Numerische Sotwareentwicklung in C und C++

Stephanie Friedhoff

Universität zu Köln

Wintersemester 2016/17

1 Organisatorisches

1.1 Einordnung der Vorlesung

Voraussetzungen

- Grundlegende Kenntnisse der Numerischen Mathematik (Numerik I und II)
- Gute Kenntnisse des Programmierens (z. B. Matlab)

Ziele

- Einstieg in grundlegende Konzepte der Programmiersprachen C und C++
- Praktische Grundlagen zum Entwurf und zur Implementierung von wissenschaftlicher Software für numerische Simulationen

1.2 Anrechenbarkeit im Master (Wi-) Mathematik

Master Mathematik

- Basismodul Angewandte Mathematik I, II
- Aufbaumodul Mathematik I, II

Master Wirtschaftsmathematik

- Basismodul Angewandte Mathematik I, II
- Basismodul Informatik
- Basismodul Wirtschaftsinformatik
- Aufbaumodul Mathematik I, II

1.3 Ablauf der Vorlesungen und Übungen

Termine

Vorlesungen: Mo. 10:00 - 11:30 Uhr
Mi. 12:00 - 13:30 Uhr
im Seminarraum 1 des Math. Instituts (Raum 005)

Übungen: Mi. 16:00 - 17:30 Uhr
im Übungsraum 2 des MIs (Gyrhofstraße 8)

Beginn der Übungen: Mi. 19.10.2016

Übungsblätter

Ausgabe: auf der Homepage zur Vorlesung

Abgabe: per E-Mail, weitere Hinweise auf der Homepage und in den Übungen

Weitere Informationen auf der Homepage

`http://www.numerik.uni-koeln.de/`

1.4 Arbeitsumgebung: C (gcc) und C++ Compiler (g++)

Mac OS X: Aus der Anwendung XCode (zu finden z. B. im App Store) Command Line Tools installieren

Linux: (Ubuntu) Geben Sie folgenden Befehl in einem Konsolenfenster ein (hier für gcc):
`sudo apt-get install gcc`

(openSUSE) Folgen Sie z. B. folgender Anleitung:
`https://lizards.opensuse.org/2013/10/07/opensuse-and-gcc-part1/`

Hinweis: Eine Linux-Distribution kann auch auf einem USB-Stick installiert werden. Siehe z. B.
`http://www.lidux.de/anleitungen/42-opensuse-usb-stick-installieren-persistent`

Windows: MinGW (`http://www.mingw.org`) oder Cygwin (`http://cygwin.com`)

1.5 Literatur

- Rouben Rostamian.
  Programming Projects in C for Students of Engineering, Science, and Mathematics.
  SIAM, 2014.
- Joe Pitt-Francis, Jonathan Whiteley.
  Guide to Scientific Computing in C++.
  Springer, 2012.

2 Motivation und Vorbemerkungen

2.1 Numerische Simulation

Als **numerische Simulation** bezeichnet man allgemein Computersimulationen, welche mittels numerischer Methoden wie zum Beispiel mit Turbulenzmodellen durchgeführt werden.

(Quelle: https://de.wikipedia.org/wiki/Numerische_Simulation)

Ziel der numerischen Simulation ist es, natürliche oder technische Vorgänge auf Rechnern zu simulieren und somit neue Erkenntnisse durch numerische Experimente zu gewinnen, z. B.:

- Menschliche DNA
- Komplexe und umfassende Klimamodelle
- Simulation des Universums
- Medizinische Simulationen von Arterien, Herz, Gehirn, ...
- Verformung und Modellierung von Festkörpern
- Forschung im Bereich von erneuerbaren Energien
- Fluid Struktur Interaktion (FSI)

2.2 Warum C und C++?

C und C++ sind relativ alte und etablierte Programmiersprachen, für die viele numerische Bibliotheken entwickelt wurden.

