Einführung in das Programmieren für Technische Mathematik

Dr. Thomas Führer

Michele Ruggeri, MSc

Marcus Page, MSc

Prof. Dr. Dirk Praetorius

Fr. 10:15 - 11:45, Freihaus HS 8

Institut für Analysis
und Scientific Computing

Formalia

- Rechte & Pflichten
- Benotung
- Anwesenheitspflicht
- Literatur

EPROG-Homepage

- http://www.asc.tuwien.ac.at/eprog/
  - alle Regeln & Pflichten & Benotungsschema
  - Download der Folien & Übungen
  - Termine der VO und UE
  - freiwilliges UE-Material (alte Tests!)
  - Evaluation & Notenspiegel

Literatur

- VO-Folien zum Download auf Homepage
- formal keine weitere Literatur nötig
- zwei freie Bücher zum Download auf Homepage
- weitere Literaturhinweise auf der nächsten Folie

„freiwillige“ Literatur

- Brian Kernighan, Dennis Ritchie
  Programmieren in C
- Klaus Schmaranz
  Softwareentwicklung in C
- Ralf Kirsch, Uwe Schmitt
  Programmieren in C, eine mathematikorientierte Einführung

- Bjarne Stroustrup
  Die C++ Programmiersprache
- Klaus Schmaranz
  Softwareentwicklung in C++
- Dirk Louis
  Jetzt lerne ich C++
- Jesse Liberty
  C++ in 21 Tagen

Das erste C-Programm

- Programm & Algorithmus
- Source-Code & Executable
- Compiler & Interpreter
- Syntaxfehler & Laufzeitfehler
- Wie erstellt man ein C-Programm?

- `main`
- `printf` (Ausgabe von Text)
- `#include <stdio.h>`

Programm

- Ein Computerprogramm oder kurz Programm ist eine Folge von Anweisungen, die den Regeln einer Programmiersprache genügen, um auf einem Computer eine bestimmte Funktionalität, Aufgaben- oder Problemstellung bearbeiten oder lösen zu können.
  - Anweisungen = Deklarationen und Instruktionen
    - Deklaration = z.B. Definition von Variablen
    - Instruktion = „tue etwas“
  - BSP: suche einen Telefonbucheintrag
  - BSP: berechne den Wert eines Integrals

Algorithmus

- Ein Algorithmus ist eine aus endlich vielen Schritten bestehende, eindeutige und ausführbare Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen.
  - BSP: Berechne die Lösung eines linearen Gleichungssystems mittels Elimination
  - BSP: Berechne die Nullstelle eines quadratischen Polynoms mittels p-q-Formel
- IdR. unendlich viele Algorithmen für ein Problem
  - IdR. sind Algorithmen unterschiedlich „gut“
    - Was heißt „gut“? (später!)

Source-Code

- in Programmiersprache geschriebener Text eines Computerprogramms
- wird bei Ausführung bzw. Compilieren schrittweise abgearbeitet
- im einfachsten Fall: sequentiell
  - Programmzeile für Programmzeile
  - von oben nach unten

Programmiersprachen

- Grobe Unterscheidung in Interpreter- und Compiler-basierte Sprachen
- Interpreter führt Source-Code zeilenweise bei der Übersetzung aus
  - d.h. Übersetzen & Ausführen ist gleichzeitig
  - z.B. Matlab, Java, PHP
- Compiler übersetzt Source-Code in ein ausführbares Programm (Executable)
  - Executable ist eigenständiges Programm
  - d.h. (1) Übersetzen, dann (2) Ausführen
  - z.B. C, C++, Fortran
- Alternative Unterscheidung (siehe Schmaranz)
  - imperative Sprachen, z.B. Matlab, C, Fortran
  - objektorientierte Sprachen, z.B. C++, Java
  - funktionale Sprachen, z.B. Lisp

Achtung

- C ist Compiler-basierte Programmiersprache
- Compilierter Code ist *systemabhängig*,
  - d.h. Code läuft idR. nur auf dem System, auf dem er compiliert wurde
- Source-Code ist *systemunabhängig*,
  - d.h. er sollte auch auf anderen Systemen compiliert werden können.
- C-Compiler unterscheiden sich leicht
  - Bitte vor Übung alle Programme auf der `lva.student.tuwien.ac.at` mit dem Compiler `gcc` compilieren und testen
  - nicht-lauffähiger Code = schlechter Eindruck und ggf. schlechtere Note...

