Datenanalyse in der Physik

Vorlesung 1

Prof. Dr. J. Mnich

DESY und Universität Hamburg

Universität Hamburg

Organisation der Vorlesung und Übung

- Termin:
  3.8. - 14.8. 2015, DESY Geb, 1b 02.230 (Seminarraum 3)
- Vorlesung: Täglich 9:00 - 10:30
- Übung: Täglich 10:30 - 13:00
- Beteiligte an Vorlesung und Übungen

Joachim Mnich	Joachim.Mnich@desy.de
Abigail O'Rourke	abigail.orourke@desy.de
Daniel Rauch	daniel.rauch@desy.de
James Robinson	james.edward.robinson@desy.de
Mehmet Özgur Sahin	Ozgur.Sahin@desy.de
Anne Schütz	anne.schuetz@desy.de
Simon Spannagel	simon.spannagel@desy.de

- Für die Vorlesung ist eine Webseite eingerichtet:

  `https://particle-physics.desy.de/e242206`

  - Inhalt der Vorlesung:
    Folien, Links zu Referenz-Handbüchern, . . .
  - Übungen

Universität Hamburg

Inhalt und Zeitplan

03.08.	Einführungen in C und MAPLE
04.08.	Wahrscheinlichkeiten in der Physik
05.08.	Wahrscheinlichkeitsverteilungen: Binomial, Poisson, Gauß, χ^2
06.08.	Zentraler Grenzwertsatz und mehrdimensionale Wahrscheinlichkeitsdichteverteilungen
07.08.	Mehrdimensionale Gauß-Verteilung und Transformation von Wahrscheinlichkeitsdichten
10.08.	Maximum-Likelihood-Methode
11.08.	Das Minimierungsprogramm MINUIT und Konfidenzgrenzen
12.08.	Methode der kleinsten Quadrate (χ^2-Methode)
13.08.	Erzeugung von Pseudo-Zufallszahlen und die Monte Carlo Methode
14.08.	Numerische Integrationsmethoden und Lösung von Differentialgleichungen

Die Inhalte und Termine sind als ungefähre Anhaltspunkte zu verstehen und können sich nach Bedarf ändern.

Universität Hamburg

Computer als Werkzeug für Physiker

Physiker setzen Computer hauptsächlich zur Lösung komplexer numerischer Probleme ein

Die Beschreibung des Problems für den Rechner sollte möglichst nahe an der menschlichen Denk- und Ausdrucksweise sein (d.h. nicht in Form von Bits und Bytes des Prozessorcodes)

Einsatz von

- Programmiersprachen zur Formulierung des Problems
  weitverbreitet sind FORTRAN (= FORmula TRANslator) und C
- Programmen zur Computeralgebra wie z.B. Mathematica oder Maple
  Höhere Form der Abstraktion als Programmiersprache

Einsatzgebiete der beiden Werkzeuge überlappen zum Teil (z.B. Numerik) hauptsächlich ergänzen sie sich aber sinnvoll

Universität Hamburg

Programmiersprache C

Wir behandeln hier die Programmiersprache C:

- C-Programme lassen sich sehr kompakt schreiben
- Unix/Linux sind in C geschrieben
- Objektorientierte Programmiersprachen wie C++ und Java bauen auf C auf

C hat aber auch Nachteile:

- C-Programme lassen sich sehr unverständlich schreiben
- C erlaubt viele Fehler (Speicherüberschreibung)

Online Dokumentation zur Programmiersprache C:

`http://www.strath.ac.uk/IT/Docs/Ccourse`

Universität Hamburg

Programmieren in C

Erstellung eines ausführbaren Programmes besteht aus 2 Schritten

1. Erzeugung des Quell-Codes
   Die Aufgabe des Programmieres ist es mit einem Editor eine Textdatei zu erstellen, die das gestellte Problem mit den Befehlen und der Syntax der Programmiersprache C beschreibt
2. Erzeugung des Maschinen-Codes
   Das ist Aufgabe eines Programmes, das „Compiler“ heisst
   Der Compiler
   (a) übersetzt die Anweisungen im Quell-Code in Maschinen-Code (compilieren)
   (b) fügt bereits übersetzte Programme aus Bibliotheken dazu (linken)
   (c) und erzeugt eine ausführbare, binäre Datei

Diese Binär-Datei kann dann in Linux durch ihren Namen aufgerufen und ausgeführt werden

Universität Hamburg

Beispiel zur Erstellung eines C-Programmes

- C-Quelldatei `sinus.c`

```
/* Berechnung des Sinus */
#include <math.h>
#include <stdio.h>
main()
{
  double x=1.3;
  printf("Der Sinus von %f hat den Wert %f \n",x,sin(x));
}
```

(Berechnet den Sinus von 1,3 rad)

- Compiler

Wir benutzen den Gnu-Compiler `gcc`

```
gcc -lm -o sinus sinus.c
```

Optionen:

  - `-lm` lädt die Mathematik-Bibliothek, die ein Programm zur Berechnung des Sinus enthält
  - `-o sinus` gibt an, dass das Programm in der Datei `sinus` abgelegt werden soll

- Ausführung des Programms

```
[flcl01]: ./sinus
Der Sinus von 1.300000 hat den Wert 0.963558
```

Universität Hamburg

Struktur und Syntax von C-Programmen

```
/* Berechnung des Sinus */
#include <math.h>
#include <stdio.h>
main()
{
  double x=1.3;
  printf("Der Sinus von %f hat den Wert %f \n",x,sin(x));
}
```

- Kommentare:

Text zwischen `/*` und `*/` oder Text nach `//` am Zeilenanfang wird vom Compiler ignoriert

Verwenden Sie sinnvolle Kommentare
Sie helfen dem Programmierer eigene und fremde Programme zu verstehen

- Funktionen:

Ein C-Programm besteht aus einer Folge von Funktionen

Die Funktion `main` muss darin enthalten sein, mit ihr beginnt die Programmausführung

Universität Hamburg

Struktur und Syntax von C-Programmen

```
/* Berechnung des Sinus */
#include <math.h>
#include <stdio.h>
main()
{
  double x=1.3;
  printf("Der Sinus von %f hat den Wert %f \n",x,sin(x));
}
```

- Gebrauch von Klammern:
  - `( )` runde Klammern umschließen Argumente von Funktionen `main()` hat hier kein Argument
  - { } geschweifte Klammern fassen Anweisungen zu einem Block, z.B. einer Funktion, zusammen
  - `[ ]` eckige Klammern kennzeichnen Indizes für Felder (siehe unten)

  Klammern treten immer paarweise auf!
- Nach jeder Anweisungen muß ein Semikolon `;` stehen
  Man kann mehrere Anweisungen in eine Zeile schreiben, tun Sie es nicht!
- Leerzeichen und Zeilenumbruch haben keine Bedeutung (außer in Zeichenketten)
  Nutzen Sie diese Freiheit zur Strukturierung ihres Programms!

