**“ bilden.

Felder sind benannte Folgen von Objekten *gleichen* Datentyps.
Dabei sind die einzelnen Objekte in aufeinander folgenden Speicherbereichen abgelegt.

Deklaration von Feldern (I)

int v[10]; vereinbart *eindimensionales* Feld vom Datentyp **int** der Länge 10, d.h. einen Block von 10 aufeinander-folgenden Speicherplätzen: **v[0], v[1],…., v[9].**

Ein Element eines eindimensionalen Feldes wird über einen Index angesprochen; der Index **0** bezeichnet das erste, der Index **n-1** das n-te Element.

char s[6]; vereinbart eindimensionales Feld vom Datentyp **char** der Länge 6 zur Aufnahme einer Zeichenkette der Länge 5. Das letzte Element **s[5]** dient zur Aufnahme des abschließenden Nullzeichens **`\0`.**

Deklaration von Feldern (II)

float a[10][20]; vereinbart ein *zweidimensionales* Feld vom Datentyp **float** mit 10 Zeilen und 20 Spalten.

a[i][j] bezeichnet das Element des Feldes **a** in der **i**-ten Zeile mit 0 <= **i** <= 9 und der **j**-ten Spalte mit 0 <= **j** <= 19.

Die Elemente eines zweidimensionalen Feldes werden in C, im Unterschied zu Fortran, zeilenweise abgelegt.

Initialisierung von Feldern

Wie bei einfachen Variablen können Felder bereits bei ihrer Deklaration initialisiert werden, z.B.

```
int v[ ] = {0,1,2,3,4};
char s[ ] = ``0123456789``;
float a[ ][4] = {{1,2,3,4}, {2,3,4,5}, {3,4,5,6}};
```

Wird die Größe eines Feldes bei der Deklaration nicht angegeben (Feld unbestimmter Länge), so ermittelt sie der Compiler durch Zählen der

- Elemente eines eindimensionalen Feldes.
- Zeichen einer Zeichenkette, vermehrt um das abschließende Nullzeichen **`\0`.**

Bei der Deklaration zweidimensionaler Felder ist wenigstens die Anzahl der Spalten anzugeben, die Anzahl der Zeilen kann der Compiler durch Abzählen ermitteln.

Eingabe von Feldern

```
int feld[10],i;

for (i=0;i<10;i++)
 cin >> feld[i];
```

Ausgabe von Feldern

```
int feld[10],i;

for (i=0;i<10;i++)
 cout << feld[i] << " ";
```

Warnung

Das Benutzen von Feldern über die deklarierten Feldgrenzen hinaus wird in C/C++ nicht kontrolliert und nicht unterbunden. Das kann bei oberflächlicher Programmierung gefährliche Folgen haben, da meist unbewusst und ungewollt Daten verändert werden.

Ein- und Ausgabe von Feldern

Eingabe von Feldelementen mit **scanf**
- vor dem Feldelement muss der *Adressoperator* stehen.

Eingabe einer Zeichenkette mit **scanf**
- der Feldname als Argument bewirkt die Übergabe der Adresse des ersten Feldelements.
- Ein Feld vom Datentyp **char** sollte um ein Feldelement länger sein als die abzulegende Zeichenkette.

Zur Steuerung der elementweisen Ein- und Ausgabe (E/A) von Vektoren und Matrizen können **for** Schleifen verwendet werden:
- *einfache* **for** Schleife zur E/A der Komponenten eines *eindimensionalen* Feldes (Vektor).
- *zwei* geschachtelte **for** Schleifen zur E/A der Elemente eines *zweidimensionalen* Feldes (Matrix).

Funktionen

Funktionen

Funktionen sind Bausteine, aus denen C-Programme aufgebaut werden. Jedes ausführbare C-Programm muss genau eine Funktion namens **main** besitzen. Sie bezeichnet das *Hauptprogramm*. Die einfachste Form der **main**-Funktion lautet:

```
int main()
{
   deklarationen
   anweisungen
}
```

Darüber hinaus gehende Funktionen sind mit subroutine/ function in Fortran oder procedure/function in Pascal vergleichbar.

Funktionen

Funktionen können in C dazu dienen, eigenständige Programmsegmente zu formulieren und so das Gesamtprogramm übersichtlicher zu machen.
Bis jetzt wurden bereits folgende Funktionen aus der C Programmierumgebung benutzt:

- **scanf, printf** zur Ein- und Ausgabe deklariert in **stdio.h**
- **sqrt, fabs, pow** aus mathematischer Bibliothek deklariert in **math.h**
- **strlen** String Funktion deklariert in **string.h**

Bezeichnungen:

Parameter (Formalparameter): Variable, die bei der *Definition* der Funktion hinter dem Funktionsnamen in der Parameterliste steht.