Wie erstellt man ein C-Programm?

- Starte Editor Emacs aus einer Shell mit `emacs &`
  - Die wichtigsten Tastenkombinationen:
    - `C-x C-f` = Datei öffnen
    - `C-x C-s` = Datei speichern
    - `C-x C-c` = Emacs beenden
- Öffne eine (ggf. neue) Datei `name.c`
  - Endung `.c` ist Kennung eines C-Programms
- Die ersten beiden Punkte kann man auch simultan erledigen mittels `emacs name.c &`
- Schreibe den sog. *Source-Code* (= C-Programm)
- Abspeichern mittels `C-x C-s` nicht vergessen
- Compilieren z.B. mit `gcc name.c`
- Falls Code fehlerfrei, erhält man *Executable* `a.out` unter Windows: `a.exe`
- Diese wird durch `a.out` bzw. `./a.out` gestartet
- Compilieren mit `gcc name.c -o output` erzeugt Executable `output` statt `a.out`

Das erste C-Programm

```
1  #include <stdio.h>
2
3  main() {
4    printf("Hello World!\n");
5  }
```

- Zeilennummern gehören *nicht* zum Code (sind lediglich Referenzen auf Folien)
- Jedes C-Programm besitzt die Zeilen 3 und 5.
- Die Ausführung eines C-Programms startet *immer* bei `main()` – egal, wo `main()` im Code steht
- Klammern {...} schließen in C sog. *Blöcke* ein
- Hauptprogramm `main()` bildet immer einen Block
- Logische Programmzeilen enden mit Semikolon, vgl. 4
- `printf` gibt Text aus (in Anführungszeichen),
  - `\n` macht einen Zeilenumbruch
- Anführungszeichen *müssen* in derselben Zeile sein
- Zeile 1: Einbinden der Standardbibliothek für Input-Output (später mehr!)

Syntaxfehler

- Syntax = Wortschatz (Befehle) & Grammatik einer Sprache (Was man wie verbinden kann...)
- Syntaxfehler = Falsche Befehle oder Verwendung
  - merkt Compiler und gibt Fehlermeldung

```
1  main() {
2    printf("Hello World!\n");
3  }
```

- Fehlt Einbindung der `stdio.h`
  - Compilieren liefert Fehlermeldung:

```
wrongworld1.c:2: warning: incompatible implicit
declaration of built-in function printf
```

```
1  #include <stdio.h>
2
3  main() {
4    printf("Hello World!\n")
5  }
```

- Fehlt Semikolon am Zeilenende 4
  - Compilieren liefert Fehlermeldung:

```
wrongworld2.c:5: error: syntax error before } token
```

Laufzeitfehler

- Fehler, der erst bei Programm-Ausführung auftritt
  - viel schwerer zu finden
  - durch sorgfältiges Arbeiten möglichst vermeiden

Variablen

- Was sind Variable?
- Deklaration & Initialisierung
- Datentypen `int` und `double`
- Zuweisungsoperator `=`
- arithmetische Operatoren `+ - * / %`
- Type Casting

- `int, double`
- `printf` (Ausgabe von Variablen)
- `scanf` (Werte über Tastatur einlesen)

Variable

- Variable = symbolischer Name für Speicherbereich
- Variable in Math. und Informatik verschieden:
  - Mathematik: Sei $x \in \mathbb{R}$ fixiert x
  - Informatik: `x=5` weist `x` den Wert 5 zu, Zuweisung kann jederzeit geändert werden z.B. `x=7`