Universität Hamburg

Variablen

```
/* Berechnung des Sinus */
#include <math.h>
#include <stdio.h>
main()
{
  double x=1.3;
  printf("Der Sinus von %f hat den Wert %f \n",x,sin(x));
}
```

Typischerweise holt ein Programm Daten aus dem Speicher, führt Rechnungen aus und legt Ergebnisse wieder im Speicher ab

- Der Computer kümmert sich darum wo genau im Speicher diese Daten stehen
- Sie als Programmierer müssen
  - den benötigten Speicherplätzen Namen geben und
  - den Typ der Speicherplätze festlegen, d. h. definieren, wie die Bits interpretiert werden müssen (ASCII-Zeichen, ganze oder reelle Zahl, . . .)

Dazu definieren sie Variablen mit Deklarationen am Anfang eines Anweisungsblocks

Universität Hamburg

Variablentypen

Typ	Bytes	Wertebereich
`char`	1	$-128 \ldots +127$ oder ASCII-Zeichen
`long`	4	$-2\,147\,483\,648 \ldots +2\,147\,483\,647$
`short`	2	$-32\,768 \ldots +32\,767$
`int`	4	wie `long`
`float`	4	$-10^{38} \ldots +10^{38}$
`double`	8	$-10^{308} \ldots +10^{308}$

- `char` kann als Zahl oder ASCII-Zeichen interpretiert werden
  Umrechung Groß- in Kleinbuchstaben durch Addition von 32 (1 im 6. Bit)
- `int`, `long`, `short` existieren auch vorzeichenlos als `unsigned`
  z.B. hat Typ `unsigned short` den Wertebereich $0 \ldots +65\,535$
- Eingabe von ganzen Zahlen entweder als Dezimalzahl (z.B. `139`), Oktalzahl (`0213`) oder Hexadezimalzahl (`0x8B`) möglich
- Fließkommazahlen werden durch Dezimalpunkt und Zehnerpotenz eingegeben, z.B. `3.14159` oder `314.59E-2`
- Logische Werte `wahr` und `falsch` werden durch ganze Zahlen dargestellt:
  `wahr` entpricht einem Wert ungleich Null
  `falsch` entspricht dem Wert Null

Variablendeklaration

- In Variablendeklaration kann Wert zugewiesen werden
  `double x=1.3;`
- Mehrere Variablen können in einer Deklaration definiert werden
  `int anzahl=100, zaehler=1, i;`
- Gültigkeitsbereich der Deklaration:
  - innerhalb eines durch { ... } begrenzten Anweisungsblocks
    Variable ist nur in diesem Block definiert
  - außerhalb eines Anweisungsblocks (Funktion)
    gilt für alle folgenden Blöcke (Funktionen)
  - ähnliches gilt für die Deklaration von Funktionen

Präprozessor und Header-Files

Vor dem Compilieren werden vom C-Präprozessor Befehle abgearbeitet

Präprozessor-Anweisungen starten mit # in der ersten Spalte

Für uns wichtig sind 2 Präprozessor-Befehle

- `#define` weist einem Symbol einen Wert oder Ausdruck zu, der vom Präprozessor im Quellcode für das Symbol eingesetzt wird

  Beispiel: `#define N 1000`
  Alternative ist die C-Anweisung `const int N=1000;`

- `#include` fügt Datei in den Quellcode ein
  - Eigene Dateien werden eingefügt durch
    `#include "file"`
  - Header-Files enthalten Funktionsdefinitionen des C-Standard:
    `#include <file.h>`

    Wichtige Header-Files:

`<math.h>`	Mathematik-Bibliothek
`<stdio.h>`	Ein- und Ausgabe-Funktionen
`<stdlib.h>`	Standardbibliothek (u.a. Zufallszahlen)
`<time.h>`	Zeitfunktionen
`<string.h>`	Manipulationen von Zeichenketten

Arithmetische Operationen

Syntax für Grundrechenarten wie in der Schulmathematik:

Addition	`a+5`
Subtraktion	`a-5`
Multiplikation	`a*5`
Division	`a/5`
Restwert (Modulo)	`a%5`

- Zuweisung des Ergebnis einer Operation durch Gleichheitszeichen $=$
  z.B. `b = a + 5;`, `b = a * 5;` etc.

  Auch `a = a + 5;` ist möglich!
  (Wert der Variablen a wird um 5 erhöht)

- Ausdrücke können zusammengefasst werden (Punkt- vor Strichrechnung)

  `a = 2 + 3 * 4;` a bekommt den Wert 14
  `a = (2 + 3) * 4;` a bekommt den Wert 20

Typumwandlung

- Bei Division ganzer Zahlen entsteht wieder ganze Zahl durch Abschneiden der Nachkommastellen (kein Runden!)
  Beispielsweise ergibt

```
int a, b=8;
a = b/3;
```

  für a den Wert 2

- Um richtiges Ergebnis zu bekommen muss hier mit Fließkommazahlen gerechnet werden
  z.B. durch Typ-Umwandlung

```
double a;
int b=8;
a = (double)b/3.;
```

  wandelt zunächst 8 in 8.0 um, bevor Division ausgeführt wird
  Achtung: Variable `b` bleibt Typ `int`

- Operatoren für Typumwandlung erhält man aus dem Namen des Variablentyps durch Klammern
  `(char)`, `(long)`, `(char)`, ...

Zuweisung

Zuweisung eines Wertes zu einer Variablen geschieht durch Gleichheitszeichen =

C kennt einige Abkürzungen
Folgende Anweisungen sind zeilenweise identisch

```
a = a+b;   a += b;
a = a+1;   a += 1;   a++;
a = a-1;   a -= 1;   a--;
```

- Statt dem + Operator können in der Abkürzung `a += b;` auch andere Operatoren verwendet werden

- Bei den Abürzungen `a++` und `a--` wird zunächst der Wert von `a` verwendet und zurückgegeben und danach wird `a` um 1 geändert
  So werden durch

```
int a=5, b, c;
b = 2*a++;
c = 2*a;
```

  den Variablen `b` und `c` die Werte 10 bzw. 12 zugewiesen

  Wird hauptsächlich für Zähler in Scheifen verwendet

Mathematische Funktionen

Einige wichtige Funktionen, die in der Mathematik-Bibliothek definiert sind:

- Typ-Deklarationen in `<math.h>`
- Binär-File wird durch Option `gcc -lm` dazu gelinkt

`sin(x)`	Trigonometrische	`sinh(x)`	Hyperbolische
`cos(x)`	Funktionen	`cosh(x)`	Funktionen
`tan(x)`	alle Winkel im	`tanh(x)`	
`asin(x)`	Bogenmaß	`asinh(x)`	
`acos(x)`		`acosh(x)`	
`atan(x)`		`atanh(x)`	
`atan2(y,x)`	$\mathrm{atan}(x/y)$		
`exp(x)`	e^x	`sqrt(x)`	$\sqrt{x}$
`log(x)`	$\log_e x$	`hypot(x,y)`	$\sqrt{x^2+y^2}$
`log10(x)`	$\log_{10} x$	`pow(x,y)`	x^y
`erf(x)`	Fehlerfunktion $2/\sqrt{\pi}\int_0^x e^{-z^2}\mathrm{d}z$		
`erfc(x)`	1 – `erf(x)`		
`lgamma(x)`	Logarithmus der Gamma-Fkt. $\log\left(\int_0^\infty t^{x-1}e^{-t}\mathrm{d}t\right)$		
`lgamma(n+1)`	Logarithmus von n-Fakultät $\log n!$		

