Einführung in das Programmieren für Technische Mathematik

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
  - vollständige Folien aus dem letzten Semester
  - aktuelle Folien wöchentlich jeweils vor Vorlesung
- 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 Gauß-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!)

main() vs. int main()

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

- Sprache C hat sich über Jahre verändert
- `main() {` in Zeile 3 ist C89-Standard
- C99 und C++ erfordern `int main() {`

```
1 #include <stdio.h>
2
3 int main() {
4   printf("Hello World!\n");
5   return 0;
6 }
```

- Bedeutung:
  - main() kommuniziert mit Betriebssystem
  - Rückgabewert (return) gibt ggf. Fehlercode
  - Rückgabe Null = kein Fehler aufgetreten
- in diesem Fall auch return 0; sinnvoll
  - Genaueres später $\longrightarrow$ Funktionen!
- Konsequenz:
  - Falls Ihr C-Compiler Code oben nicht akzeptiert, Code unten verwenden!
  - Code von Folien entsprechend anpassen!

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 }
```

- Warnung, weil Einbindung der `stdio.h` fehlt

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

- C++ Compiler liefert Fehler wegen `int main() {`

  ```
  wrongworld1.c:1: error: C++ requires a type specifier
  for all declarations
  ```

```
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`

Variablen-Namen

- bestehen aus Zeichen, Ziffern und Underscore `_`
  - maximale Länge = 31
  - erstes Zeichen darf keine Ziffer sein
- Klein- und Großschreibung wird unterschieden
  - d.h. `Var`, `var`, `VAR` sind 3 verschiedene Variablen
- **Konvention:** Namen sind `klein_mit_underscores`

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 $\Rightarrow$ 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 $\Rightarrow$ 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` $\Rightarrow$ `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` $\Rightarrow$ `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 $\Rightarrow$ `statementA` wird ausgeführt
  - Bedingung falsch $\Rightarrow$ `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 }
```

- Eine Bedingung ist wahr, falls Wert $\neq 0$
  - z.B. Zeile 15, aber besser: `if (var2 != 0)`
- 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 Zahlen $x_1, x_2 \in \mathbb{R}$
- Zahlen werden aufsteigend sortiert
  - ggf. vertauscht
- Ergebnis wird ausgegeben

Innen oder Außen?

```
 1 #include <stdio.h>
 2
 3 main() {
 4   double r = 0;
 5   double x1 = 0;
 6   double x2 = 0;
 7   double z1 = 0;
 8   double z2 = 0;
 9   double dist2 = 0;
10
11   printf("Radius des Kreises r=");
12   scanf("%lf",&r);
13   printf("Mittelpunkt des Kreises x = (x1,x2)\n");
14   printf(" x1=");
15   scanf("%lf",&x1);
16   printf(" x2=");
17   scanf("%lf",&x2);
18   printf("Punkt in der Ebene z = (z1,z2)\n");
19   printf(" z1=");
20   scanf("%lf",&z1);
21   printf(" z2=");
22   scanf("%lf",&z2);
23
24   dist2 = (x1-z1)*(x1-z1) + (x2-z2)*(x2-z2);
25   if ( dist2 < r*r ) {
26       printf("z liegt im Kreis\n");
27   }
28   else {
29     if ( dist2 > r*r ) {
30       printf("z liegt ausserhalb vom Kreis\n");
31     }
32     else {
33       printf("z liegt auf dem Kreisrand\n");
34     }
35   }
36 }
```

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!

Blöcke

- Blöcke `{...}`
- Deklaration von Variablen
- Lifetime & Scope
- Lokale & globale Variablen

Lifetime & Scope

- Lifetime einer Variable
  = Zeitraum, in dem Speicherplatz zugewiesen ist
  = Zeitraum, in dem Variable existiert
- Scope einer Variable
  = Zeitraum, in dem Variable sichtbar ist
  = Zeitraum, in dem Variable gelesen/verändert werden kann
- Scope $\subseteq$ Lifetime

Globale & Lokale Variablen

- globale Variablen
  = Variablen, die globale Lifetime haben (bis Programm terminiert)
  - eventuell lokaler Scope
  - werden am Anfang außerhalb von `main` deklariert
- lokale Variablen
  = Variablen, die nur lokale Lifetime haben
- **Konvention:** erkenne Variable am Namen
  - lokale Variablen sind `klein_mit_underscores`
  - globale Var. haben `auch_underscore_hinten_`

Blöcke

- Blöcke stehen innerhalb von { ... }
- Jeder Block startet mit Deklaration zusätzlich benötigter Variablen
  - Variablen *können/dürfen* nur am Anfang eines Blocks deklariert werden
- Die innerhalb des Blocks deklarierten Variablen werden nach Blockende vergessen (= gelöscht)
  - d.h. Lifetime endet
  - lokale Variablen
- Schachtelung { ... { ... } ... }
  - beliebige Schachtelung ist möglich
  - Variablen aus äußerem Block können im inneren Block gelesen und verändert werden, umgekehrt *nicht*. Änderungen bleiben wirksam.
    - d.h. Lifetime & Scope nur nach Innen vererbt
  - Wird im äußeren und im inneren Block Variable `var` deklariert, so wird das „äußere“ `var` überdeckt und ist erst wieder ansprechbar (mit gleichem Wert wie vorher), wenn der innere Block beendet wird.
    - d.h. äußeres `var` ist nicht im inneren Scope
    - Das ist schlechter Programmierstil!

Einfaches Beispiel

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int x = 7;
 5   printf("a) %d\n", x);
 6   x = 9;
 7   printf("b) %d\n", x);
 8   {
 9     int x = 17;
10     printf("c) %d\n", x);
11   }
12   printf("d) %d\n", x);
13 }
```