Datentypen

- Bevor man Variable benutzen darf, muss man idR. erklären, welchen Typ Variable haben soll
- Elementare Datentypen:
  - Gleitkommazahlen (ersetzt $\mathbb{Q}$, $\mathbb{R}$), z.B. `double`
  - Integer, Ganzzahlen (ersetzt $\mathbb{N}$, $\mathbb{Z}$), z.B. `int`
  - Zeichen (Buchstaben), idR. `char`
- `int x;` deklariert Variable `x` vom Typ `int`

Deklaration

- Deklaration = das Anlegen einer Variable
  - d.h. Zuweisung von Speicherbereich auf einen symbolischen Namen & Angabe des Datentyps
  - Zeile `int x;` deklariert Variable `x` vom Typ `int`
  - Zeile `double var;` deklariert `var` vom Typ `double`

Initialisierung

- Durch Deklaration einer Variablen wird lediglich Speicherbereich zugewiesen
- Falls noch kein konkreter Wert zugewiesen:
  - Wert einer Variable ist zufällig
- Deshalb direkt nach Deklaration der neuen Variable Wert zuweisen, sog. Initialisierung
  - `int x;` (Deklaration)
  - `x = 0;` (Initialisierung)
- Deklaration & Initialisierung auch in einer Zeile möglich: `int x = 0;`

Ein erstes Beispiel zu `int`

```
1  #include <stdio.h>
2
3  main() {
4    int x = 0;
5
6    printf("Input: x=");
7    scanf("%d",&x);
8    printf("Output: x=%d\n",x);
9  }
```

- Einbinden der Input-Output-Funktionen (Zeile 1)
  - `printf` gibt Text (oder Wert einer Var.) aus
  - `scanf` liest Tastatureingabe ein in eine Variable
- Prozentzeichen `%` in Zeile 7/8 leitet Platzhalter ein

Datentyp	Platzhalter `printf`	Platzhalter `scanf`
`int`	`%d`	`%d`
`double`	`%f`	`%lf`

- Beachte `&` bei `scanf` in Zeile 7
  - `scanf("%d",&x)`
  - aber: `printf("%d",x)`
- Wenn man `&` vergisst ⇒ Laufzeitfehler
  - Compiler merkt Fehler nicht (kein Syntaxfehler!)
  - Sorgfältig arbeiten!

Dasselbe Beispiel zu `double`

```
1  #include <stdio.h>
2
3  main() {
4    double x = 0;
5
6    printf("Input: x=");
7    scanf("%lf",&x);
8    printf("Output: x=%f\n",x);
9  }
```

- Beachte Platzhalter in Zeile 7/8
  - `scanf("%lf",&x)`
  - aber: `printf("%f",x)`
- Verwendet man `%f` in 7 ⇒ Falsches Einlesen!
  - vermutlich Laufzeitfehler!
  - sorgfältig arbeiten!

Zuweisungsoperator

```
1   #include <stdio.h>
2
3   main() {
4     int x = 1;
5     int y = 2;
6
7     int tmp = 0;
8
9     printf("a) x=%d, y=%d, tmp=%d\n",x,y,tmp);
10
11    tmp = x;
12    x = y;
13    y = tmp;
14
15    printf("b) x=%d, y=%d, tmp=%d\n",x,y,tmp);
16  }
```

- Das einfache Gleich `=` ist Zuweisungsoperator
  - Zuweisung immer rechts nach links!
- Zeile `x = 1;` weist den Wert auf der rechten Seite der Variablen `x` zu
- Zeile `x = y;` weist den Wert der Variablen `y` der Variablen `x` zu
  - insb. haben `x` und `y` danach denselben Wert
  - d.h. Vertauschen der Werte nur mit Hilfsvariable
- Output:

```
a) x=1, y=2, tmp=0
b) x=2, y=1, tmp=1
```