Funktionen und Argumente sind alle vom Typ `double`

Universität Hamburg

Mathematische Konstanten

Im Header-File `<math.h>` sind einige häufig gebrauchte mathematische Konstanten definiert (Typ `double`)

```
# define M_E            2.7182818284590452354   /* e */
# define M_LOG2E        1.4426950408889634074   /* log_2 e */
# define M_LOG10E       0.43429448190325182765  /* log_10 e */
# define M_LN2          0.69314718055994530942  /* log_e 2 */
# define M_LN10         2.30258509299404568402  /* log_e 10 */
# define M_PI           3.14159265358979323846  /* pi */
# define M_PI_2         1.57079632679489661923  /* pi/2 */
# define M_PI_4         0.78539816339744830962  /* pi/4 */
# define M_1_PI         0.31830988618379067154  /* 1/pi */
# define M_2_PI         0.63661977236758134308  /* 2/pi */
# define M_2_SQRTPI     1.12837916709551257390  /* 2/sqrt(pi) */
# define M_SQRT2        1.41421356237309504880  /* sqrt(2) */
# define M_SQRT1_2      0.70710678118654752440  /* 1/sqrt(2) */
```

Universität Hamburg

Pseudo-Zufallszahlen

Viele Anwendungen auf dem Computer benötigen Zufallszahlen

Monte-Carlo-Methode

Obwohl ein Computer deterministisch ist, gibt es mathematische Algorithmen, die Zahlen erzeugen, die in statistischen Tests zufällig verteilt erscheinen
Pseudo-Zufallszahlen-Generator $x_{n+1} = f(x_n)$

Generatoren für solche Zahlen in C sind (`stdlib.h`):

- `random()`
  liefert ganze Zahlen zwischen 0 und `RAND_MAX` $= 2\,147\,483\,647$
- `drand48()`
  liefert liefert Fließkommazahlen (`double`) im Intervall [0,1]

`drand48` ist vorzuziehen, weil der Algorithmus eine größere Periode hat

Problem: Mehrfacher Programmaufruf erzeugt immer die gleiche Zahlensequenz

Ausweg: Setze Startwert (Seed) z.B. durch Zeitfunktion
`time(0)`, definiert in Header-File `<time.h>`, gibt Zahl der Sekunden zurück, die seit einem bestimmten Zeitpunkt verstrichen sind

Setzen der Seed durch `srand48(time(0));` bzw. `srandom(time(0));`


Universität Hamburg

Logische Operationen

Logische Operationen haben als Resultat den Wert `wahr` oder `falsch`

- Operatoren zum Vergleich zweier Zahlen

gleich	`a == b`	ungleich	`a != b`
kleiner als	`a < b`	größer als	`a > b`
kleiner oder gleich	`a <= b`	größer oder gleich	`a >= b`

Im Beispiel wird der Variable c der Wert 0, d.h. `falsch` zugewiesen:

```
int a=5, b=3, c;
c = a==b;
```

- Operatoren zum Verknüpfen logischer Argumente

UND	`a && b`	ODER	`a || b`	NICHT	`!a`

Im Beispiel wird der Variable c ein Wert ungleich 0, d.h. `wahr` zugewiesen:

```
int a=5, b=3, c;
c = a&&b
```

- Reihenfolge bei logischen Operationen: Vergleich, NICHT, UND, ODER
  Setzen Sie im Zweifelsfall Klammern `( ... )`!


Universität Hamburg

Bedingte Anweisungen I

Mit Hilfe der `if`- und `switch`-Anweisungen können Anweisungsblöcke abhängig von Ergebnissen ausgeführt werden

- `if`-Anweisung

  `if (`*`test`*`) {`*`Block1`*`} else {`*`Block2`*`}`

  Falls der logische Ausdruck *`test`* den Wert `wahr` hat, werden die Anweisungen in *`Block1`* ausgeführt, sonst die in *`Block2`*

  Einfaches Beispiel:

```
if(x<0) { x = -x;
          y = sqrt(x); }
else      y = sqrt(x);
```

  Falls Block nur aus einer Anweisung besteht dürfen Klammern weggelassen werden

Schleifen I

Häufig müssen Programmteile sehr oft ausgeführt werden

$\Rightarrow$ Programmschleifen durch `while`-, `do`- und `for`-Befehle

- `while`-Schleifen

  `while (`*`test`*`) {`*`Block`*`}`

  Der Anweisungsblock wird ausgeführt solange *`test`* den Wert `wahr` ergibt

```
int n=100, nquadrat;
while (n--) nquadrat=n*n;
```

  Berechnet Quadratzahlen von 99*99 bis 0

- `do`-Schleifen

  `do {Block} while (`*`test`*`);`

  Wie `while`, aber der Anweisungsblock wird mindestens einmal ausgeführt

Schleifen II

- `for`-Schleifen

  `for (`*`start;test;inkrement`*`) {`*`Block`*`}`

  1. Führe *`start`*-Anweisung aus
  2. Ist *`test`* `wahr`?
  3. Wenn ja, dann führe Anweisungs-*`Block`* aus
  4. Führe *`inkrement`*-Anweisung aus
  5. Gehe zu b), solange bis *`test`* `falsch` ist

```
int n, nquadrat;
for( n=1; n<=100; n++ )  nquadrat = n*n;
```

Berechnet Quadratzahlen von 1 bis 100

- Abbruch von Schleifen durch `continue` und `break`
  - `continue` überspringe Rest des Anweisungsblocks und fahre mit *`inkrement`*-Anweisung fort (bzw. Schleifenanfang)
  - `break` bricht die Schleife sofort ab und fährt mit 1. Anweisung nach Block fort

Monitorausgabe

Formatierte Ausgabe von Zahlen und Text mit der Funktion

`printf()`

- Textausgabe:
  Einschluss der Zeichenkette in Anführungsstriche

```
printf("Dieser Text erscheint auf dem Monitor!");
```

- Zahlenausgabe:
  Formatzeichen, die mit `%` starten, und Anfügen der Variable als Argument

```
printf("Die Erdbeschleunigung ist %lf kg m/s2 \n",g);
```

- Im gleichen `printf()`-Befehl können mehrere Variablen ausgegeben werden
  Für jede Variable muss ein geeignetes Formatzeichen in der Zeichenkette enthalten sein

```
printf(" Der ggT von %i und %i ist %i \n",a,b,ggt(a,b));
```

Formatzeichen

Wichtige Formatzeichen	
`%c`	ASCII-Zeichen vom Typ `char`
`%s`	Zeichenketten Typ `char[]`
`%i`	Ganzzahl (Integer) dezimal `int, short, long`
`%x`	Integer hexadezimal (vorzeichenlos)
`%o`	Integer oktal (vorzeichenlos)
`%f`	Fließkommazahl `float` ohne Exponent
`%e`	Fließkommazahl `float` mit Exponent
`%g`	Fließkommazahl `float` mit oder Exponent
`%lf,%le, %lg`	Fließkommazahl `double`

- Erweiterung der Formatzeichen durch

Zahl	zur Angabe der Größe des Ausgabefeldes
`-`*Zahl*	für linksbündige Ausgabe
`+`	zur Ausgabe auch positiver Vorzeichen
`0`*Zahl*	führende Nullen anstatt Leerzeichen