- zwei verschiedene *lokale* Variablen x
  - Deklaration + Initialisierung (Zeile 4, 9)
  - unterscheide von Zuweisung (Zeile 6)

- Output:

  a) 7

  b) 9

  c) 17

  d) 9

Komplizierteres Beispiel

```
 1 #include <stdio.h>
 2
 3 int var0 = 5;
 4
 5 main() {
 6   int var1 = 7;
 7   int var2 = 9;
 8
 9   printf("a) %d, %d, %d\n", var0, var1, var2);
10   {
11     int var1 = 17;
12
13     printf("b) %d, %d, %d\n", var0, var1, var2);
14     var0 = 15;
15     var2 = 19;
16     printf("c) %d, %d, %d\n", var0, var1, var2);
17     {
18       int var0 = 25;
19       printf("d) %d, %d, %d\n", var0, var1, var2);
20     }
21   }
22   printf("e) %d, %d, %d\n", var0, var1, var2);
23 }
```

- Output:

  a) 5, 7, 9

  b) 5, 17, 9

  c) 15, 17, 19

  d) 25, 17, 19

  e) 15, 7, 19

- zwei Variablen mit Name `var0` (Zeile 3 + 18)
  - Namenskonvention absichtlich verletzt
- zwei Variablen mit Name `var1` (Zeile 6 + 11)

Funktionen

- Funktion
- Eingabe- / Ausgabeparameter
- Call by Value / Call by Reference

- `return`
- `void`

Funktionen

- Funktion = Zusammenfassung mehrerer Anweisungen zu einem aufrufbaren Ganzen
  - `output = function(input)`
    - Eingabeparameter `input`
    - Ausgabeparameter (Return Value) `output`
- Warum Funktionen?
  - Zerlegung eines großen Problems in überschaubare kleine Teilprobleme
  - Strukturierung von Programmen (Abstraktionsebenen)
  - Wiederverwertung von Programm-Code
- Funktion besteht aus Signatur und Rumpf (Body)
  - Signatur = Fkt.name & Eingabe-/Ausgabepar.
    - Anzahl & Reihenfolge ist wichtig!
  - Rumpf = Programmzeilen der Funktion

Namenskonvention

- lokale Variablen sind `klein_mit_underscores`
- globale Var. haben `auch_underscore_hinten_`
- Funktionen sind `erstesWortKleinKeineUnderscores`

Funktionen in C

- In C können Funktionen
  - mehrere (oder keinen) Parameter übernehmen
  - einen einzigen oder keinen Rückgabewert liefern
  - Rückgabewert muss elementarer Datentyp sein
    - z.B. `double`, `int`
- Signatur hat folgenden Aufbau
  `<type of return value> <function name>(parameters)`
  - Funktion ohne Rückgabewert:
    - `<type of return value>` = `void`
  - Sonst: `<type of return value>` = Variablentyp
  - `parameters` = Liste der Übergabeparameter
    - getrennt durch Kommata
    - *vor* jedem Parameter Variablentyp angeben
    - *kein* Parameter $\Rightarrow$ leere Klammer `()`
- Rumpf ist ein Block
  - Rücksprung ins Hauptprogramm mit `return` oder bei Erreichen des Funktionsblock-Endes, falls Funktionstyp = `void`
  - Rücksprung ins Hauptprogramm mit `return output`, falls die Variable `output` zurückgegeben werden soll
  - Häufiger Fehler: `return` vergessen
    - Dann Rückgabewert zufällig!
    - $\Rightarrow$ Irgendwann Chaos (Laufzeitfehler!)

Variablen

- Alle Variablen, die im Funktionsblock deklariert werden, sind lokale Variablen
- Alle elementaren Variablen, die in Signatur deklariert werden, sind lokale Variablen
- Funktion bekommt Input-Parameter als Werte, ggf. Type Casting!

Call by Value

- Dass bei Funktionsaufrufen Input-Parameter in lokale Variablen kopiert werden, bezeichnet man als Call by Value
  - Es wird neuer Speicher angelegt, der Wert der Eingabe-Parameter wird in diese abgelegt

Beispiel: Quadrieren

```
 1 #include <stdio.h>
 2
 3 double square(double x) {
 4   return x*x;
 5 }
 6
 7 main() {
 8   double x = 0;
 9   printf("Input x = ");
10   scanf("%lf",&x);
11   printf("%f^2 = %f\n",x,square(x));
12 }
```

- Compiler muss Funktion vor Aufruf kennen
  - d.h. Funktion vor aufrufender Zeile definieren
- Ausführung startet immer bei `main()`
- Die Variable `x` in Funktion `square` und die Variable `x` in Funktion `main` sind verschieden!
- Eingabe von 5 ergibt als Output

```
Input x = 5
5^2 = 25.000000
```