Argument (Aktualparameter): Variable, die beim *Aufruf* einer Funktion in der Parameterliste steht.

Funktionen gemäß ANSI-C

Funktions*deklaration* vor der main Funktion (Funktionsprototyp):

```
rueckgabe_datentyp
   function_name(parameter_datentypen);
```

Funktions*definition*:

```
rueckgabe_datentyp
   function_name(parameter_deklarationen)
{
    deklarationen
    anweisungen
}
```

Funktionen nach altem Stil

Funktions*deklaration* vor der **main** Funktion:

```
rueckgabe_datentyp function_name( );
```

Funktions*definition*:

```
rueckgabe_datentyp
   function_name(parameter_liste)
deklarationen_parameter_liste
{
    deklarationen
    anweisungen
}
```

Gegenüberstellung beider Funktionsschreibweisen

Die unter ANSI-C eingeführte neue Syntax von Funktionsprototypen ermöglicht es dem Compiler zur Übersetzungszeit, beim Funktionsaufruf *Fehler in der Anzahl und in den Datentypen der Argumente* festzustellen.
Für Funktionen im ANSI-C und im alten Stil gilt:

- mit Hilfe der **return** Anweisung kann ein skalares Ergebnis aus einer Funktion an die rufende Funktion zurückgegeben werden.
- Namen von Parametern und Variablen einer Funktion gelten nur innerhalb dieser Funktion. Andere Funktionen können dieselben Variablen und Parameternamen verwenden.

Call by value

Die Argumente einer Funktion, die einfache Variable sind, werden in C durch „*call by value*“ übergeben. Das bedeutet:

- Der aufgerufenen Funktion werden die Argumente als *temporäre* Variable (Kopie) übergeben, *nicht als Originale*.
- Diese *temporären* Variablen (Kopien) können innerhalb der Funktion verändert werden, ohne dass die *Originale* in der rufenden Funktion verändert werden.

Felder als Funktionsparameter (I)

Sind Felder Funktionsparameter, wird beim Funktionsaufruf nur die *Adresse des ersten Feldelementes* übergeben („*call by reference*“).
Dies hat folgende Auswirkungen:

- Anders als bei einfachen Variablen wird ein Feld, das als Argument an eine Funktion übergeben wird, in gleicher Weise auch in der aufrufenden Funktion verändert.
- Beim Funktionsaufruf mit Feldern als Argumenten findet keine Feldgrenzenüberprüfung statt.

Felder als Funktionsparameter (II)

*Ein*dimensionale Felder als Funktionsparameter können mit *unbestimmter Elementzahl* definiert werden (Felder unbestimmter Länge).
Beispiel: Skalarprodukt zweier Vektoren.
Bei *zwei-* und höherdimensionalen Feldern als Funktionsparametern darf nur die Größenangabe für die erste Dimension entfallen.

Funktion auf Parameterposition (I)

Beispiel: Numerische Integration einer im Intervall (a,b) definierten Funktion mit der Sehnentrapezregel.

```
double strapez(double f(double), double a, double b, int n)
{  int i;
   double h,s,xi;
   h = (b-a)/n;
   s = 0.5*(f(a)+f(b));  /* Anfangswert != 0 bei Summation */
   for(i=2; i<=n; i++) {
      xi = a + (i-1)*h;
      s = s + f(xi);
   }
   return s*h;
}
```

Beispiel einer nicht integrierbaren Funktion: f(x) = exp(-x*x)

Funktion auf Parameterposition (II)

Beispiel: Numerische Integration einer im Intervall (a,b) definierten Funktion mit der Tangententrapezregel.

```
double ttrapez(double f(double), double a, double b, int n)
{ int i;
  double h, s=0.0;
  h = (b-a)/n;
  for(i=1; i<=n; i++)
      s = s + f(a+(i-0.5)*h);
  return s*h;
}
```

Funktionen mit Datentyp void (leer)(I)

Bei diesen Funktionen kann über die *return* Anweisung kein Wert an die aufrufende Funktion zurückgegeben werden.
Die *Ergebnisrückgabe* erfolgt dann *über die Parameterliste*.

Beispiel 1: Berechnung des *Vektorprodukts* im 3-dim Raum.
Die Funktion vektor besitzt drei Parameter: Zwei dienen zur Übergabe der Eingabedaten, der dritte dient zur Rückgabe des Ergebnisses.

Beispiel 2: *Umkehrung einer Zeichenkette*. Die Funktion reverse besitzt einen Parameter zur Übergabe der Eingabedaten und zur Rückgabe des Ergebnisses.