Universität Hamburg

Formatzeichen

- Beispiel für Integer:
  Folgende Tabellenausgabe wird erzeugt durch

```
printf(" %3i   %-3i %+4i  %03i \n",i*i,i*i,i*i,i*i);
```

```
  1   1     +1  001
  4   4     +4  004
  9   9     +9  009
 16   16   +16  016
 25   25   +25  025
 36   36   +36  036
 49   49   +49  049
 64   64   +64  064
 81   81   +81  081
100   100 +100  100
```

- Fließkommazahlen:
  Gesamtgröße des Ausgabefeldes und Zahl der Nachkommastellen werden durch Punkt getrennt angeben
  Beispiel:

```
printf(" %6.3lg   %12.5le \n",M_PI,M_PI);
   3.14    3.14159e+00
```

Universität Hamburg

Steuerzeichen

- Zur Formatierung und zum Ausgeben von Sonderzeichen dienen Steuerzeichen in der Zeichenkette, die mit \ anfangen

Wichtige Steuerzeichen	
\n	NL Zeilenvorschub
\a	BEL akustisches Signal
\%	Ausgabe von %
\?	Ausgabe von ?
\\	Ausgabe von \
\¨	Ausgabe von ¨
\’	Ausgabe von ’

Normalerweise ist letztes Zeichen in Zeichenkette der Zeilenvorschub \n

Zeiger I

Ein wichtiges Element von C sind Zeiger (engl. Pointer)
Zeiger enthalten Adresse eines Speicherplatzes

- Deklaration einer Zeigervariablen durch einen Stern `*`
  Beispiel: `int *z;` deklariert eine Zeigervariable, die die Adresse einer Variablen von Typ `int` enthält
- Zuweisung einer Adresse geschieht durch Operator `&`
  Beispiel: Nach
  ```
  int a, *z;
  z = &a;
  ```
  enthält der Zeiger `z` die Adresse der Variablen `a`
- Wert der Variablen, deren Adresse im Zeiger steht, kann durch Operator `*` ermittelt werden
  Im obigen Beispiel kann der Wert der Variable `a` auch durch den Ausdruck `*z` ausgedrückt werden

Zeiger werden hauptsächlich in Funktionsargumenten gebraucht

Mit Zeigern können arithmetische und logische Operationen ausgeführt werden

Zeiger II

Zusammenfassung	
`int a;`	Deklaration einer Variablen vom Typ `int`
`int *z;`	Deklaration eines Zeiger auf eine Variable vom Typ `int`
`z = &a;`	Zeiger `z` enthält nun Adresse der Variablen `a`
`&a`	Adresse der Variablen `a`
`z`	Adresse der Variablen `a`
`a`	Wert der Variablen `a`
`*z`	Wert der Variablen `a`

Felder

Bisher: Deklaration einzelner Speicherplätze (Variablen)

Jetzt: Reservierung eines ganzen Speicherbereiches durch Deklaration eines Feldes (engl. Array)

- Deklaration eines eindimensionalen Feldes, d.h. Vektor:
  wie bei Variablen plus Angabe der Länge des Feldes in eckigen Klammern

  Typ Name [*Anzahl*]

  Beispiele:
  - `long a[100];` reserviert 100 sequentielle Speicherplätze für ganze Zahlen zu je 4 Byte
  - `double b[10];` 10 Plätze für 8 Byte Fließkommazahlen etc.

Achtung: Indices laufen von 0 bis n-1

- Referenz auf Feldelement durch Angabe des Index: `a[0], a[1], ...`

  Beispiel:
  `a[5] = 10;` weist dem 6. Element des Feldes `a` den Wert 10 zu

Felder: Beipiel I

```
/* Einfaches Beispiel zur Verwendung von Feldern */
#include <math.h>                /* Definition Mathematik-Bibliothek   */
#include <stdio.h>               /* Standard Input-Output-Funktionen   */
#define ISIZE 100             /* Definiere Groesse eines Integer-Feldes */
#define RSIZE 10              /* und eines Feldes von Fliesskommazahlen */
main()
{
 int i;             /* Schleifenindex                                    */
 long   a[ISIZE];   /* Deklariere Feld mit 8-Byte Ganzzahlen             */
 double b[RSIZE];   /* Deklariere Feld mit 10 8-Byte Fliesskommazahlen */

 /* Speichere alle Quadratzahlen von 1 bis ISIZE im Feld a[]           */
 for (i=1; i<=ISIZE; i++) a[i-1] = i*i;
 /* Speichere die Logarithmen der Zahlen 1.0, ..., RSIZE in b[]        */
 for (i=1; i<=RSIZE; i++) b[i-1] = log ((double) i);

 /* Ausgabe der berechneten und gespeicherten Ergebnisse               */
 for (i=1; i<=ISIZE; i++) printf("%3i * %-3i = %5i \n",i,i,a[i-1]);
 printf("\n");                                    /* Erzeugt Leerzeile */
 for (i=1; i<=RSIZE; i++)
                  printf("log %-4.1f  = %lf \n",(double) i,b[i-1]);
}
```


Initialisierung

- Initialisierung von Feldern:
  Wie bei Variablen können den Feldelementen Werte zugewiesen werden:
  Beispiele:
  - `int a[6] = {1, -3, 3, 5, 9, 12}`
  - `int prime[] = {1,2,3,5,7}` definiert ein Feld `int prime[5]` der ersten 5 Primzahlen

    Der Compiler kann die benötigte Feldgröße selbst berechnen
- Achtung:
  Der Gnu-Compiler initialisiert Variablen und Felder nicht automatisch zu Null!

  Variablen und Feldelemente, die als Zähler benutzt werden, müssen
  - mit Nullen initialisiert werden
  - oder im Programm zu Null gesetzt werden

Mehrdimensionale Felder

C erlaubt Verwendung von Feldern mit beliebig vielen Dimensionen (Indices)

- Deklaration eines zweidimensionalen Feldes, d.h. Matrix:
  Angabe der Zeilen und Spalten

  Typ Name`[`*Zeilen*`][`*Spalten*`]`

  Beispiel: Die Matrix

  $$\begin{pmatrix} 8 & 2 & 0 & 0 \\ -4 & 3 & -2 & 0 \\ 1 & 1 & 0 & -1 \end{pmatrix}$$

  kann deklariert werden durch

  ```
  int a[3][4] = { {8,2,0,0}, {-4,3,-2,0}, {1,1,0,-1} };
  ```

- Auch bei Matrizen laufen die Indizes von 0 bis n-1

Felder und Zeiger

Die Deklaration eines Feldes reserviert Speicherplatz UND definiert Zeiger, der Adresse des ersten Feldelements enthält

- Eindimensionales Feld:
  Name des Feldes ist Zeiger auf Startadresse des Feldes

  Die Deklaration `double v[12];` definiert den Zeiger `v`, der die Adresse des Speicherplatzes `v[0]` enthält

  $\Longrightarrow$ `v` ist identisch zu `&v[0]`

- Zweidimensionales Feld:
  Name des Feldes ist Zeiger auf Startadresse eines Feldes, das die Startadressen der Zeilen enthält:

  Die Deklaration `int a[3][4]` definiert auch ein Feld `a[3]`, das die Adresse folgender Elemente enthält:
  `a[0] = &a[0][0]`, `a[1] = &a[1][0]`, `a[2] = &a[2][0]`
  Name des Feldes `a` ist ein Zeiger auf den Speicherplatz des Zeigers `a[0]`, also `a = &a[0]`

  $\Longrightarrow$ a ist Zeiger auf Zeiger

  Wird beliebig kompliziert für höher-dimensionale Felder . . .