Arithmetische Operatoren

- Bedeutung eines Operators kann vom Datentyp abhängen!
- Operatoren auf Ganzzahlen:
  - `a=b`, `-a` (Vorzeichen)
  - `a+b`, `a-b`, `a*b`, `a/b` (Division ohne Rest), `a%b` (Divisionsrest)
- Operatoren auf Gleitkommazahlen:
  - `a=b`, `-a` (Vorzeichen)
  - `a+b`, `a-b`, `a*b`, `a/b` (“normale“ Division)
- Achtung: `2/3` ist Ganzzahl-Division, also Null!
- Notation für Gleitkommazahlen:
  - Vorzeichen –, falls negativ
  - Vorkommastellen
  - Dezimalpunkt
  - Nachkommastellen
  - `e` oder `E` mit *ganzzahligem* Exponenten (10er Potenz!), z.B. $2e2 = 2E2 = 2 \cdot 10^2 = 200$
    - Wegfallen darf entweder Vor- oder Nachkommastelle (sonst sinnlos!)
    - Wegfallen darf entweder Dezimalpunkt oder `e` bzw. `E` mit Exponent (sonst Integer!)
- Also: `2./3.` ist Gleitkommadivision $\approx 0.\overline{6}$

Type Casting

- Operatoren können auch Variablen verschiedener Datentypen verbinden
- Vor der Ausführung werden beide Variablen auf denselben Datentyp gebracht (Type Casting)

```
1   #include <stdio.h>
2
3   main() {
4     int x = 1;
5     double y = 2.5;
6
7     int sum_int = x+y;
8     double sum_dbl = x+y;
9
10    printf("sum_int = %d\n",sum_int);
11    printf("sum_dbl = %f\n",sum_dbl);
12  }
```

- Welchen Datentyp hat `x+y` in Zeile 7, 8?
  - Den mächtigeren Datentyp, also `double`!
  - Type Casting von Wert `x` auf `double`
- Zeile 7: Type Casting, da `double` auf `int` Zuweisung
  - durch Abschneiden, nicht durch Rundung!
- Output:

```
sum_int = 3
sum_dbl = 3.500000
```


Implizites Type Casting

```
1   #include <stdio.h>
2
3   main() {
4     double dbl1 = 2 / 3;
5     double dbl2 = 2 / 3.;
6     double dbl3 = 1E2;
7     int int1 = 2;
8     int int2 = 3;
9
10    printf("a) %f\n",dbl1);
11    printf("b) %f\n",dbl2);
12
13    printf("c) %f\n",dbl3 * int1 / int2);
14    printf("d) %f\n",dbl3 * (int1 / int2) );
15  }
```

- Output:

```
a) 0.000000
b) 0.666667
c) 66.666667
d) 0.000000
```

- Warum Ergebnis 0 in `a)` und `d)` ?
  - 2, 3 sind `int` ⇒ `2/3` ist Ganzzahl-Division
- Werden Variablen verschiedenen Typs durch arith. Operator verbunden, Type Casting auf „gemeinsamen“ (mächtigeren) Datentyp
  - vgl. Zeile 5, 13, 14
  - 2 ist `int`, `3.` ist `double` ⇒ `2/3.` ergibt `double`

Explizites Type Casting

```
1   #include <stdio.h>
2
3   main() {
4     int a = 2;
5     int b = 3;
6     double dbl1 = a / b;
7     double dbl2 = (double) (a / b);
8     double dbl3 = (double) a / b;
9     double dbl4 = a / (double) b;
10
11    printf("a) %f\n",dbl1);
12    printf("b) %f\n",dbl2);
13    printf("c) %f\n",dbl3);
14    printf("d) %f\n",dbl4);
15  }
```

- Kann dem Compiler mitteilen, in welcher Form eine Variable interpretiert werden muss
  - Dazu Ziel-Typ in Klammern voranstellen!
- Output:

  a) 0.000000

  b) 0.000000

  c) 0.666667

  d) 0.666667
- In Zeile 7, 8, 9: Explizites Type Casting (jeweils von `int` zu `double`)
- In Zeile 8, 9: Implizites Type Casting