Funktionen mit Datentyp void (II)

Beispiel 1: Vektorprodukt

```
#include <iostream>
using namespace std;

void kreuz(float a[], float b[], float c[])
{
   c[0] = a[1]*b[2] - a[2]*b[1];
   c[1] = a[2]*b[0] - a[0]*b[2];
   c[2] = a[0]*b[1] - a[1]*b[0];
}

int main()
{
   float x[3]={1.0,0.0,0.0};  // Deklaration der Vektoren
   float y[3]={0.0,1.0,0.0};  // und Initialisierung
   float z[3]={0.0,0.0,0.0};

   kreuz(x,y,z);              // Aufruf der Funktion

   cout << z[0] << ' ' << z[1] << ' ' << z[2] << endl;

   return 0;
}
```

Rekursive Funktionen (I)

Eine rekursive Funktion ruft sich selbst wieder auf.

Beispiel 2: Rekursive Umkehrung einer Zeichenkette

```
void reverse(char s[], int left, int right)
/* s an Ort und Stelle rekursiv umkehren */
{
   char c;
   cout << s << ' ' << left << ' ' << right << endl;
   if (left < right) {
      c = s[left];
      s[left] = s[right];
      s[right] = c;
      reverse(s,left+1, right-1);
   }
}
```

Rekursive Funktionen (II)

Beispiel 3: Fakultät rekursiv berechnen: n! = n*(n-1)!

```
long int facultaet(long int x)
{
   if (x > 1) // ohne Überlaufcheck!!
      return x*facultaet(x-1);
   else
      return 1;
}
```

Rekursive Funktionen (III)

Bei rekursiven Aufrufen von Funktionen ist zu beachten, dass die **Rekursionstiefe abschätzbar begrenzt** ist, und dass es überhaupt einen Abbruch der Rekursion gibt.

Bei Nichtbeachtung läuft man Gefahr, einen Stackoverflow zu verursachen, denn bei jedem neuen Aufruf der Funktion werden die lokalen Variablen der jeweiligen Funktionsinstanz auf dem Stapelspeicher abgelegt. Der Speicher ist dann stackseitig voll und es geht nichts mehr.

Dateien

Arbeiten mit Dateien

Irgendwann kommt der Wunsch auf, Informationen nicht nur über die Standardein- und -ausgabe zu leiten.

Da unsere Betriebssysteme strukturierte Dateisysteme zur Verfügung stellen und die Datenumfänge in Größenordnungen vorkommen, die nicht mehr von Hand zu bewältigen sind, liegt es nahe, aus Dateien zu lesen und in Dateien zu schreiben.

Dazu gibt es die Bibliothek `fstream`

Arbeiten mit Dateien (II)

Dateien müssen benannt werden. Hier eröffnet sich eine Problematik, die in den Unterschieden der Betriebssysteme liegt. Hier sollten die Besonderheiten der Betriebssysteme derart berücksichtigt werden, dass verwendete Namen in allen vorgesehenen BS eindeutig sind und funktionieren.
Eine weitere Problematik liegt in unterschiedlichen Zeilenende-Zeichen in Textdateien. Das kommt zum Tragen, wenn die benutzten Textdateien zwischen unterschiedlichen BS ausgetauscht werden sollen.

Dateien zum Lesen

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

string wort;

ifstream ifile(“eingabe.txt“);

if (!ifile.eof())
  ifile >> wort; // lesen bis zum Whitespace

cout << “Erstes Wort: “ << wort << endl;
```

Dateien zum Schreiben

```
#include <fstream>
#include <iostream>
using namespace std;

ofstream ofile(“ausgabe.txt“);

ofile << “Das ist eine Probezeile!\n“;

ofile.close(); // Datei schließen
```

Textdateien

Das Lesen aus Dateien und Schreiben in Dateien ist formal das Gleiche wie bei den Zeichenstreams für die Consolen-Ausgabe (cout) und die Consolen-Eingabe (cin).

Es werden also beim Lesen Zeichenfolgen gelesen und standardmäßig möglichst sinnvoll interpretiert. Beim Schreiben werden Inhalte von Variablen und Konstanten in für Menschen lesbare Zeichenketten umgewandelt und ausgegeben.

Abweichend vom Standard kann in die Funktionsweise beim Lesen und Schreiben eingegriffen werden.

Textdateien, Whitespaces

Beim Lesen von Texten werden Zeichenmengen zwischen sog. Whitespaces als zusammen gehörig verstanden und einer angegebenen Variablen typentsprechend zugewiesen.

Whitespaces sind Leerzeichen, Tabulatorzeichen und Zeilenwechsel oder Vielfache davon. Alle aufeinander folgenden Whitespace-Zeichen werden zusammen als ein Whitespace aufgefasst.

Binärdateien

Man kann Informationen auch in binärer Form aus Dateien lesen und in Dateien schreiben. Diese Dateien sind jedoch nicht sinnvoll mittels Texteditor les- und veränderbar.

Die Lese- und Schreibzugriffe sind aber bedeutend effizienter, da die umfangreichen Interpretationen und Umwandlungen entfallen.

Binärdateien

Beispiel:

```
ofstream outbin; // binäre stl-datei
float vek[3];
…

outbin.write((char*)vek,12);
…

outbin.close();
```

Datenstrukturen

Strukturen

Struktur: Ansammlung mehrere Variabler unter einem Namen. Die in einer Struktur zusammengefassten Variablen heißen *Komponenten*. Sie können *unterschiedliche Datentypen* haben.
Strukturen können zur besseren Handhabung komplizierter Daten verwendet werden.

Grundlagen (I)

Mit **struct** wird die Erklärung eines Strukturtypen eingeleitet.

```
struct struct_name{
  komponenten_deklarationen
};
```

struct_name: Name des Strukturdatentyps
{….} Liste der Deklarationen für die Komponenten.

Beispiel:
```
struct punkt{
  double x; double y;
};
```

Ist ein Strukturdatentyp einmal festgelegt worden, können Variable von diesem Datentyp deklariert werden.

```
punkt pt;
```

Grundlagen (II)

Die Initialisierung kann wie bei einfachen Variablen bereits bei der Deklaration erfolgen:

```
punkt pt = {0,1};
```

Mit dem *Strukturkomponentenoperator* ``**.**`` kann auf die einzelnen Komponenten einer Variablen **var_name** mit Strukturdatentyp Bezug genommen werden:

var_name.komponente

Beispiel: Eingabe der Komponenten der Variablen **pt** vom Datentyp **punkt**:

```
cin >> pt.x >> pt.y;
```

Beispiel: Ausgabe der Komponenten der Variablen **pt** vom Datentyp **punkt**:

```
cout << pt.x << " " << pt.y;
```

Grundlagen (III)

Bei der Deklaration weiterer Strukturtypen kann auf bereits eingeführte Strukturdatentypen zurückgegriffen werden.

Beispiel:

```
struct rechteck{
        punkt pt1;
        punkt pt2;
};
```

Der Datentyp **rechteck** besteht aus zwei Komponenten vom Datentyp **punkt**. (Beschreibung eines Rechtecks in der x-y-Ebene durch zwei sich diagonal gegenüberliegende Eckpunkte). Nach der Deklaration

```
rechteck fenster;
```

wird mit

```
fenster.pt1.x
```

die x-Koordinate des Eckpunktes **pt1** von **fenster** angesprochen.

Grundlagen (IV)

Beispiel:

```
struct kreis{
        punkt mp;
        double radius;
};
```

Der Strukturdatentyp **kreis** besteht aus zwei Komponenten: dem Mittelpunkt **mp** vom Datentyp **punkt** und dem Radius **radius** vom Datentyp **double**. Nach der Deklaration

```
kreis ks;
```

bezeichnen **`ks.mp.x`** und **`ks.mp.y`** die Koordinaten des Mittelpunktes und **`ks.radius`** den Radius von **ks**.

Strukturen und Funktionen (I)

Variable vom Strukturdatentyp

```
struct complex{
  double re; double im;
};
```

können mit der folgenden Funktion **make_cmplx** *dynamisch initialisiert* werden:

```
complex make_cmplx(double x, double y)
{  complex tmp;
   tmp.re = x;
   tmp.im = y;
   return tmp;
}
```

Strukturen und Funktionen (II)

Die nächste Funktion dient zur Addition zweier komplexer Zahlen. Sowohl der Rückgabedatentyp wie auch die Parameter sind vom Datentyp **complex**.