- Achtung: Die falsche Verwendung (als Wert) eines Feldnamens wird vom Compiler nicht als Fehler erkannt

Zeichenketten I

Zeichenkette (engl. String) ist besondere Form eines eindimensionalen Feldes
Folge von ASCII-Zeichen (Variablen-Typ `char`), die mit `\0` abgeschlossen werden

- Deklaration einer Zeichenkette durch `char[`*Länge*`]`
  Initialisierung ist möglich, z.B.

```
char text[] = "Uni Hamburg"
```

- Letztes Element ist immer das ASCII-Zeichen `\0`
  (ASCII-Zeichen mit Integer-Wert 0)
  Wird automatisch vom Compiler eingefügt

  Obige Zeichenkette enthält also folgende 12 (!!!) ASCII-Zeichen:

```
 'U' 'n' 'i' ' ' 'H' 'a' 'm' 'b' 'u' 'r' 'g' '\0'
  0   1   2   3   4   5   6   7   8   9  10   11
```

- Lange Zeichenketten können durch " unterteilt und auf mehrere Zeilen verteilt geschrieben werden:

```
char dogma[] = "Der Computer ist zum unverzichtbaren Werkzeug "
               "in den Natur- und Ingenieurwissenschaften geworden.";
```


Zeichenketten II

- Zuweisung im Programm über Zeiger:

```
char *text;  /* text ist ein Zeiger auf ein ASCII-Zeichen */
text = "Uni Hamburg";
```

  Mit dem Operator = wird nur die Adresse der Zeichenkette zugewiesen, nicht deren Inhalt

  Folgende Konstruktion geht deshalb NICHT:

```
char text[12];        /* Zeichenkette aus 12 Bytes */
text =  "Uni Hamburg";
```

- Mögliche Alternativen
  - Zuweisung der einzelnen ASCII-Zeichen:

```
char text[11];
text[0] = 'R'; text[1] = 'W'; ... ; text[11] = '\0';
```

  - Kopieren von Zeichenketten mit Funktionen aus `<string.h>`

```
char text[] = "Uni Hamburg";
char zeile[80];
strcpy(zeile,text);
```

Strukturen I

Erweiterung der Feldkonstruktion

Feld: Bereich von Speicherplätze für Variablen gleicher Art und Größe

Struktur: Definition eines Datentyps (Speicherbereichs), der aus elementaren Variablen unterschiedlichen Typs bestehen kann

Anwendungsbeispiel ist Datenbank:
Durch Deklaration einer Struktur können Name, Geburtsdatum, Matrikel-Nr. etc. in einer einzigen Variablen enthalten sein

- Deklaration einer Struktur
  `struct` *`Typname`* {*`Deklarationen`*};

Beispiel aus der Physik: Für die kinetische Beschreibung eines Gasmoleküls braucht man die Art des Moleküls, Masse, Orts- und Impulskoordinaten:

$\Longrightarrow$ Definiere Struktur mit Typnamen `molekuel`

```
struct molekuel {
  char *sorte;              /* Zeiger auf Name des Molekuels   */
  int molgew;               /* Mol-Gewicht des Molekuels       */
  double x[3];              /* Ortskoordinaten                 */
  double p[3];              /* Impulskoordinaten               */
};
```

Universität Hamburg

Strukturen II

- Struktur definiert neuen Datentyp (Variablentyp)
  Deklaration einer Variablen diesen Typs durch
  `struct` *`Typname Variablenname`*;

  Beispiel:

  ```
  struct molekuel teilchen;
  ```

  deklariert die Variable `teilchen` vom Typ `molekuel`

  Bemerkung: Deklaration der Struktur und der Variablen können auch zusammengefasst werden
  `struct` *`Typname`* {*`Deklarationen`*} *`Variablenname`*;

- Zuweisungen
  Bezeichnung der einzelnen Komponenten geschieht durch
  `Variablenname.Komponente`

  Beispiel:

  ```
  teilchen.sorte  = "Sauerstoff";
  teilchen.molgew = 16;
  teilchen.x[0]   = 3.142;
  teilchen.x[1]   = 2.718;
  ...
  ```

Universität Hamburg

Strukturen III

- Felder von Struktur-Variablen:
  Struktur definiert neuen Datentyp (Variablentyp), der auch dazu benutzt werden kann, Felder dieses Typs zu deklarieren

  Beispiel: Simulation eines Gases mit vielen Molekülen

  ```
  struct molekuel t[1000]
  ```

  deklariert ein Feld `t[1000]` von Variablen der oben definierten Struktur `molekuel`

  Zugriff erfolgt wie bei normalen Feldern durch Angabe des Index, z.B. auf das `i+1`-Molekül

`t[i].sorte`	Zeiger auf Namen des Moleküls
`t[i].molgew`	enthält Molgewicht
`t[i].x[0]`	1. Ortskoordinate
`t[i].x[1]`	2. Ortskoordinate
`...`	usw.

- Strukturen können verschachtelt sein
  d.h. Strukturen können Strukturen als Komponenten haben

Zeiger auf Strukturen

- Analog zu „normalen“ Variablen können auch auf Zeiger auf Struktur-Variablen definiert werden

  Dem Zeiger wird die Adresse der Struktur-Variablen zugewiesen
  Beispiel:

  - `struct molekuel *teilchenptr` definiert einen Zeiger `teilchenptr` auf eine Variable vom Typ `molekuel` und
  - `teilchenptr = &teilchen` weist diesem Zeiger die Adresse der Variablen `teilchen` zu

- Es können Felder von Zeigern auf Felder von Struktur-Variablen deklariert werden
  Beispiel: `struct molekuel *tptr[1000]`
  Zuweisung: `for(i=0; i<1000; i++) tptr[i] = &t[i];`

- Zugriff auf Wert einer Variablen, auf die ein Zeiger zeigt, durch Operator `->`

  `Zeiger->Komponente`

  (Analog zum Operator `*` bei Zeigern auf Variable)

  Beispiel:

  ```
  teilchenptr->sorte  = "Sauerstoff";
  tptr[i]->x[0]   = 3.142;
  ...
  ```

Zeiger auf Strukturen

Gegenüberstellung:

Zeiger auf Variable $\longleftrightarrow$ Zeiger auf Struktur

	Variable	Struktur
Deklaration (Reservierung Speicherplatz)	`int a;`	`struct ALPHA { int a; double b; char c;} A;`
Deklaration Zeiger	`int *z;`	`struct ALPHA *Z;`
Zuweisung der Adresse	`z = &a;`	`Z = &A;`
Wert des Wertes *oder*	`a = 5;` `*z = 5;`	`A.a = 5; A.b = 3.14; A.c = 'Y';` `Z->a = 5; Z->b = 3.14; Z->c = 'Y';`

- Achtung:
  Der Speicherplatz für die Struktur `A` wird durch deren Deklaration reserviert
  Die Deklaration des Zeigers `Z` reserviert nur den Platz zur Speicherung einer Adresse

Funktionen I

Hohe Kunst des Programmierens:

Zerlege Problem in kleine, überschaubare Einzelprobleme

$\Longrightarrow$ Funktionen

- Funktionen enthalten Arbeits- oder Rechenvorschriften, die mit verschiedenen Argumenten abgearbeitet werden können
  Im allgemeinen liefert die Funktion einen Wert zurück
- C-Programme bestehen aus einzelnen Funktionen
  Jedes C-Programm enthält mindestens die Funktion `main`, die beim Programmstart aufgerufen wird
  `main` ruft Funktionen auf, die wiederum Funktionen aufrufen usw.
- Viele Funktionen gehören zu den Standard C-Bibliotheken
  `printf`, `drand48`, ...