Beispiel: Minimum zweier Zahlen

```
 1 #include <stdio.h>
 2
 3 double min(double x, double y) {
 4   if (x > y) {
 5     return y;
 6   }
 7   else {
 8     return x;
 9   }
10 }
11
12 main() {
13   double x = 0;
14   double y = 0;
15
16   printf("Input x = ");
17   scanf("%lf",&x);
18   printf("Input y = ");
19   scanf("%lf",&y);
20   printf("min(x,y) = %f\n",min(x,y));
21 }
```

- Eingabe von 10 und 2 ergibt als Output

  ```
  Input x = 10
  Input y = 2
  min(x,y) = 2.000000
  ```

- Programm erfüllt Aufgabenstellung der UE:
  - Funktion mit gewisser Funktionalität
  - aufrufendes Hauptprogramm mit
    - Daten einlesen
    - Funktion aufrufen
    - Ergebnis ausgeben

Deklaration von Funktionen

```
 1 #include <stdio.h>
 2
 3 double min(double, double);
 4
 5 main() {
 6   double x = 0;
 7   double y = 0;
 8
 9   printf("Input x = ");
10   scanf("%lf",&x);
11   printf("Input y = ");
12   scanf("%lf",&y);
13   printf("min(x,y) = %f\n",min(x,y));
14 }
15
16 double min(double x, double y) {
17   if (x > y) {
18     return y;
19   }
20   else {
21     return x;
22   }
23 }
```

- Bei vielen Funktionen wird Code unübersichtlich
  - Alle Funktionen oben deklarieren, vgl. Zeile 3
    - Compiler weiß dann, wie Funktion agiert
  - vollständiger Fkt.code folgt, vgl. Zeile 16-23
- Alternative Deklaration = Fkt.code ohne Rumpf
  - `double min(double x, double y);`
    vgl. Zeile 3, 16
- in Literatur: *Forward Declaration* und *Prototyp*

Call by Value

```
 1 #include <stdio.h>
 2
 3 void test(int x) {
 4   printf("a) x=%d\n", x);
 5   x = 43;
 6   printf("b) x=%d\n", x);
 7 }
 8
 9
10 main() {
11   int x = 12;
12   printf("c) x=%d\n", x);
13   test(x);
14   printf("d) x=%d\n", x);
15 }
```

- Output:

  c) x=12

  a) x=12

  b) x=43

  d) x=12

Call by Reference

- Bei anderen Programmiersprachen, wird nicht der Wert eines Input-Parameters an eine Funktion übergeben, sondern dessen Speicheradresse (Call by Reference)
  - d.h. Änderungen an der Variable sind auch außerhalb der Funktion sichtbar

```
 1 void test(int y) {
 2   printf("a) y=%d\n", y);
 3   y = 43;
 4   printf("b) y=%d\n", y);
 5 }
 6
 7
 8 main() {
 9   int x = 12;
10   printf("c) x=%d\n", x);
11   test(x);
12   printf("d) x=%d\n", x);
13 }
```

- Dieser Source-Code ist kein C-Code!
  - Ziel: nur was-wäre-wenn erklären!
- *Call by Reference* würde folgenden Output liefern:

  c) x=12

  a) y=12

  b) y=43

  d) x=43

Type Casting & Call by Value

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

- Type Casting von `int` auf `double` bei Übergabe
- Output:

  a) 0.000000

  b) 0.666667

Type Casting (Negativbeispiel!)

```
 1 #include <stdio.h>
 2
 3 int isEqual(int, int);
 4
 5 main() {
 6   double x = 4.1;
 7   double y = 4.9;
 8
 9   if (isEqual(x,y)) {
10       printf("x == y\n");
11   }
12   else {
13       printf("x != y\n");
14   }
15 }
16
17 int isEqual(int x, int y) {
18   if (x == y) {
19     return 1;
20   }
21   else {
22     return 0;
23   }
24 }
```

- Output:

  ```
  x == y
  ```

- Aber eigentlich $x \neq y$!
  - Implizites Type Casting von `double` auf `int` durch Abschneiden, denn Input-Parameter sind `int`
- Achtung mit Type Casting bei Funktionen!

Rekursion

- Was ist eine rekursive Funktion?
- Beispiel: Berechnung der Faktorielle
- Beispiel: Bisektionsverfahren

Rekursive Funktion

- Funktion ist rekursiv, wenn sie sich selber aufruft
- natürliches Konzept in der Mathematik:
  - $n! = n \cdot (n-1)!$
- d.h. Rückführung eines Problems auf einfacheres Problem derselben Art
- Achtung:
  - Rekursion darf nicht endlos sein
  - d.h. Abbruchbedingung für Rekursion ist wichtig
  - z.B. $1! = 1$
- häufig Schleifen statt Rekursion möglich (später!)
  - idR. Rekursion eleganter
  - idR. Schleifen effizienter

Beispiel: Faktorielle

```
 1 #include <stdio.h>
 2
 3 int factorial(int n) {
 4   if (n <= -1) {
 5     return -1;
 6   }
 7   else {
 8     if (n > 1) {
 9       return n*factorial(n-1);
10     }
11     else {
12       return 1;
13     }
14   }
15 }
16
17 main() {
18   int n = 0;
19   int nfac = 0;
20   printf("n=");
21   scanf("%d",&n);
22   nfac = factorial(n);
23   if (nfac <= 0) {
24     printf("Fehleingabe!\n");
25   }
26   else {
27     printf("%d!=%d\n",n,nfac);
28   }
29 }
```