```
complex cadd(complex z1, complex z2)
{  complex tmp;
   tmp.re = z1.re + z2.re;
   tmp.im = z1.im + z2.im;
   return tmp;
}
```

Da die Variablen **z1** und **z2** vom Datentyp **complex** durch „call by value“ übergeben werden, kann die Variable **tmp** auch vermieden werden:

```
complex cadd(complex z1, complex z2)
{ z1.re += z2.re;
  z1.im += z2.im;
  return z1;
}
```

Strukturen zur Definition von Bitfeldern freier Größe

Beispiel: Farben in Truecolor

```
struct col_comp
{      // für additive Farbmischung
       unsinged int r:8; // rot
       unsigned int g:8; // grün
       unsigned int b:8; // blau
       unsigned int  :8; // Rest zu 32 Bits
};

col_comp hintergrund={255,255,255}; // weiß
```

Strukturen zur Definition von Bitfeldern freier Größe (II)

Bitfelder werden erst richtig nützlich, wenn man den Vereinigungstyp **union** verwendet, um auf den gleichen Speicherinhalt in unterschiedlichen Weisen zuzugreifen. Durch diese Technik kann ein Programm an Übersichtlichkeit und Effizienz gewinnen. Darin stecken allerdings auch Gefahren bei unsachgemäßem Gebrauch.

```
union farbe
{
  unsigned int wert; // Truecolor-Farbe 32 Bits
  col_comp c; // Zum Ansprechen der einzelnen Komp.
};
```

Strukturen zur Definition von Bitfeldern freier Größe (III)

Die Definition einer Farbe erfolgt dann bequem und leicht lesbar:

```
farbe warn_farbe; // Deklaration einer Zweckfarbe

warn_farbe.c.r = rot_anteil;
warn_farbe.c.g = gruen_anteil;
warn_farbe.c.b = blau_anteil;
```

statt

```
warn_farbe.wert = (blau*256 + gruen)*256 + rot;
```

Zeigertechnik

Zeiger, Adressen

Zeiger: Variable, die Adresse einer anderen Variablen enthält.

Folgende Operatoren werden in Verbindung mit Zeigern benutzt:

***Adressoperator* &:** liefert, angewendet auf ein Objekt, die Adresse dieses Objekts.

***Inhaltsoperator* *:** liefert, angewendet auf einen Zeiger, das Objekt, das unter dieser Adresse abgelegt ist.

Wirkungsweise der Operatoren & und *

`int x=3;`	Deklaration und Initialisierung
`int y=4;`	zweier Ganzzahlvariablen
`int *ip;`	Deklaration des Zeigers ip; *ip ist vom Datentyp int, bzw. ip zeigt auf den Datentyp int.
`ip = &x;`	Der Zeigervariablen ip wird die Adresse von x zugewiesen; man sagt ip zeigt auf x. Damit hat *ip denselben Wert wie x.
`y = *ip;`	Zuweisung des Wertes von *ip bzw. x an y. Der ursprüngliche Wert von y wird überschrieben.
`*ip = 0;`	Zuweisung des Wertes 0 an *ip bzw. x. Der ursprüngliche Wert von x wird überschrieben.
`ip = &z[0];`	Die Zeigervariable ip zeigt auf das Anfangselement des Feldes z.

Zeiger und Funktionsargumente (I)

Mit Argumenten vom Zeigertyp kann man aus einer Funktion heraus Objekte in der aufrufenden Funktion (z.B. main) ansprechen und verändern.
Beispiel: Auswechseln der Werte zweier Variabler

```
#include <iostream>
using namespace std;

void swap(int *px, int *py)  // px zeigt auf a, py zeigt auf b
{  int tmp;                  // d.h. *px und a bzw. *py und b
   tmp = *px;                // sind dieselben Werte
   *px = *py;
   *py = tmp;
}

int main()
{  int a=4, b=5;
   cout << a << b;
   swap(&a, &b);  // Aufruf: Übergabe der Adressen von a und b
   cout << a << b;
}
```

Zeiger und Funktionsargumente (II)

Rückgabe eines Ergebnisses aus einer Funktion in die aufrufende Funktion mittels einer Zeigervariablen, d.h. ohne Verwendung der return Anweisung.

Beispiel 1: Von einer Punktmenge wird einer als dem Schwerpunkt am nächsten ermittelt, und die Adresse des beschreibenden Vektors in einem Parameter (call-by-reference) zurück geliefert.

Beispiel 2: Wenn mehrere Ergebnisse zurück geliefert werden müssen, z.B. Mittelwert und Standardabweichung, kann dieses über Parameter (call-by-reference) erfolgen.

Zeiger und Felder

In C besteht zwischen Zeigern und Feldern eine ausgeprägte Verwandschaft. Jede Operation, die durch die Indizierung von Feldelementen ausgedrückt werden kann, kann auch mit Zeigern formuliert werden.

int *pa; pa ist Zeiger auf den Datentyp int.
pa=&a[0]; pa zeigt auf das 0-te Feldelement von a, d.h. pa enthält die Adresse von a[0].
pa=a; pa zeigt auf das 0-te Feldelement von a, d.h. pa enthält die Adresse von a[0].

Drei Möglichkeiten, die Elemente eines Feldes a anzusprechen:
a[i], *(pa+i), *(a+i)

Enger Zusammenhang von Indizieren und Zeigerarithmetik:
pa+i bedeutet Adresse des i-ten Objekts hinter demjenigen, auf welches pa zeigt.

Zeiger und Strukturen

Beispiel: Punkte, Dreiecke und Tetraeder im 3D-Raum