  Jetzt: Programmierung eigener Funktionen

Funktionen II

- Deklaration einer Funktion

  Typ Name (Typen und Namen der Variablen) {*Anweisungen*}

  - Funktionstyp gibt Art des zurückgelieferten Wertes an
    (`int`, `double`, `char`, ...)
    Funktionen ohne Rückgabewert werden als Typ `void` deklariert
  - Funktion muss mindestens einmal die Anweisung
    `return` *Rückgabewert*`;` enthalten
  - Die Funktion `main` ist vom Typ `int` und gibt Integer-Wert zurück
    Eigentlich hätten wir immer `int main()` schreiben und die
    Anweisung `return` *Wert*`;` einfügen müssen ...

    Rückgabewert von `main` kann nach Programm-Ende durch `echo $?` abgefragt werden
    In `<stdlib.h>` sind Codes für Programm-Erfolg vereinbart

```
#define EXIT_FAILURE 1 /* Failing exit status.  */
#define EXIT_SUCCESS 0 /* Successful exit status.  */
```

- Aufruf der Funktion

  Name (Werte der Variablen)

  Bereits bekannte Beispiele:
  `printf("Hello World");`, `x = drand48();`, ...

Universität Hamburg

Beispiel einer einfachen Funktion

```
#include <stdio.h>
/*                                                              */
/* Funktion zur Berechnung des Betrages einer Fliesskommazahl */
/*                                                              */
double betrag(double x)
{
  if(x<0) x = -x;
  return x;
}
/*                                                              */
/* Programm zum Testen der Funktion betrag                     */
/*                                                              */
main()
{
  double z = -1.414, zabs;
  zabs = betrag(z);
  printf("Der Betrag von %lf ist %lf \n",z,zabs);
}
```

- Rückgabewerte von Funktionen können direkt an andere Funktionen übergeben werden
  zum Beispiel

```
printf("Der Betrag von %lf ist %lf \n",z,betrag(z));
```

Universität Hamburg

Argumente von Funktionen

- In Funktionsdeklaration werden Variablen-Typen und Namen definiert
  Standard Variablentypen, aber auch Strukturen
- Beim Aufruf einer Funktion wird der Wert einer Variablen übergeben
  Variablenwert wird in lokale Variable der Funktion kopiert
  Deshalb kann die Funktion den Inhalt der Variablen nicht ändern
- Ausweg: Übergabe von Zeigern, d.h. Speicheradressen
  Funktion kann so Inhalt der Speicherplätze, d.h. Wert der Variablen, ändern

Beispiel:

```
#include <stdio.h>

/* Funktion zum Vertauschen zweier Variablen     */
void tausche(double* zx, double* zy)
{
  double w;               /* Zwischenspeicher      */

     w = *zx;             /* Inhalt von zx nach w  */
   *zx = *zy;             /* Inhalt von zy nach zx */
   *zy = w;               /* w nach zy             */
}

main()
{
  double x=5., y=10.;

  tausche(&x, &y);        /* Adressen von x und y  */
  printf(" x=%.1f, y=%.1f \n", x, y);
}
```


Funktionen und Felder I

Zeiger erlauben es, Felder als Argumente an Funktionen zu übergeben

- Eindimensionale Felder
  Angabe von Typ, Startadresse und Länge definiert ein Feld

Beispiel:

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

/* Funktion zur Berechnung der Laenge eines n-dim. Vektors       */
double vleng( double *v, int n)
{
  double qsumme = 0;             /* Quadratsumme der Komponenten */
  int i;                         /* Schleifenindex               */

  for (i=0; i<n; i++) qsumme += v[i]*v[i];
  return sqrt(qsumme);
}

/* Programm zum Testen der Funktion vleng                        */
main()
{
  double a[3] = {3.0, 4.0, 0.0};

  printf("Laenge von a ist %lf \n",vleng(a,3));
}
```

Beachte: Symbol `a` in `main` ist ein Zeiger auf eine `double` Zahl,
genau wie `v` in der Deklaration von `vleng`

Funktionen und Felder II

- Zweidimensionales Felder
  i.a müssen Zahl der Zeilen und Spalten in Funktion bekannt sein

Beispiel Multiplikation von Matrizen fester Größe

```
/*** Matrixmultiplikation ***/
#include <math.h>
#include <stdio.h>

#define Z 3
#define S 4

void matrix_mult( int x[][4], int y[][3], int z[][3])
{
     int i,j,k;
     int sum;

     for( i=0; i<Z; i++)
     for( j=0; j<Z; j++)
      {  sum=0;
         for( k=0; k<S; k++) sum+= x[i][k]*y[k][j];
         z[i][j] = sum;
      }
}
main()
{
  int a[3][4]={ {1,2,3,4},{5,6,7,8},{9,1,2,3}};
  int b[4][3]={ {1,2,3},{5,6,7},{9,1,2},{1,2,3}};
  int c[3][3];
  int i,j;

  matrix_mult(a,b,c);
}
```


Wir ersparen uns höherdimensionale Felder

Argumente von `main`

Auch der Funktion `main` können Argumente übergeben werden
Dazu muss die Deklaration wie folgt ausssehen:

```
main(int argc, char *argv[])
```

- `argc` ist Anzahl der Zeichenketten inklusive des Programmnamens
  Zeichenketten werden durch Leerzeichen getrennt
- `argv` ist ein Zeiger auf ein Feld von Zeigern, von denen jeder auf eine Zeichenkette zeigt

Das Programm

```
/* Testet die Uebergabe von Parametern an main */
#include <stdio.h>
main(int argc, char *argv[])
{
  int i;
  printf("Diesem Aufruf sind %i Strings uebergeben worden \n",argc);
  for (i=0; i<argc; i++) printf(" String %2i : %s \n",i,argv[i]);
}
```

erzeugt bei diesem Aufruf folgende Ausgabe:

```
mnich 165: ./testargv par1 par2
Diesem Aufruf sind 3 Strings uebergeben worden
 String  0 : testargv
 String  1 : par1
 String  2 : par2
```

Umwandlung von Zeichenketten

Normalerweise ist Übergabe einer Zahl gewünscht

Umwandlung von Zeichenkette in Integer bzw. Fließkommazahl durch die Funktionen `atoi` bzw. `atof` in `<stdlib.h>`

Beispiel: Berechnung von π mit Übergabe der Zahl der Schüsse