Bisektionsverfahren

- Gegeben: stetiges $f : [a, b] \to \mathbb{R}$ mit $f(a)f(b) \leq 0$
  - Toleranz $\tau > 0$
- Tatsache: Zwischenwertsatz $\Rightarrow$ mind. eine Nst
  - denn $f(a)$ und $f(b)$ haben versch. Vorzeichen
- Gesucht: $x_0 \in [a, b]$ mit folgender Eigenschaft
  - $\exists \widetilde{x}_0 \in [a, b] \quad f(\widetilde{x}_0) = 0$ und $|x_0 - \widetilde{x}_0| \leq \tau$
- Bisektionsverfahren = iterierte Intervallhalbierung
  - Solange Intervallbreite $|b - a| > 2\,\tau$
    - Berechne Intervallmittelpunkt m und $f(m)$
    - Falls $f(a)f(m) \leq 0$, betrachte Intervall $[a, m]$
    - sonst betrachte halbiertes Intervall $[m, b]$
  - $x_0 := m$ ist schließlich gesuchte Approximation
- Verfahren basiert nur auf Zwischenwertsatz
- terminiert nach endlich vielen Schritten, da jeweils Intervall halbiert wird
- Konvergenz gegen Nst. $\widetilde{x}_0$ für $\tau = 0$.

Beispiel: Bisektionsverfahren

```
 1 #include <stdio.h>
 2
 3 double f(double x) {
 4   return x*x + 1/(2 + x) - 2;
 5 }
 6
 7 double bisection(double a, double b, double tol){
 8   double m = 0.5*(a+b);
 9   if ( b - a <= 2*tol ) {
10     return m;
11   }
12   else {
13     if ( f(a)*f(m) <= 0 ) {
14       return bisection(a,m,tol);
15     }
16     else {
17       return bisection(m,b,tol);
18     }
19   }
20 }
21
22 main() {
23   double a = 0;
24   double b = 10;
25   double tol = 1e-12;
26   double x = bisection(a,b,tol);
27
28   printf("Nullstelle x=%g\n",x);
29   printf("Funktionswert f(x)=%g\n",f(x));
30 }
```

- Platzhalter bei `printf` für `double`
  - `%f` als Fixpunktdarstellung `1.30278`
  - `%e` als Exponentialdarstellung `-5.64659e-13`
  - `%g` wähle geeignetere Darstellung `%f` bzw. `%e`
- siehe auch UNIX Manual Pages mittels Shell-Befehl
  - `man 3 printf`

Mathematische Funktionen

- Preprocessor, Compiler, Linker
- Object-Code
- Bibliotheken
- mathematische Funktionen

- `#define`
- `#include`

Preprocessor, Compiler & Linker

- Ein Compiler besteht aus mehreren Komponenten, die nacheinander abgearbeitet werden
- Preprocessor wird intern gestartet, *bevor* der Source-Code compiliert wird
  - Ersetzt Text im Code durch anderen Text
  - Preprocessor-Befehle beginnen *immer* mit # und enden *nie* mit Semikolon, z.B.
    - `#define text replacement`
      - in allen nachfolgenden Zeilen wird der Text `text` durch `replacement` ersetzt
      - zur Definition von Konstanten
      - **Konvention:** `GROSS_MIT_UNDERSCORES`
    - `#include file`
      - einfügen der Datei `file`
- Compiler übersetzt (Source-)Code in Object-Code
  - Object-Code = Maschinencode, bei dem symbolische Namen (z.B. Funktionsnamen) noch vorhanden sind
- Weiterer Object-Code wird zusätzlich eingebunden
  - z.B. Bibliotheken (= Sammlungen von Fktn)
- Linker ersetzt symbolische Namen im Object-Code durch Adressen und erstellt dadurch ein ausführbares Programm, sog. Executable

Bibliotheken & Header-Files

- (Funktions-) Bibliothek (z.B. math. Funktionen) besteht immer aus 2 Dateien
  - Object-Code
  - zugehöriges Header-File
- Im Header-File steht die Deklaration aller Fktn, die in der Bibliothek vorhanden sind
- Will man Bibliothek verwenden, muss man zugehöriges Header-File einbinden
  - `#include <header>` bindet Header-File `header` aus Standardverzeichnis `/usr/include/` ein,
    - z.B. `math.h` (Header-File zur math. Bib.)
  - `#include "datei"` bindet Datei aus *aktuellem* Verzeichnis ein (z.B. Downloads vom Internet)
  - idR. führt C-Compiler `#include <stdio.h>` von allein aus (in zugehöriger Bib. liegt z.B. `printf`)
- Ferner muss man den Object-Code der Bibliothek hinzulinken
  - Wo Object-Code der Bibliothek liegt, muss `gcc` mittels Option `-`ℓ (und `-`L) mitgeteilt werden
  - z.B. `gcc file.c -`ℓ`m` linkt math. Bibliothek
  - Standardbibliotheken automatisch gelinkt, z.B. `stdio` (also keine zusätzliche Option nötig)

Mathematische Funktionen

- Deklaration der math. Funktionen in `math.h`
  - Input & Output der Fktn sind vom Typ `double`
- Wenn diese Funktionen benötigt werden
  - im Source-Code: `#include <math.h>`
  - Compilieren des Source-Code mit *zusätzlicher* Linker-Option `-ℓm`, d.h.