Historische Entwicklung:

- Erster Entwurf der Programmiersprache C zu Beginn der 70er Jahre (Kernighan, Ritchie)
- Weiterentwicklungen in Richtung Objektorientierung 1979: C++ (Stroustrup)
- C-Standards

  C89 Erster ANSI-Standard 1989
  C99 Weit verbreiteter neuer C-Standard 1999
  C11 Neuester C-Standard 2011

⊕ Jahrzente an Erfahrung und Verbesserungen

Eigenschaften von C:

- Universal-Programmiersprache
- Kleiner Sprachkern, aber viele Operatoren
- Motto: Trust the programmer
- Sehr hohe Verbreitung
- Auf fast jeder Hardware verfügbar
- „Maschinennah“ (Bitmanipulationen, Pointer, . . .)
- Viele Betriebssysteme (z. B. Unix) sind zum größten Teil in C geschrieben

⊕ C- und C++-Programme sind schnell, da Code in Maschinencode umgewandelt wird (kompilierende Sprache).

Viele Skriptsprachen wie MATLAB oder Python sind interpretierende Sprachen, d h. der Code wird zur Laufzeit übersetzt.

2.3 Warum (und wann) nicht C und C++?

In einigen Situationen sind andere Sprachen besser geeignet, z. B.:

- Prototypen (kurze Programme), um Ideen, Algorithmen, etc. schnell auszuprobieren
  (In C und C++ müssten z. B. verschiedene Bibliotheken zur Verwendung komplexerer Datentypen eingebunden werden.)
- Erstellen von Grafiken
  (In C und C++ kann nicht einfach eine Funktion zur Visualisierung der Ergebnisse aufgerufen werden.)

2.4 Aufbau eines C-Programms

Inhalt der Datei `hello.c` (*Hinweise:* Mit der Endung `.c` kennzeichnen wir ein C-Programm; die Zeilennummern stehen **nicht** in der Datei, sondern wurden nachträglich eingefügt.)

```
1  /* Funktion: Ausgabe des Textes "Hello World" */
2
3  #include <stdio.h>
4
5  int main (void)
6  {
7      printf("Hello world!");
8
9      return 0;
10 }
```

Erläuterungen

1: **Kommentarzeile** zur Erläuterung, die bei der Programmausführung ignoriert wird; Kommentare sind

- alles, was zwischen `/*` und `*/` steht
- Zeilen, die mit `//` beginnen (C99-Standard)

3: **Präprozessor-Direktive** bewirkt, dass an dieser Stelle der Inhalt der Datei `stdio.h` kopiert wird

- stdio.h (**St**andar**d** **I**nput and **O**utput Library) enthält Ein- und Ausgabefunktionen wie z. B. `printf`

5: Hier wird eine **Funktion**

- mit Funktionsname `main` (Hauptfunktion)
- ganzzahligem (`int`) Ergebnis und
- ohne Argumente (Parameter `void`)

definiert.

- Ein C-Programm enthält *genau eine* `main()`-Funktion.
- Mit der `main()`-Funktion beginnt die spätere Programmausführung.

6, 10: Geschweifte Klammern `{` und `}` begrenzen **Programmblöcke**; die Zeilen zwischen den geschweiften Klammern enthalten **Anweisungen**, die der Computer ausführen soll.

7: Ausgabefunktion `printf()` (in `stdio.h` deklariert) zur Ausgabe eines Textes auf den Bildschirm

- Auszugebender Text steht zwischen doppelten Anführungszeichen `"` und `"`

- `\n` bewirkt Umschalten auf die nächste Zeile am Ende der Ausgabe.
- Jede Anweisung wird mit einem Semikolon `;` beendet.

9: Rückgabewert der Funktion an das Betriebssystem (0 signalisiert i. A. *fehlerfreie* Beendigung des Programms).