```
#include <stdlib.h>
main(int argc, char **argv)
{
 ...
  /* Falls Parameter vorhanden setze Zahl der Schuesse */
  if(argc>1) schuss = atoi(argv[1]);
  else       schuss = 10000;
 ...
}
```

Gültigkeitsbereich von Deklarationen

Variablen und Funktionen werden durch Angabe von Typ und Namen deklariert

Grundsätzlich gilt: Deklarationen sind nur gültig

- im auf sie folgenden Quelltext
- innerhalb des von Klammern { ...} umschlossenen Anweisungsblocks

Daraus folgt:

- Variablen können global definiert werden
  am Anfang des Quelltextes, außerhalb einer Funktion
- oder lokal, z.B. innerhalb einer Funktion oder sogar nur im Anweisungsblock einer `for`-Schleife
  Außerhalb dieses Blocks kann der gleiche Variablenname für einen anderen Speicherplatz verwendet werden
- lokale Variablen müssen am Anfang eines Blocks deklariert werden
- Funktionen müssen vor ihrem ersten Aufruf deklariert werden. Deshalb
  - stehen Header-Files am Anfang
  - ist in unseren Beispielen zuerst die Funktion und dann `main` deklariert

Tipp: Deklarieren Sie Variablen so lokal wie möglich und verwenden Sie globale Variablen nur für Größen, auf die mehrere Funktionen zugreifen

Übergabe von Argumenten an Funktionen

- Beim Aufruf einer Funktion wird der Wert einer Variablen übergeben
- Variablenwert wird in lokale Variable der Funktion kopiert

Programm:

```
void myfunct(int x, int y)
{
...
return;
}
main()
{
int a,b;
a = 5;
b = 9;
                <===
...
}
```

Speicherbelegung:

Adresse	Wert
000 000	?
000 004	?
&a = 0A1 002	a = 5
&b = 0A1 006	b = 9

Übergabe von Argumenten an Funktionen

- Beim Aufruf einer Funktion wird der Wert einer Variablen übergeben
- Variablenwert wird in lokale Variable der Funktion kopiert

Programm:

```
void myfunct(int x, int y)
{
                <===
...
return;
}
main()
{
int a,b;
a = 5;
b = 9;
myfunct(a,b);
...
}
```

Speicherbelegung:

Adresse	Wert
000 000	?
000 004	?
&a = 0A1 002	a = 5
&b = 0A1 006	b = 9
&x = 0A1B08	x = 5
&y = 0A1B0C	y = 9

Übergabe von Argumenten an Funktionen II

- Durch Übergabe von Zeigern kann Funktion auf bestimmte Speicherplätze (Variablen) zugreifen
- Besonders effizient bei Feldern und Strukturen

Programm:

```
void myfunct(int *x, int *y)
{
...
return;
}
main()
{
int a,b;
a = 5;
b = 9;
                <===
...
}
```

Speicherbelegung:

Adresse	Wert
000 000	?
000 004	?
&a = 0A1 002	a = 5
&b = 0A1 006	b = 9

Übergabe von Argumenten an Funktionen II

- Durch Übergabe von Zeigern kann Funktion auf bestimmte Speicherplätze (Variablen) zugreifen
- Besonders effizient bei Feldern und Strukturen

Programm:

```
void myfunct(int *x, int *y)
{
                <===
*x = ...
return;
}
main()
{
int a,b;
a = 5;
b = 9;
myfunc(&a,&b);
...
}
```

Speicherbelegung:

Adresse	Wert
000 000	?
000 004	?
&a = 0A1 002	a = 5
&b = 0A1 006	b = 9
0A1 B08	x = 0A1 002
0A1 B0C	y = 0A1 006

Funktionsprototypen

Deklarationen von Funktionen können auch separat in Prototypen erfolgen:

- Funktionsprototyp deklariert Typ der Funktion und Typen der Argumente

Aufruf der Funktion kann nach Prototyp erfolgen
Die eigentliche Funktion kann dann später im Quelltext stehen

Beispiel:

```
#include <math.h>
#include <stdio.h>
double vleng(double *, int);      /* Prototyp der Funktion vleng */
/* Programm zum Testen der Funktion vleng                        */
main()
{
  double a[3] = {3.0, 4.0, 0.0};
  printf("Laenge von a ist %lf \n",vleng(a,3));
}
/* Funktion zur Berechnung der Laenge eines n-dim. Vektors       */
double vleng( double *v, int n)
{
  double qsumme = 0;              /* Quadratsumme der Komponenten */
  int i;                          /* Schleifenindex               */
  for (i=0; i<n; i++) qsumme += v[i]*v[i];
  return sqrt(qsumme);
}
```


Private Header-Files

Normalerweise werden Funktions-Prototypen in Header-Files am Anfang des Quelltextes eingefügt:

- Header-Files mit Funktionsdefinitionen des C-Standard in Verzeichnis `/usr/include/`

  Einfügen dieser Header-Files durch

  `#include <file.h>`

- Eigene Header-Files werden eingefügt durch

  `#include "file"`

  bzw. durch vollständige Angabe des Pfades, wenn nicht im aktuellen Verzeichnis

- Der GNU-Compiler mahnt von sich aus keine fehlenden Funktionsdeklarationen an

  Warnmeldung kann durch folgende Option angeschaltet werden:

  `-Wimplicit-function-declaration`

  Beispiel:

  ```
  gcc testfunc.c -o testfunc -lm -Wimplicit-function-declaration
  ```

Rekursionen

In C kann sich eine Funktion selbst wieder aufrufen

- Aufrufe werden gespeichert, so lange bis eine `return` Anweisung erreicht und ein Wert berechnet wird
- Wert wird dann sukzessive an vorherige Aufrufe übergeben
- Beispiel zur Illustration: Rekursive Berechnung der Fakultät

```
/* Funktion zur rekursiven Berechnung der Fakultaet */
int rekfak(int x)
{
  if (x<0)  return(0);        /* Fehler, x! nicht definiert  */
  if (x==0) return(1);        /* 0! ist 1, Abbruchbedingung  */
  return ( x * rekfak(x-1));
}
```

(Achtung: funktioniert nur für $x \leq 13$)

Tastatureingabe I

Das Gegenstück zur Monitorausgabe mit `printf` ist Tastatureingabe mit `scanf`

- `scanf` liest eingegebene Zeichenkette bis `return`
- Zeichenkette wird unterteilt
  dabei dient `<Leertaste>` zur Trennung
- Steuerzeichen werden interpretiert und ASCII-Text in Zahlenwerte umgewandelt
  Steuerzeichen sind identisch zu `printf`
- `scanf` erwartet mindestens 2 Argumente:
  1. Zeichenkette mit Steuerzeichen
  2. Zeiger auf eine Variable Achtung: Unterschied zu `printf`
- Rückgabewert ist Anzahl der erfolgreich eingelesenen Variablen

Beispiel:

```
float celsius;
printf("Eingabe in Celsius:     ");
scanf("%f", &celsius);
```

Der eingegebene Zahlenwert wird an die Adresse `&celsius` geschrieben

Tastatureingabe II

- Eingabe von mehreren, durch `<Leertaste>` getrennten, Variablen-Werten