    ```
    gcc file.c -o output -ℓm
    ```

    erzeugt Executable `output`
- Diese Bibliothek stellt u.a. zur Verfügung
  - Trigonometrische Funktionen
    - `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `cosh`, `sinh`, `tanh`
  - Exponentialfunktion und Logarithmus
    - `exp`, `log`, `log10`
  - Potenz- und Wurzelfunktion
    - `pow`, `sqrt` (wobei $x^y = \texttt{pow(x,y)}$)
    - **NICHT** x^3 mittels `pow`, **SONDERN** `x*x*x`
    - **NICHT** $(-1)^n$ mittels `pow`, **SONDERN** ...
  - Absolutbetrag `fabs`
  - Rundung auf ganze Zahlen: `round`, `floor`, `ceil`
- **ACHTUNG:** In der Bibliothek `stdlib.h` gibt es `abs`
  - `abs` ist Absolutbetrag für `int`
  - `fabs` ist Absolutbetrag für `double`

Elementares Beispiel

```
1 #include <stdio.h>
2 #include <math.h>
3
4 main() {
5   double x = 2.;
6   double y = sqrt(x);
7   printf("sqrt(%f)=%f\n",x,y);
8 }
```

- Precompiler-Befehle in 1, 2 ohne Semikolon
- Compilieren mit `gcc sqrt.c -ℓm`
- Vergisst man `-ℓm` $\Rightarrow$ Fehlermeldung des Linkers

  ```
  In function ‘main’
  sqrt.c:(.text+0x24): undefined reference to ‘sqrt’
  collect2: ld returned 1 exit status
  ```

- Output:

  ```
  sqrt(2.000000)=1.414214
  ```

Arrays (=Felder)

- Vektoren, Matrizen
- Operator `[...]`
- Matrix-Vektor-Multiplikation
- Lineare Gleichungssysteme

Vektoren

- Deklaration eines Vektors $x = (x_0, \ldots, x_{N-1}) \in \mathbb{R}^N$:
  - `double x[N];` $\mapsto$ `x` ist `double`-Vektor
- Zugriff auf Komponenten:
  - `x[j]` entspricht x_j
  - Jedes `x[j]` ist vom Typ `double`
- Analoge Deklaration für andere Datentypen
  - `int y[N];` $\mapsto$ `y` ist `int`-Vektor
- ACHTUNG mit der Indizierung der Komponenten
  - Indizes $0, \ldots, N-1$ in C
  - idR. Indizes $1, \ldots, N$ in Mathematik
- Initialisierung bei Deklaration möglich:
  - `double x[3] = {1,2,3};` dekl. $x = (1, 2, 3) \in \mathbb{R}^3$
- Vektor-Initialisierung nur bei Deklaration erlaubt
  - Später zwingend komponentenweises Schreiben!
    - d.h. `x[0] = 1; x[1] = 2; x[2] = 3;` ist OK!
    - `x = {1,2,3}` ist verboten!

Beispiel: Einlesen eines Vektors

```
 1 #include <stdio.h>
 2
 3 main() {
 4   double x[3] = {0,0,0};
 5
 6   printf("Einlesen eines Vektors x in R^3:\n");
 7   printf("x_0 = ");
 8   scanf("%lf",&x[0]);
 9   printf("x_1 = ");
10   scanf("%lf",&x[1]);
11   printf("x_2 = ");
12   scanf("%lf",&x[2]);
13
14   printf("x = (%f, %f, %f)\n",x[0],x[1],x[2]);
15 }
```

- Ausgabe `double` über `printf` mit Platzhalter `%f`
- Einlesen `double` über `scanf` mit Platzhalter `%lf`

Achtung: Statische Arrays

- Die Länge von Arrays ist statisch
  - nicht veränderbar während Programmablauf
  - $x \in \mathbb{R}^3$ kann nicht zu $x \in \mathbb{R}^5$ erweitert werden
- Programm kann nicht selbständig herausfinden, wie groß ein Array ist
  - d.h. Programm weiß bei Ablauf nicht, dass Vektor $x \in \mathbb{R}^3$ Länge 3 hat
  - Aufgabe des Programmierers!
- Achtung mit Indizierung!
  - Indizes laufen $0, \ldots, N-1$ in C
  - Prg kann nicht wissen, ob `x[j]` definiert ist
    - x muss mindestens Länge $j+1$ haben!
    - falsche Indizierung ist kein Syntaxfehler!
    - sondern bestenfalls Laufzeitfehler!
- Arrays dürfen nicht Output einer Funktion sein!
- Arrays werden mit Call by Reference übergeben!
- Dasselbe gilt für Matrizen bzw. allgemeine Arrays

Arrays & Call by Reference

```
 1 #include <stdio.h>
 2
 3 void callByReference(double y[3]) {
 4   printf("a) y = (%f, %f, %f)\n",y[0],y[1],y[2]);
 5   y[0] = 1;
 6   y[1] = 2;
 7   y[2] = 3;
 8   printf("b) y = (%f, %f, %f)\n",y[0],y[1],y[2]);
 9 }
10
11 main() {
12   double x[3] = {0,0,0};
13
14   printf("c) x = (%f, %f, %f)\n",x[0],x[1],x[2]);
15   callByReference(x);
16   printf("d) x = (%f, %f, %f)\n",x[0],x[1],x[2]);
17 }
```

- Output:

```
c) x = (0.000000, 0.000000, 0.000000)
a) y = (0.000000, 0.000000, 0.000000)
b) y = (1.000000, 2.000000, 3.000000)
d) x = (1.000000, 2.000000, 3.000000)
```

- Call by Reference bei Vektoren!
- Erklärung folgt später ($\rightarrow$ Pointer!)