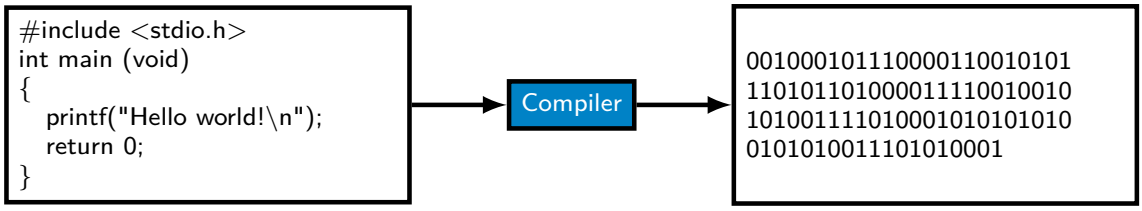
Allgemeine Bemerkungen

- Ein C-Programm besteht aus einer oder mehreren **Übersetzungseinheiten** (= Dateien mit Funktion-/ Variablen-vereinbarungen).
- Dateien werden mit einem beliebigen Texteditor erstellt und gespeichert.
- Ein C-Programm muss zuerst übersetzt (**compiliert**) werden, bevor es ausgeführt werden kann.

2.5 Der Übersetzungsvorgang

Ein **Compiler** ist ein Computerprogramm, das Quellcode (ein in einer bestimmten Programmiersprache geschriebenes Programm) in eine Form übersetzt, die von einem Computer ausgeführt werden kann.

(Quelle: `https://de.wikipedia.org/wiki/Compiler`)



Quellcode ausführbares Programm

- Für die Programmiersprache C gibt es verschiedene Compiler.
- Wir verwenden `gcc` aus der **G**NU **C**ompiler **C**ollection.

Kompilieren und Ausführen im Konsolenfenster

```
gcc hello.c -o hello_world
```

- `gcc` ist der Befehl für den C-Compiler
- `hello.c` ist die zu kompilierende Quelldatei
- `-o` ist die Option von gcc zur Wahl des Namens des ausführbaren Programms
- `hello_world` ist der Name des ausführbaren Programms
- *Hinweis:*

  ```
  gcc hello.c
  ```

  erzeugt das ausführbare Programm `a.out`
- Die Ausführung des Programms erfolgt mit dem Befehl `./Programmname`, z. B.

  `./hello_world` oder `./a.out`

2.6 Hinweise zum Schreiben von Programmen

```
1  /* Funktion: Berechnung der Fakultaet einer Zahl */
2
3  #include <stdio.h>
4
5  /* fakultaet() - Liefert die Fakultaet der Zahl n,
6        n! = 1 * 2 * ... * (n-1) * n,
7     zurueck. Dabei ist 0! = 1.                    */
8  int fakultaet (int n)
9  {
10    int i;         /* durchlaeuft die Zahlen 1, ..., n */
11    int produkt;   /* Produkt der Zahlen 1, ..., i     */
12
13    produkt = 1;
14
15    i = 1;         /* Aufmultiplizieren in einer Schleife */
16    while (i <= n)
17    {
18       produkt = produkt * i;
19       i = i + 1;
20    }
21
22    return produkt;
23 }
```

```
24
25 /* Hauptfunktion - Liest eine Zahl ein und gibt ihre
26    Fakultaet aus */
27 int main (void)
28 {
29    int zahl;
30
31    printf ("Bitte eine nichtnegative ganze Zahl eingeben: ");
32    scanf ("%d", &zahl);
33
34    printf ("Fakultaet = %d\n", fakultaet(zahl));
35
36    return 0;
37 }
```

Erläuterungen

- Dieses C-Programm besteht aus den zwei Funktionen `main()` und `fakultaet()`.
  - Die Programmausführung startet mit dem Aufruf von `main()`.
  - In `main()` wird der Wert der Variablen `zahl` mit der in `stdio.h` deklarierten Eingabefunktion `scanf()` eingelesen (Zeile 32), die Funktion `fakultaet()` mit dem aktuellen Argument `zahl` aufgerufen und das Ergebnis ausgegeben (Zeile 34).