```
int i,z;
double d;
char c,str[100]
z = scanf("%s %c %i %lg",str,&c,&i,&d)
```

Falls `scanf` beim Einlesen auf einen Fehler trifft wird die Funktion abgebrochen und `z` enthält Anzahl der erfolgreich eingelesenen Variablen

- Einzelne ASCII-Zeichen können mit den Funktionen `putchar` und `getchar` ausgegeben bzw. eingelesen werden

Definition in `<stdio.h>`

```
char c1;
printf("ASCII-Zeichen Eingabe: ");
c1 = getchar();
printf("ASCII-Zeichen Ausgabe: ");
putchar(c1);
putchar('\n');                          /* Zeilenvorschub */
```

Auch die Eingabe eines Zeichens mit `getchar` muß mit `<return>` abgeschlossen werden

Universität Hamburg

Lesen & Schreiben von Dateien

Eingabe und Ausgabe eines C-Programms soll nicht nur über Tastatur und Monitor erfolgen sondern auch durch Lesen und Schreiben von Dateien

- Linux

Umlenken von Eingabe und Ausgabe (Input und Output) durch < und >:

Standard: Input von Tastatur gelesen Output auf Bildschirm

`befehl < file`	Input für Befehl aus Datei
`befehl > file`	Output des Befehls in Datei
`befehl >> file`	Hänge Output des Befehls an Datei an

`befehl` ist hier der Name der Binärdatei des C-Progammes

- Lesen und Schreiben von Dateien im C-Programm
  - Mehrere, verschiedene Dateien für Ein- und Ausgabe
  - Ein-/Ausgabe auf Tastatur/Monitor und in Dateien
  - Binärdateien

C liefert eine Reihe von Funktionen zum Bearbeiten von Dateien

- Definitionen im Header-File `<stdio.h>`
- Sie starten alle mit dem Buchstaben `f` für File

Universität Hamburg

Öffnen und Schließen von Dateien

Vor dem Lesen oder Schreiben muss eine Datei zunächst geöffnet und nach der Bearbeitung geschlossen werden

Wir behandeln hier nur Textdateien:

- Öffnen einer Datei durch Funktion `fopen`
  Syntax *`fileptr`* `= fopen(`*`Dateiname`*`,`*`Modus`*`)`
  - *`fileptr`* ist ein Zeiger auf die in `<stdio.h>` definierte Struktur `FILE`
    Rückgabewert ist `0`, d.h. `falsch`, bei nicht erfolgreichem Öffnen der Datei, sonst eindeutige Datei-Identifizierung (erlaubt Öffnen mehrerer Dateien)
  - *`Dateiname`* ist Zeiger auf Zeichenkette mit Dateinamen
  - *`Modus`* ist Zeiger auf Zeichenkette mit Code für den Modus

`"r"`	Nur Lesen (read), Datei muss bereits existieren
`"r+"`	Lesen & Schreiben, Datei muss bereits existieren
`"w"`	Schreiben (write), existierende Datei wird gelöscht
`"w+"`	Lesen & Schreiben, existierende Datei wird gelöscht
`"a"`	Schreiben ans Ende einer Datei (append)

- Schließen der Datei durch Funktion `fclose`
  Syntax `fclose(`*`fileptr`*`)`

Beispiel: Öffnen und Schließen von Dateien

```
/* Liest Werte aus ASCII-File und speichert sie in Feldern          */
#include <stdio.h>
#define MAXLINES 100
main()
{
  FILE *fpr;
  char fname[] = "/user/mnich/uebung/uebung01/tabelle";
  double sqs[MAXLINES],xs[MAXLINES],stat[MAXLINES];
  int lines=0;
  fpr = fopen(fname,"r");    /* Oeffnet Datei zum Lesen               */
  if(!fpr)                   /* Fehlermeldung falls nicht erfolgreich */
    {
      printf("Datei %s konnte nicht geoeffnet werden. \n",fname);
      return;                /* Abbruch                               */
    }

/* Hier kann die Datei gelesen werden   ...                           */
  fclose (fpr);              /* schliesse Datei                       */
}
```

Lesen & Schreiben von Dateien

Wichtige Funktionen aus `<stdio.h>` zum Lesen und Schreiben:

`fputc(c,`*`fileptr`*`)`	Schreibe das ASCII-Zeichen `c`
`c = fgetc(`*`fileptr`*`)`	Lese ein ASCII-Zeichen
`fputs(str,`*`fileptr`*`)`	Schreibe die Zeichenkette `str`
`fprintf(`*`fileptr`*`,...)`	Schreibe Text (wie `printf`)
`fscanf(`*`fileptr`*`,...)`	Lese Text (wie `scanf`)
`feof(`*`fileptr`*`)`	Testet End-of-File Flag von *`fileptr`*

- Die Konstante `EOF` (End-of-File) ist in `<stdio.h>` definiert
  Rückgabewert von `fgetc`, `fscanf` bei Erreichen des Dateiendes bzw. bei Fehlern in `fputc`, `fputs`, `fprintf`
- `feof` gibt einen Wert ungleich 0, d.h. `wahr` zurück, falls bei vorherigem Leseversuch EOF erreicht wurde

Beispiel: Öffnen, Lesen und Schließen von Dateien

```
/* Liest Werte aus ASCII-File und speichert sie in Feldern          */
#include <stdio.h>
#define MAXLINES 100
main()
{
  FILE *fpr;
  char fname[] = "/user/mnich/uebung/uebung01/tabelle";
  double sqs[MAXLINES],xs[MAXLINES],stat[MAXLINES];
  int lines=0;
  fpr = fopen(fname,"r");     /* Oeffnet Datei zum Lesen              */
  if(!fpr)                    /* Fehlermeldung falls nicht erfolgreich */
    {
      printf("Datei %s konnte nicht geoeffnet werden. \n",fname);
      return;                 /* Abbruch                              */
    }
  while(!feof(fpr))           /* solange End-of-File nicht erreicht   */
    {                         /* lese Zeile                           */
     fscanf(fpr,"%lf %lf %lf",&sqs[lines],&xs[lines],&stat[lines]);
     lines++;                 /* zaehle eingelesene Zeilen            */
     if(lines >= MAXLINES)    /* ueberpruefe Feldgroesse              */
       {
         printf("Zeilenzahl ueberschreitet Feldgroesse %i\n",MAXLINES);
         break;               /* Verlasse while-Schleife              */
       }
    }
  lines--;                    /* korrigiere Zeilenzahl                */
  fclose (fpr);               /* schliesse Datei                      */
  ...
}
```

Programmierstil

Achten Sie beim Erstellen von Programmen auf Verständlichkeit und Übersichtlichkeit

Denken Sie daran,

- dass Sie Fehler möglichst vermeiden bzw. schnell finden wollen
- dass Sie und andere die Funktion des Programmes möglichst schnell verstehen sollen

Einige Grundregeln:

- Strukturierung: Zusammengehörige Anweisungen (Block) sollten auf einen Blick erkennbar sein
- Vergeben Sie intuitive Namen für Variablen und Funktionen
- Fügen Sie sinnvolle Kommentare ein
- Unterteilen Sie komplexe Programme in kleine, überschaubare Einheiten (Funktionen)
- Fehlersuche:
  - Testen Sie die einzelnen Funktionen ausführlich (illegale Parameter, Division durch Null, . . .)
  - Gehen Sie den Programmablauf Schritt für Schritt durch
  - Geben Sie den Inhalt von Variablen aus