```
struct pnkt3d{float x; float y; float z;};
struct dreieck{pnkt3d *p1; pnkt3d *p2; pnkt3d *p3;};
struct tetraeder{dreieck *d1; dreieck *d2;
                 dreieck *d3; dreieck *d4;};
pnkt3d p[100];   // Feld mit max. 100 Punkten
dreieck d[50];   // Feld mit max. 50 Dreiecken
tetraeder t[10]; // Feld mit max. 10 Tetraedern
p[0].x=10.0; p[0].y=20.0; p[0].z=5.0; // z.B. Punkt0
d[2].p1=&p[3]; d[2].p2=&p[8]; d[2].p3=&p[0]; // D2
t[0].d1=&d[2]; t[0].d2=&d[3]; // z.B. Tetraeder0
  t[0].d3=&d[4]; t[0].d4=&d[7];

p[8].z-=6.25; // Absenkung (z-Richtung) von Punkt8
/* Welche Konsequenzen hat das für Tetraeder0? */
```

Dynamisch verwendeter Speicher

Zu Zeigervariablen können vom Betriebssystem typentsprechende Speicherstücke angefordert werden. Das ist die Möglichkeit, Speicher flexibel zu verwenden.

```
int *a;       // Das ist nur eine Zeigervariable

a = new int; // Jetzt zeigt sie auf eine neue
             // int-Speicherzelle
```

In dieser Form ist das natürlich weder nützlich noch verständlich. Interessant wird die Nutzung von dynamischem Speicher beim Aufbau flexibler Datenstrukturen.

Zeigerstrukturen

Mit **Zeigerstrukturen** bei Verwendung von **dynamischem Speicher** kann man der primitiven Vektorstruktur und festgelegten Größe von Feldern entrinnen. Es können komplizierte Netzstrukturen aufgebaut werden, wobei immer nur so viel Speicher verwendet wird, wie auch tatsächlich gebraucht wird.

Beispiel: Beliebig viele Kinder treffen sich auf einem Spielplatz und bilden einen Kreis. Dann spielen sie ein Abzählreimspiel, bei dem ein Kind nach dem anderen wieder aus dem Kreis ausscheidet.

```
struct t_kind{
  string name;
  t_kind *next;
};
```

Zeigerstrukturen (II)

```
struct t_kind{      // Strukturtyp
  string name;
  t_kind *next;
};

t_kind *neues_kind; // Zeigervariable
neues_kind = NULL;  // Initialisierung

neues_kind = new t_kind; // Anforderung von Speicher
                         // passend für t_kind
// und Speicherung der Adresse in neues_kind

neues_kind->name = “Susi“; // Belegung des Speichers
neues_kind->next = NULL;
```

Zeigerstrukturen (III)

```
t_kind *aktuelles_kind; aktuelles_kind = NULL;

aktuelles_kind = neues_kind; /* Sichern der Adresse
  neues_kind */

neues_kind = new t_kind; // weiteren Speicher holen
neues_kind->name = “Peter“; // und nutzen
neues_kind->next = aktuelles_kind; // verketten
aktuelles_kind = neues_kind; /* zeigt auf das
  neueste Kind */
```

Diese Gruppe von Anweisungen kann wiederholt werden, um beliebig viele Kinder in einer Zeigerkettenstruktur anzulegen.

Zeigerstrukturen (IV)

Das Wandern in der Zeigerstruktur erfolgt durch wiederholtes:

```
aktuelles_kind = aktuelles_kind->next;
```

Das Erkennen des Endes einer Zeigerstruktur erfolgt bei vorheriger sauberer Initialisierung aller Zeiger z.B. so:

```
while (aktuelles_kind->next != NULL)
    aktuelles_kind = aktuelles_kind->next;
```

oder

```
if (aktuelles_kind->next == NULL)
    cout << “Diese ist das Letzte in der Kette\n“;
```

Freigeben von dynamischem Speicher

Wenn eine Zeigervariable eine gültige Speicheradresse trägt, kann und sollte der Speicher nach Gebrauch wieder freigegeben werden, damit er für andere Zwecke wieder verfügbar wird.