  - Die Funktion `fakultaet()` führt die eigentliche Berechnung mithilfe einer Schleife durch (Zeilen 16-20) und liefert den Wert der Variablen `produkt` als Ergebnis an die aufrufende Funktion `main()` zurück (Zeile 22).
- In den Zeilen 10, 11 und 29 werden **Variablen vereinbart**, die innerhalb der jeweiligen Funktion zum Speichern ganzzahliger Werte verwendet werden können.
- Zeilen 13, 15, 18 und 19 sind **Zuweisungen**:
  - Die rechte Seite wird „ausgewertet“
  - Der Ergebniswert wird in der links stehende Variablen gespeichert.

Ist man mit C etwas vertraut, so ist dieses Programm unmittelbar verständlich, denn:

- Kommentare erklären, was in den Funktionen gemacht wird.
- Die Programmstruktur wird durch Einrücken und Leerzeilen gut erkennbar.

Hält man sich nicht an derartige Konventionen, so werden die Programme schwer lesbar!

3 Erste Grundlagen der Programmiersprache C

3.1 Der C-Zeichensatz

umfasst die Zeichen

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
! " # % & ’ ( ) * + , - . / : ; < = > ? [ \ ] ^ _ { |
} ~
```

sowie das Leerzeichen (SPACE), horizontal tab und vertical tab (TAB, VT) und newline (NL).

3.2 Namen

brauchen wir um Variablen, Funktionen, Macros, Typen, ... anzusprechen.

Regeln:

- Ein Name besteht aus einer Folge von Buchstaben, Ziffern und dem *Unterstrich* _
- Das erste Zeichen im Namen darf keine Ziffer sein.
- Groß- und Kleinschreibung wird unterschieden:

  `Summe` $\neq$ `summe` $\neq$ `SumMe`
- Mindestens die ersten 31 Zeichen werden unterschieden.
- Es dürfen keine **reservierten Wörter** für eigene Namen verwendet werden. Die **32 reservierten Wörter** in C sind:

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


Quiz: Welche Namen sind in C zulässig?

```
zaehler   2fel      eineEingabe   long
long1     preisIn$  Übung         x_hoch_4
my_var    Eingabe   auto          einAuto
```

3.3 Datentypen

Der **Datentyp** einer Variablen beschreibt, wie der Inhalt (die Bitfolge) zu verstehen ist, z. B. als ganze Zahl, als Gleitkommazahl, als Zeichen, als Zeichenkette (String), ...

In C gibt es drei vordefinierte elementare Datentypen:

- Ganze Zahlen (`int`, ...) z. B. `42, -7, 1038305739`
- Gleitkommazahlen (`float, double`) z. B. `3.1416, -15.67`
- Zeichen (`char`, ...) z. B. `'a', '+', 'N'`

sowie den benutzerdefinierten

- **Aufzählungstyp** (`enum`) z. B.