Fehlerquelle beim Type Casting

```
1   #include <stdio.h>
2
3   main() {
4     int a = 2;
5     int b = 3;
6     double dbl = (double) a / b;
7
8     int i = dbl;
9
10    printf("a) %f\n",dbl);
11    printf("b) %f\n",dbl*b);
12    printf("c) %d\n",i);
13    printf("d) %d\n",i*b);
14  }
```

- Output:

  a) 0.666667

  b) 2.000000

  c) 0

  d) 0
- Implizites Type Casting sollte man vermeiden!
  - d.h. Explizites Type Casting verwenden!
- Bei Rechnungen Zwischenergebnisse in richtigen Typen speichern!

Einfache Verzweigung

- Logische Operatoren `== != > >= < <=`
- Logische Junktoren `! && ||`
- Wahrheit und Falschheit bei Aussagen
- Verzweigung
- `if`
- `if - else`

Logische Operatoren

- Es seien `a, b` zwei Variablen (auch versch. Typs!)
  - Vergleich (z.B. `a < b`) liefert Wert 1, falls wahr
  - bzw. 0, falls falsch
- Übersicht über Vergleichsoperatoren:

`==`	Gleichheit (ACHTUNG mit Zuweisung!)
`!=`	Ungleichheit
`>`	echt größer
`>=`	größer oder gleich
`<`	echt kleiner
`<=`	kleiner oder gleich

- Stets bei Vergleichen Klammer setzen!
  - fast immer unnötig, aber manchmal eben nicht!
- Weitere logische Iunktoren:

`!`	nicht
`&&`	und
`||`	oder

Logische Verkettung

```
1   #include <stdio.h>
2
3   main() {
4     int result = 0;
5
6     int a = 3;
7     int b = 2;
8     int c = 1;
9
10    result = (a > b > c);
11    printf("a) result=%d\n",result);
12
13    result = (a > b) && (b > c);
14    printf("b) result=%d\n",result);
15  }
```

- Output:

  ```
  a) result=0
  b) result=1
  ```

- Warum ist Aussage in 10 falsch, aber in 13 wahr?
  - Auswertung von links nach rechts:
    - `a > b` ist wahr, also mit `1` bewertet
    - `1 > c` ist falsch, also mit `0` bewertet
    - Insgesamt wird `a > b > c` mit falsch bewertet!
  - Aussage in 10 ist also nicht korrekt formuliert!

if-else

- einfache Verzweigung: *Wenn - Dann - Sonst*
- `if (condition) statementA else statementB`

- nach `if` steht Bedingung *stets* in runden Klammern
- nach Bedingung steht *nie* Semikolon
- Bedingung ist *falsch*, falls sie 0 ist bzw. mit 0 bewertet wird, sonst ist die Bedingung *wahr*
  - Bedingung wahr ⇒ `statementA` wird ausgeführt
  - Bedingung falsch ⇒ `statementB` wird ausgeführt
- Statement ist
  - entweder eine Zeile
  - oder mehrere Zeilen in geschwungenen Klammern { ... }, sog. Block
- `else`-Zweig ist optional
  - d.h. `else statementB` darf entfallen

Beispiel zu if

```
1   #include <stdio.h>
2
3   main() {
4     int x = 0;
5
6     printf("Input x=");
7     scanf("%d",&x);
8
9     if (x < 0)
10      printf("x=%d is negative\n",x);
11
12    if (x > 0) {
13      printf("x=%d is positive\n",x);
14    }
15  }
```

- abhängige Zeilen einrücken (Lesbarkeit!)
- WARNUNG: Nicht-Verwendung von Blöcken `{...}` ist fehleranfällig
- könnte zusätzlich `else` in Zeile 11 schreiben
  - da `if`'s sich ausschließen