```
t_kind *raus_kind;
// …
raus_kind = aktuelles_kind; // Eine Adresse zuweisen
// …
delete raus_kind; // Jetzt ist der Speicher wieder frei
                  // und die Adresse unsinnig
raus_kind = NULL; // deshalb uminitialisieren
```

Anhang

ASCII CODE Tabelle

Byte

Bit	7	6	5	4	3	2	1	0
Wert	128	64	32	16	8	4	2	1

hexadezimal dezimal

B6 B5 B4 / B3 B2 B1 B0	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
0 0 0 0	00 NUL 0	10 DLE 16	20 SP 32	30 0 48	40 @ 64	50 P 80	60 ` 96	70 p 112
0 0 0 1	01 SOH 1	11 DC1 17	21 ! 33	31 1 49	41 A 65	51 Q 81	61 a 97	71 q 113
0 0 1 0	02 STX 2	12 DC2 18	22 " 34	32 2 50	42 B 66	52 R 82	62 b 98	72 r 114
0 0 1 1	03 ETX 3	13 DC3 19	23 # 35	33 3 51	43 C 67	53 S 83	63 c 99	73 s 115
0 1 0 0	04 EOT 4	14 DC4 20	24 $ 36	34 4 52	44 D 68	54 T 84	64 d 100	74 t 116
0 1 0 1	05 ENQ 5	15 NAK 21	25 % 37	35 5 53	45 E 69	55 U 85	65 e 101	75 u 117
0 1 1 0	06 ACK 6	16 SYN 22	26 & 38	36 6 54	46 F 70	56 V 86	66 f 102	76 v 118
0 1 1 1	07 BEL 7	17 ETB 23	27 ' 39	37 7 55	47 G 71	57 W 87	67 g 103	77 w 119
1 0 0 0	08 BS 8	18 CAN 24	28 (40	38 8 56	48 H 72	58 X 88	68 h 104	78 x 120
1 0 0 1	09 HT 9	19 EM 25	29) 41	39 9 57	49 I 73	59 Y 89	69 i 105	79 y 121
1 0 1 0	0A LF 10	1A SUB 26	2A * 42	3A : 58	4A J 74	5A Z 90	6A j 106	7A z 122
1 0 1 1	0B VT 11	1B ESC 27	2B + 43	3B ; 59	4B K 75	5B [91	6B k 107	7B { 123
1 1 0 0	0C FF 12	1C FS 28	2C , 44	3C < 60	4C L 76	5C \ 92	6C l 108	7C | 124
1 1 0 1	0D CR 13	1D GS 29	2D - 45	3D = 61	4D M 77	5D] 93	6D m 109	7D } 125
1 1 1 0	0E SO 14	1E RS 30	2E . 46	3E > 62	4E N 78	5E ^ 94	6E n 110	7E ~ 126
1 1 1 1	0F SI 15	1F US 31	2F / 47	3F ? 63	4F O 79	5F _ 95	6F o 111	7F DEL 127

```cpp
#include <iostream>
using namespace std;

int main()
{
    struct punkt3d {float x; float y; float z;};
    struct dreieck {punkt3d *p1; punkt3d *p2; punkt3d *p3;};
    struct tetraed {dreieck *d1; dreieck *d2; dreieck *d3; dreieck *d4;};

    punkt3d p[100];   // Feld mit max. 100 Punkten
    dreieck d[50];    // Feld mit max. 50 Dreiecken
    tetraed t[10];    // Feld mit max. 10 Tetraedern

    // Punkte einlesen oder bestimmen
    p[0].x=0.0; p[0].y=0.0; p[0].z=0.0;
    p[3].x=6.6; p[3].y=0.0; p[3].z=0.0;
    p[8].x=0.0; p[8].y=8.8; p[8].z=0.0;
    p[6].x=0.0; p[6].y=0.0; p[6].z=4.4;
    // ...

    // Dreiecke einlesen oder definieren
    // Adressen der beschreibenden Punkte zuweisen
    d[2].p1=&p[0]; d[2].p2=&p[8]; d[2].p3=&p[3]; // Grundfläche (z=0)
    d[3].p1=&p[0]; d[3].p2=&p[3]; d[3].p3=&p[6]; // links-hinten (y=0)
    d[4].p1=&p[0]; d[4].p2=&p[6]; d[4].p3=&p[8]; // rechts-hinten (x=0)
    d[7].p1=&p[6]; d[7].p2=&p[3]; d[7].p3=&p[8]; // Schrägfläche
    // ...