```
enum Richtung { NORD, OST, SUED, WEST };
enum Ampel { ROT, GELB, GRUEN };
```

Außerdem gibt es noch abgeleitete Datentypen:

- **Felder (Arrays)** zur Bündelung gleichartiger Daten
- **Strukturen (Structs)** zur Bündelung verschiedenartiger Daten
- **Zeiger (Pointer)** zur Verwaltung dynamischer Datenmengen

Ganzzahlige Datentypen
sind Variationen des Datentyps `int` durch sog. **Qualifier**, die sich durch ihren Wertebereich und ihren Speicherbedarf unterscheiden

- Garantierter Wertebereich: -32 768, ..., 32 767
- `signed` gibt an, dass positive und negative Zahlen dargestellt werden; Wertebereich i. A. $-2^{n-1}, \ldots, 2^{n-1}-1$ bei n bit; Default für `int`
- `unsigned` erlaubt nur nichtnegative Zahlen; i. A. $0, \ldots, 2^n-1$ bei n bit
- `short` für kleine Zahlen (Speicherbedarf höchstens so viele bits wie `int`)
- `long` für große Zahlen (Speicherbedarf mindestens so viele bits wie `int`)
- Es gilt:

$$\texttt{short} \subseteq \texttt{int} \subseteq \texttt{long}$$

Bemerkung: Theoretisch kann für Zahlen von 0 bis 255 auch der Datentyp `char` verwendet werden.

Gleitkommazahlen

- 3 verschiedene Datentypen: `float`, `double` und `long double`
- Codierung heute i. A. im Gleitkomma-Format gemäß IEEE-Standard
- `float` (meist als IEEE single precision realisiert)
  - mindestens 6 Dezimalstellen
  - maximaler Betrag $\geq 10^{37}$, minimaler positiver Betrag $\leq 10^{-37}$
- `double` (meist als IEEE double precision realisiert)
  - Wertebereich mindestens wie bei `float`
- `long double` (oft als IEEE extended precision realisiert)
  - Wertebereich mindestens wie bei `double`
- Bei Realisierung mit IEEE-Formaten gilt:

$$\texttt{float} \subseteq \texttt{double} \subseteq \texttt{long double}$$

Zeichen

Eine `char`-Variable kann ein einzelnes Zeichen speichern, z. B.

```
char c = 'A';
```

- Zeichenkonstanten werden in einfache Anführungszeichen ' und ' eingeschlossen.
- Zur Darstellung werden mindestens (i. A. genau) 8 bits verwendet.
- Häufigste Codierung: ASCII-Code; es gibt aber auch andere Zeichen-Codierungen
  Achtung: Man sollte sich nicht auf eine bestimmte Zeichencodierung verlassen!

3.4 Variablen

Eine **Variable** ist ein Objekt eines Standard-Datentyps. Ihr Zustand wird durch einen Wert des entsprechenden Typs bestimmt und kann sich im Verlauf eies Programms ändern.

- Jede Variable muss vor ihrer ersten Verwendung vereinbart (**deklariert**) werden. Dabei wird
  - ein Name zur Ansprechung der Variablen eingeführt
  - Speicherplatz für die Darstellung des Wertes reserviert

```
Datentyp Name1, Name2;
```

- **Initialisierung** einer Variablen mit einem Wert hat die Form

```
Datentyp Name = Konstante;
```

- Beispiel:

```
1 int    row, column; /* Deklaration von zwei int-Variablen */
2 double temperature; /* Deklaration einer double-Variablen */
3
4 /* Initialisierung der Variablen mit Werten */
5 row         = 1;
6 column      = 2;
7 temperature = 3.0;
```


- *Hinweis:* Variablen können auch bei der Deklaration initialisiert werden, z. B.

```
1 int    row = 1, column = 2;
2 double temperature = 3.0;
```

- Das Attribut `const` sorgt dafür, dass die Variable „schreib-geschützt“ ist, d. h. nach der Initialisierung darf ihr nichts mehr zugewiesen werden. Daher i. A. Initialisierung direkt bei der Definition, z. B.

```
const double tolerance = 1.0e-8;
```

- Variablen können unterschiedliche Gültigkeitsbereiche haben:

  Externe (globale) Variablen
  - werden außerhalb aller Funktionen vereinbart
  - werden automatisch mit 0 initialisiert
  - sind im ganzen Programm gültig

  Lokale Variablen
  - werden innerhalb eines Blockes (gekennzeichnet durch { und }) vereinbart
  - sind nur innerhalb dieses Blockes gültig
  - werden ohne explizite Angabe eines Initialisierungswertes nicht initialisiert

- Es können mehrere Variablen gleichen Namens vereinbart werden
  - Es gilt: „Innere“ Namen verdecken „äußere“ Namen
  - Dies ist aus Gründen der Übersichtlichkeit nicht empfehlenswert!