Beispiel zu if-else

```
1   #include <stdio.h>
2
3   main() {
4     int var1 = -5;
5     double var2 = 1e-32;
6     int var3 = 5;
7
8     if (var1 >= 0) {
9       printf("var1 >= 0\n");
10    }
11    else {
12      printf("var1 < 0\n");
13    }
14
15    if (var2) {
16      printf("var2 != 0, i.e., cond. is true\n");
17    }
18    else {
19      printf("var2 == 0, i.e., cond. is false\n");
20    }
21
22    if ( (var1 < var2) && (var2 < var3) ) {
23      printf("var2 lies between the others\n");
24    }
25  }
```

- Output:

  ```
  var1 < 0
  var2 != 0, i.e., cond. is true
  var2 lies between the others
  ```

Gerade oder Ungerade?

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int x = 0;
 5
 6    printf("Input x=");
 7    scanf("%d",&x);
 8
 9    if (x > 0) {
10      if (x%2 != 0) {
11        printf("x=%d is odd\n",x);
12      }
13      else {
14        printf("x=%d is even\n",x);
15      }
16    }
17    else {
18      printf("Error: Input has to be positive!\n");
19    }
20  }
```

- Programm überprüft, ob eingegebene Zahl `x` gerade Zahl ist oder nicht
- Man kann Verzweigungen schachteln:
  - Einrückungen machen Code übersichtlicher
    - formal nicht notwendig, aber trotzdem!
  - Abhängigkeiten werden verdeutlicht

Zwei Zahlen aufsteigend sortieren

```
 1  #include <stdio.h>
 2
 3  main() {
 4    double x1 = 0;
 5    double x2 = 0;
 6    double tmp = 0;
 7
 8    printf("Unsortierte Eingabe:\n");
 9    printf(" x1=");
10    scanf("%lf",&x1);
11    printf(" x2=");
12    scanf("%lf",&x2);
13
14    if (x1 > x2) {
15      tmp = x1;
16      x1 = x2;
17      x2 = tmp;
18    }
19
20    printf("Aufsteigend sortierte Ausgabe:\n");
21    printf(" x1=%f\n",x1);
22    printf(" x2=%f\n",x2);
23  }
```

- Eingabe von zwei Zahlem $x_1, x_2 \in \mathbb{R}$
- Zahlen werden aufsteigend sortiert
  - ggf. vertauscht
- Ergebnis wird ausgegeben

Innen oder Außen?

```
 1  #include <stdio.h>
 2  main() {
 3    double r = 0;
 4    double x1 = 0;
 5    double x2 = 0;
 6    double z1 = 0;
 7    double z2 = 0;
 8    double dist2 = 0;
 9
10    printf("Radius des Kreises r=");
11    scanf("%lf",&r);
12    printf("Mittelpunkt des Kreises x = (x1,x2)\n");
13    printf(" x1=");
14    scanf("%lf",&x1);
15    printf(" x2=");
16    scanf("%lf",&x2);
17    printf("Punkt in der Ebene z = (z1,z2)\n");
18    printf(" z1=");
19    scanf("%lf",&z1);
20    printf(" z2=");
21    scanf("%lf",&z2);
22
23    dist2 = (x1-z1)*(x1-z1) + (x2-z2)*(x2-z2);
24    if ( dist2 < r*r ) {
25        printf("z liegt im Kreis\n");
26    }
27    else {
28      if ( dist2 > r*r ) {
29        printf("z liegt ausserhalb vom Kreis\n");
30      }
31      else {
32        printf("z liegt auf dem Kreisrand\n");
33      }
34    }
35  }
```


Gleichheit vs. Zuweisung

- Nur Erinnerung: `if (a==b)` vs. `if (a=b)`
  - beides ist syntaktisch korrekt!
  - `if (a==b)` ist Abfrage auf Gleichheit
    - ist vermutlich so gewollt...
  - ABER: `if (a=b)`
    - weist `a` den Wert von `b` zu
    - Abfrage, ob $a \neq 0$
    - ist schlechter Programmierstil!