    // Tetraeder einlesen oder definieren
    // Adressen der beschreibenden Dreiecke zuweisen
    t[0].d1=&d[2]; t[0].d2=&d[3]; t[0].d3=&d[4]; t[0].d4=&d[7];
    // ...

    cout << "Punkt8(t0.d1): "
        << t[0].d1->p2->x << ' '
        << t[0].d1->p2->y << ' '
        << t[0].d1->p2->z << endl;
    cout << "Punkt8(t0.d3): "
        << t[0].d3->p3->x << ' '
        << t[0].d3->p3->y << ' '
        << t[0].d3->p3->z << endl;
    cout << "Punkt8(t0.d4): "
        << t[0].d4->p3->x << ' '
        << t[0].d4->p3->y << ' '
        << t[0].d4->p3->z << endl;

    p[8].z-=6.25; cout << "Absenkung um 6.25" << endl;

    cout << "Punkt8(t0.d1): "
        << t[0].d1->p2->x << ' '
        << t[0].d1->p2->y << ' '
        << t[0].d1->p2->z << endl;
    cout << "Punkt8(t0.d3): "
        << t[0].d3->p3->x << ' '
        << t[0].d3->p3->y << ' '
        << t[0].d3->p3->z << endl;
    cout << "Punkt8(t0.d4): "
        << t[0].d4->p3->x << ' '
        << t[0].d4->p3->y << ' '
        << t[0].d4->p3->z << endl;

    return 0;
}
```

```
#include <fstream>
#include <string>
using namespace std;

int main() // Auf- und Abbau einer geschlossenen Zeigerkette (Abzählreim)
{
    const int anzahl_silben=11;

    struct t_kind{
        string name;
        t_kind *next;
    };

    t_kind *neues_kind;         neues_kind = NULL;
    t_kind *aktuelles_kind;     aktuelles_kind = NULL;
    t_kind *raus_kind;          raus_kind = NULL;

    ifstream f("kinder.txt");
    ofstream o("ergebnis.txt");

    while (!f.eof()) { neues_kind = new t_kind;
        f >> neues_kind->name;              // Inhalt belegen
        neues_kind->next = aktuelles_kind;  // next-Adresse belegen
        aktuelles_kind = neues_kind;        // aktuelles_kind aktualisieren
    }
    f.close(); // Datei wieder schließen

    while (aktuelles_kind->next != NULL) {
        o << aktuelles_kind->name << " ";   // zur Kontrolle ausgeben ...
        aktuelles_kind = aktuelles_kind->next;  // ... und aktuelles_kind-Zeiger weitersetzen
    }
    aktuelles_kind->next = neues_kind;          // bei der Gelegenheit Ring schließen
    neues_kind = NULL;                          // unnötig, aber zur Sicherheit
    o << "\nAktuelles Kind: " << aktuelles_kind->name << "\n\n";

    // jetzt kommt der Abzählreim
    // der Zeiger "aktuelles kind" läuft für jede Silbe um ein Kind weiter
    o << "Ein Abzaehlreim pro Zeile:\n";
    while (aktuelles_kind!=aktuelles_kind->next) { // solange noch mindestens ein anderes Kind da
        for (int i=1; i<=anzahl_silben-1; i++) {   // eine Silbe weniger, damit Vorgänger noch bekannt
            aktuelles kind = aktuelles kind->next;
            o << aktuelles_kind->name+" ";
        }
        o << aktuelles_kind->next->name << " fliegt raus\n"; // letzte Silbe
        raus kind = aktuelles kind->next;
        aktuelles_kind->next = raus_kind->next; // Ring neu schließen
        delete raus_kind; // Speicherzellen für dieses Kind wieder freigeben
        raus_kind = NULL;
```

```
    }
    o << aktuelles_kind->name << " bleibt zum Schluss uebrig.\n\n";
    delete aktuelles_kind; // Nun ist auch dieser Speicher wieder freigegeben
    aktuelles_kind = NULL;

    return 0;
}
/* Eingabe-Datei "kinder.txt"
Anton Berta Carl Dora Emil Fritz
*/
/* Ausgabe-Datei "ergebnis.txt"
Fritz Emil Dora Carl Berta
Aktuelles Kind: Anton

Ein Abzaehlreim pro Zeile:
Fritz Emil Dora Carl Berta Anton Fritz Emil Dora Carl Berta fliegt raus
Anton Fritz Emil Dora Carl Anton Fritz Emil Dora Carl Anton fliegt raus
Fritz Emil Dora Carl Fritz Emil Dora Carl Fritz Emil Dora fliegt raus
Carl Fritz Emil Carl Fritz Emil Carl Fritz Emil Carl Fritz fliegt raus
Emil Carl Emil Carl Emil Carl Emil Carl Emil Carl Emil fliegt raus
Carl bleibt zum Schluss uebrig.
*/
```