```c
/* Funktion: Sichtbarkeit von Variablen */

#include <stdio.h>

int a = 3, b = 3; /* globale Variablen */

int main (void)
{
   int a = 7;      /* in main() lokale Variable */
   printf ("Zeile 10: a = %d, b = %d\n", a, b);

   {
      int a = -1, b = 13; /* in Block1 lokale Variablen */
      printf ("Zeile 14: a = %d, b = %d\n", a, b);
   }

   printf ("Zeile 17: a = %d, b = %d\n", a, b);
   {
      int b = -4; /* in Block2 lokale Variable */
      printf ("Zeile 20: a = %d, b = %d\n", a, b);
   }

   return 0;
}
```

Was gibt das Programm aus? („Inneres“ verdeckt „Äußeres“)

3.5 Explizite Typ-Anpassung (type cast)

Eine Anweisung der Form

`(Ziel-Typ) Ausdruck`

konvertiert den Wert von `Ausdruck` in einen Wert vom `Ziel-Typ`, z. B.

```
1 int    i;
2 double d;
3
4 i = (int) 3.14;        /* entspricht i = 3 (double -> int)    */
5 d = (double) (i + 1); /* entspricht d = 4.0 (int -> double) */
```

Bemerkungen:

- Bei Umwandlung in einen „kleineren“ Typ kann Information verloren gehen!
- Bei Umwandlung von Gleitkomma-Zahlen `x` in ganzzahlige Werte werden die Nachkomma-Stellen abgeschnitten.

  Rundung zur nächsten ganzen Zahl erreicht man für nichtnegative Werte `x` durch

  ```
  (int) (x + 0.5);
  ```


3.6 Arithmetik- und Zuweisungsoperationen

Grundlegende mathematische Operatoren und ihre Bedeutung

+	Addition, Vorzeichen
-	Subtraktion, Vorzeichen
/	Division
%	Rest der ganzzahligen Division (modulo)
=	Zuweisung

- Für alle elementaren Datentypen definiert
- Es gilt die „Punkt-vor-Strich“-Regel
- **Klammerung** möglich und zu empfehlen
- Spezielle **Zuweisungsoperatoren**: Ausdrücke der Form

  `Variable = Variable` $\circ$ `Ausdruck`

  mit einem Operator $\circ \in \{+, -, *, /, \%\}$ können durch

  `Variable =`$\circ$ `Ausdruck`

  abgekürzt werden, z. B.

  ```
  a += b; /* entspricht a = a + b */
  ```

3.7 Mathematische Standardfunktionen

Viele mathematischen Funktionen sind in der Mathematik-Bibliothek `math.h` deklariert, z. B.

`sqrt (x)`	$\sqrt{x}$
`pow (x, y)`	x^y
`log (x)`	$\ln(x)$
`log10 (x)`	$\log_{10}(x)$
`fabs (x)`	$\lvert x \rvert$
`sin (x)`	$\sin(x)$
`cos (x)`	$\cos(x)$
`ceil (x)`	$\lceil x \rceil$
`floor (x)`	$\lfloor x \rfloor$

Bemerkungen:

- Argumente und Resultate sind i. A. vom Datentyp `double`
- Die Mathematik-Bibliothek muss eingebunden werden

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

- Beim Übersetzen muss mit der Mathematik-Bibliothek gelinkt werden

```
gcc programm.c -lm
```


3.8 Ausgabe mit `printf()`

Die Anweisung

```
printf ("Formatstring", Wert1, Wert2, Variable, ...);
```

gibt den `Formatstring` auf die **Standardausgabe** (üblicherweise den Bildschirm) aus.

Bemerkungen:

- Die Standard-Input-Output-Bibliothek muss eingebunden werden

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

- Die im `Formatstring` verwendeten **Umsetzungssequenzen** (gekennzeichnet durch `%T`, wobei `T` den Datentyp angibt) werden der Reihe nach ersetzt, d. h. die i-te Sequenz wird durch den i-ten nach dem Formatstring angegeben Wert ersetzt.
- Der Backslash `\` im `Formatstring` markiert sog. Escape-Sequenzen, mit denen sich z. B. Zeilenumbrüche (`\n`) realisieren lassen.
- Auszugebende Variablen bzw. Werte werden durch Kommata getrennt hinter dem `Formatstring` aufgelistet.

Beispiel:

```c
/* Funktion: Berechnung der Flaeche eines Rechtecks */

#include <stdio.h>

int main (void)
{
    int    hoehe  = 2;
    double breite = 3;

    printf ("Hoehe = %d, Breite = %f, Flaeche = %f\n",
            hoehe, breite, hoehe*breite);

    return 0;
}
```

erzeugt die Ausgabe

```
Hoehe = 2, Breite = 3.000000, Flaeche = 6.000000
```

`printf()`-Umsetzungszeichen sind z. B.

- `%d`, `%i` für `int` (Ausgabe: `ddd`)
- `%f` für `float` und `double` (Ausgabe: `[-]ddd.ddd`)
- `%e`, `%E` für `float` und `double` (Ausgabe: `[-]ddd.ddd[{e|E}]±dd`)
- `%c` für `char`

Formatierte Ausgabe

- Zwischen `%` und Umsetzungszeichen können Angaben für die Ausgabefeldbreite und Präzision eingefügt werden, z. B.

Anweisung	Ausgabe
`printf ("|%d|", 1234);`	`|1234|`
`printf ("|%6d|", 1234);`	`|  1234|`
`printf ("|%.6d|", 1234);`	`|001234|`
`printf ("|%-6d|", 1234);`	`|1234  |`
`printf ("|%+6d|", 1234);`	`| +1234|`
`printf ("|%f|", 1.2345);`	`|1.234500|`
`printf ("|%6.3f|", 1.2345);`	`| 1.235|`

- *Achtung:* Vorzeichen und Dezimalpunkt zählen zur Ausgabefeldbreite!

3.9 Eingabe mit `scanf()`

Die Anweisung

```
scanf ("Formatstring", Adresse1, Adresse2, &Variable, ...);
```

liest gemäß `Formatstring` Werte von der **Standardeingabe** (üblicherweise der Tastatur) ein.

Bemerkungen:

- Die Standard-Input-Output-Bibliothek muss eingebunden werden

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

- Die im `Formatstring` verwendeten **Umsetzungssequenzen** (gekennzeichnet durch `%T`, wobei `T` den Datentyp angibt) geben an, wie die Werte zu interpretieren sind; der i-te gelesen Wert wird im Speicher bei der Adresse `Adresse`i abgelegt.
- Die Adresse einer Variablen erhält man mit dem **Adressoperator** `&`.
- Ein WhiteSpace-Zeichen (Leer-, Return- oder Tabulator-Zeichen) im Formatstring erkennt eine *beliebig lange Folge* von WhiteSpaces in der Eingabe; andere Zeichen werden „wie geschrieben“ erwartet.

- Die Feldbreite kann wie bei `printf ()` angegeben werden; sonst wird jeweils bis zum nächsten WhiteSpace-Zeichen gelesen.

  Beispiel:

```
1  /* Funktion: Berechnung der Flaeche eines Rechtecks */
2
3  #include <stdio.h>
4
5  int main (void)
6  {
7     double hoehe, breite;
8
9     printf ("Bitte Hoehe und Breite eingeben: ");
10    scanf ("%lf %lf", &hoehe, &breite);
11
12    printf ("Flaeche des Rechtecks = %f\n", hoehe*breite);
13
14    return 0;
15 }
```

  Programmablauf

```
Bitte Hoehe und Breite eingeben: 2   3.4
Flaeche des Rechtecks = 6.800000
```

  `scanf()`-Umsetzungszeichen sind meist wie bei `printf()`
  Achtung: `%lf`, `%le`, `%lE` für `double`, `%f`, `%e`, `%E` für `float`