Falsche Indizierung von Vektoren

```
 1 #include <stdio.h>
 2 #define WRONG 1000
 3
 4 main() {
 5   int x[3] = {0,1,2};
 6
 7   x[WRONG] = 43;
 8
 9   printf("x = (%d, %d, %d), x[%d] = %d\n",
10          x[0],x[1],x[2],WRONG,x[WRONG]);
11 }
```

- Zeile 2 definiert Konstante WRONG
  - **Konvention:** Konst. sind GROSS_MIT_UNDERSCORES
- Zeile 7, 9-10: Falscher Zugriff auf Vektor x
  - Trotzdem keine Fehlermeldung/Warnung vom Compiler!
  - Für korrekte Indizes sorgt der Programmierer!
- Output:

  ```
  x = (0, 1, 2), x[1000] = 43
  ```

- Für WRONG klein ⇒ i.a. keine Fehlermeldung
- Für WRONG groß genug ⇒ Laufzeitfehler

Matrizen

- Matrix $A \in \mathbb{R}^{M \times N}$ ist rechteckiges Schema

$$A = \begin{pmatrix} A_{00} & A_{01} & A_{02} & \dots & A_{0,N-1} \\ A_{10} & A_{11} & A_{12} & \dots & A_{1,N-1} \\ A_{20} & A_{21} & A_{22} & \dots & A_{2,N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ A_{M-1,0} & A_{M-1,1} & A_{M-1,2} & \dots & A_{M-1,N-1} \end{pmatrix}$$

  mit Koeffizienten $A_{jk} \in \mathbb{R}$

- zentrale math. Objekte der Linearen Algebra
- Deklaration einer Matrix $A \in \mathbb{R}^{M \times N}$:
  - `double A[M][N];` $\mapsto$ `A` ist `double`-Matrix
- Zugriff auf Komponenten:
  - `A[j][k]` entspricht A_{jk}
  - Jedes `A[j][k]` ist vom Typ `double`
- zeilenweise Initialisierung bei Deklaration möglich:
  - `double A[2][3] = {{1,2,3},{4,5,6}};`

    deklariert + initialisiert $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$
  - Nur bei gleichzeitiger Deklaration erlaubt, vgl. Vektoren

Allgemeine Arrays

- Vektor ist ein 1-dim. Array
- Matrix ist ein 2-dim. Array
- Ist `type` Datentyp, so deklariert
  - `type x[N];` einen Vektor der Länge N
  - Koeffizienten `x[j]` sind Variablen vom Typ `type`
- Ist `type` Datentyp, so deklariert
  - `type x[M][N];` eine $M \times N$ Matrix
  - `x[j]` ist Vektor vom Typ `type` (der Länge N)
  - Koeff. `x[j][k]` sind Variablen vom Typ `type`
- Auch mehr Indizes möglich
  - `type x[M][N][P];` deklariert 3-dim. Array
  - `x[j]` ist $N \times P$ Matrix vom Typ `type`
  - `x[j][k]` ist Vektor vom Typ `type` (der Länge P)
  - Koeff. `x[j][k][p]` sind Variablen vom Typ `type`
- etc.

Zählschleife `for`

- Mathematische Symbole $\sum_{j=1}^{n}$ und $\prod_{j=1}^{n}$
- Zählschleife

- `for`

Schleifen

- Schleifen führen einen oder mehrere Befehle wiederholt aus
- In Aufgabenstellung häufig Hinweise, wie
  - Vektoren & Matrizen
  - Laufvariablen $j = 1, \ldots, n$
  - Summen $\sum_{j=1}^{n} a_j := a_1 + a_2 + \cdots + a_n$
  - Produkte $\prod_{j=1}^{n} a_j := a_1 \cdot a_2 \cdots a_n$
  - Text wie z.B. *solange bis* oder *solange wie*
- Man unterscheidet
  - Zählschleifen (`for`): Wiederhole etwas eine gewisse Anzahl oft
  - Bedingungsschleifen: Wiederhole etwas bis eine Bedingung eintritt

Die for-Schleife

- for (init. ; cond. ; step-expr.) statement
- Ablauf einer for-Schleife
  - (1) Ausführen der Initialisierung init.
  - (2) Abbruch, falls Bedingung cond. nicht erfüllt
  - (3) Ausführen von statement
  - (4) Ausführen von step-expr.
  - (5) Sprung nach (2)
- statement ist
  - entweder eine logische Programmzeile
  - oder mehrere Prg.zeilen in Klammern {...}, sog. Block

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int j = 0;
 5
 6   for (j=5; j>0 ; j=j-1)
 7     printf("%d ",j);
 8
 9   printf("\n");
10 }
```

- j=j-1 in 6 ist Zuweisung, keine math. Gleichheit!
- Output:

  5 4 3 2 1

Vektor einlesen & ausgeben

```
 1 #include <stdio.h>
 2
 3 void scanVector(double input[], int dim) {
 4   int j = 0;
 5   for (j=0; j<dim; j=j+1) {
 6     input[j] = 0;
 7     printf("%d: ",j);
 8     scanf("%lf",&input[j]);
 9   }
10 }
11
12 void printVector(double output[], int dim) {
13   int j = 0;
14   for (j=0; j<dim; j=j+1) {
15     printf("%f ",output[j]);
16   }
17   printf("\n");
18 }
19
20 main() {
21   double x[5];
22   scanVector(x,5);
23   printVector(x,5);
24 }
```

- Funktionen müssen Länge von Arrays kennen!
  - d.h. zusätzlicher Input-Parameter nötig
- Arrays werden mit Call by Reference übergeben!

Namenskonvention (Wh)

- lokale Variablen sind `klein_mit_underscores`
- globale Variablen haben `auch_underscore_hinten_`
- Konstanten sind `GROSS_MIT_UNDERSCORES`
- Funktionen sind `erstesWortKleinKeineUnderscores`

Minimum eines Vektors

```
 1 #include <stdio.h>
 2 #define DIM 5
 3
 4 void scanVector(double input[], int dim) {
 5   int j = 0;
 6   for (j=0; j<dim; j=j+1) {
 7     input[j] = 0;
 8     printf("%d: ",j);
 9     scanf("%lf",&input[j]);
10   }
11 }
12
13 double min(double input[], int dim) {
14   int j = 0;
15   double minval = input[0];
16   for (j=1; j<dim; j=j+1) {
17     if (input[j]<minval) {
18       minval = input[j];
19     }
20   }
21   return minval;
22 }
23
24 main() {
25   double x[DIM];
26   scanVector(x,DIM);
27   printf("Minimum des Vektors ist %f\n", min(x,DIM));
28 }
```

- Hinweise zur Realisierung (vgl. UE)
  - Vektorlänge ist Konstante im Hauptprogramm
    - d.h. Länge im Hauptprg nicht veränderbar
  - aber Input-Parameter der Funktion `scanVector`
    - d.h. Funktion arbeitet für beliebige Länge

Beispiel: Summensymbol $\sum$

- Berechnung der Summe $S = \sum_{j=1}^{N} a_j$:
  - Abkürzung $\sum_{j=1}^{N} a_j := a_1 + a_2 + \cdots + a_N$
- Definiere theoretische Hilfsgröße $S_k = \sum_{j=1}^{k} a_k$
- Dann gilt
  - $S_1 = a_1$
  - $S_2 = S_1 + a_2$
  - $S_3 = S_2 + a_3$ etc.
- Realisierung also durch N-maliges Aufsummieren
  - ACHTUNG: Zuweisung, keine Gleichheit
    - $\mathrm{S} = \mathrm{a}_1$
    - $\mathrm{S} = \mathrm{S} + \mathrm{a}_2$
    - $\mathrm{S} = \mathrm{S} + \mathrm{a}_3$ etc.

Beispiel: Summensymbol $\sum$

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int j = 0;
 5   int n = 100;
 6
 7   int sum = 0;
 8
 9   for (j=1; j<=n; j=j+1) {
10     sum = sum+j;
11   }
12
13   printf("sum_{j=1}^{%d} j = %d\n",n,sum);
14 }
```

- Programm berechnet $\sum_{j=1}^{n} j$ für $n = 100$.
- Output:

  ```
  sum_{j=1}^{100} j = 5050
  ```

- ACHTUNG: Bei iterierter Summation nicht vergessen, Ergebnisvariable auf Null zu setzen vgl. Zeile 7
  - Anderenfalls: Falsches/Zufälliges Ergebnis!
- statt `sum = sum + j;`
  - Kurzschreibweise `sum += j;`

Beispiel: Produktsymbol $\prod$

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int j = 0;
 5   int n = 5;
 6
 7   int factorial = 1;
 8
 9   for (j=1; j<=n; j=j+1) {
10     factorial = factorial*j;
11   }
12
13   printf("%d! = %d\n",n,factorial);
14 }
```

- Prg berechnet Faktorielle $n! = \prod_{j=1}^{n} j$ für $n = 5$.
- Output:

  ```
  5! = 120
  ```

- ACHTUNG: Bei iteriertem Produkt nicht vergessen, Ergebnisvariable auf Eins zu setzen vgl. Zeile 7
  - Anderenfalls: Falsches/Zufälliges Ergebnis!
- statt `factorial = factorial*j;`
  - Kurzschreibweise `factorial *= j;`

Matrix-Vektor-Multiplikation

- Man darf `for`-Schleifen schachteln
  - Typisches Beispiel: Matrix-Vektor-Multiplikation
- Seien $A \in \mathbb{R}^{M \times N}$ Matrix, $x \in \mathbb{R}^N$ Vektor
- Def $b := Ax \in \mathbb{R}^M$ durch $b_j = \sum_{k=0}^{N-1} A_{jk} x_k$
  - Indizierung in C startet bei 0
- $Ax = b$ ist also Schreibweise für lineares GLS

$$
\begin{array}{ccccccc}
A_{00}x_0 & + & A_{01}x_1 & + \ldots + & A_{0,N-1}x_{N-1} & = & b_0 \\
A_{10}x_0 & + & A_{11}x_1 & + \ldots + & A_{1,N-1}x_{N-1} & = & b_1 \\
A_{20}x_0 & + & A_{21}x_1 & + \ldots + & A_{2,N-1}x_{N-1} & = & b_2 \\
\vdots & & \vdots & & \vdots & & \vdots \\
A_{M-1,0}x_0 & + & A_{M-1,1}x_1 & + \ldots + & A_{M-1,N-1}x_{N-1} & = & b_{M-1}
\end{array}
$$

- Implementierung
  - äußere Schleife über j, innere für Summe

```
for (j=0; j<M; j=j+1) {
  b[j] = 0;
  for (k=0; k<N; k=k+1) {
    b[j] = b[j] + A[j][k]*x[k];
  }
}
```

- ACHTUNG: Init. `b[j] = 0` nicht vergessen!

Matrix spaltenweise speichern

- math. Bibliotheken speichern Matrizen idR. spaltenweise als Vektor
  - $A \in \mathbb{R}^{M \times N}$, gespeichert als $a \in \mathbb{R}^{MN}$
  - $a = (A_{00}, A_{10}, ..., A_{M-1,0},\ A_{01}, A_{11}, \ldots,\ A_{M-1,N-1})$
  - A_{jk} entspricht also a_ℓ mit $\ell = j + k \cdot M$
- muss Matrix spaltenweise speichern, wenn ich solche Bibliotheken nutzen will
  - diese meistin Fortran programmiert
- Matrix-Vektor-Produkt
  - $b := Ax \in \mathbb{R}^M$, $b_j = \sum_{k=0}^{N-1} A_{jk} x_k$
  - mit `double A[M][N];`

```
for (j=0; j<M; j=j+1) {
  b[j] = 0;
  for (k=0; k<N; k=k+1) {
    b[j] = b[j] + A[j][k]*x[k];
  }
}
```

- Matrix-Vektor-Produkt (spaltenweise gespeichert)
  - mit `double A[M*N];`

```
for (j=0; j<M; j=j+1) {
  b[j] = 0;
  for (k=0; k<N; k=k+1) {
    b[j] = b[j] + A[j+k*M]*x[k];
  }
}
```

MinSort (= Selection Sort)

- Gegeben: Ein Vektor $x \in \mathbb{R}^n$
- Ziel: Sortiere x, sodass $x_1 \leq x_2 \leq \cdots \leq x_n$

- Algorithmus (1. Schritt)
  - suche Minimum x_k von $x_1, \ldots, x_n$
  - vertausche x_1 und x_k, d.h. x_1 ist kleinstes Elt.
- Algorithmus (2. Schritt)
  - suche Minimum x_k von $x_2, \ldots, x_n$
  - vertausche x_2 und x_k, d.h. x_2 zweit kleinstes Elt.
- nach $n-1$ Schritten ist x sortiert

- Hinweise zur Realisierung (vgl. UE)
  - Länge n ist Konstante im Hauptprogramm
    - d.h. n ist im Hauptprg nicht veränderbar
  - aber n ist Inputparameter der Funktion `minsort`
    - d.h. Funktion arbeitet für beliebige Länge

```
 1 #include <stdio.h>
 2 #define DIM 5
 3
 4 void scanVector(double input[], int dim) {
 5   int j = 0;
 6   for (j=0; j<dim; j=j+1) {
 7     input[j] = 0;
 8     printf("%d: ",j);
 9     scanf("%lf",&input[j]);
10   }
11 }
12
13 void printVector(double output[], int dim) {
14   int j = 0;
15   for (j=0; j<dim; j=j+1) {
16     printf("%f ",output[j]);
17   }
18   printf("\n");
19 }
20
21 void minsort(double vector[], int dim) {
22   int j, k, argmin;
23   double tmp;
24   for (j=0; j<dim-1; j=j+1) {
25     argmin = j;
26     for (k=j+1; k<dim; k=k+1) {
27       if (vector[argmin] > vector[k]) {
28         argmin = k;
29       }
30     }
31     if (argmin > j) {
32       tmp = vector[argmin];
33       vector[argmin] = vector[j];
34       vector[j] = tmp;
35     }
36   }
37 }
38
39 main() {
40   double x[DIM];
41   scanVector(x,DIM);
42   minsort(x,DIM);
43   printVector(x,DIM);
44 }
```

Aufwand

- Aufwand von Algorithmen
- Landau-Symbol $\mathcal{O}$

- `time.h`, `clock_t`, `clock()`

Aufwand eines Algorithmus

- wichtige Kenngröße für Algorithmen
  - um Algorithmen zu bewerten / vergleichen
- Aufwand = Anzahl benötigter Operationen
  - Zuweisungen
  - Vergleiche
  - arithmetische Operationen
- programmspezifische Operationen nicht gezählt
  - Deklarationen & Initialisierungen
  - Schleifen, Verzweigungen etc.
  - Zählvariablen
- Aufwand wird durch „einfaches“ Zählen ermittelt
- Konventionen zum Zählen nicht einheitlich
- in der Regel ist Aufwand für worst case interessant
  - d.h. maximaler Aufwand im schlechtesten Fall

Beispiel: Maximum suchen

```
 1 double maximum(double vector[], int n) {
 2   int i = 0;
 3   double max = 0;
 4
 5   max = vector[0];
 6   for (i=1; i<n; i=i+1) {
 7     if (vector[i] > max) {
 8       max = vector[i];
 9     }
10   }
11
12   return max;
13 }
```

- Beim Zählen wird jede Schleife zu einer Summe!
  - d.h. `for` in Zeile 6 ist $\sum_{i=1}^{n-1}$
- Aufwand:
  - 1 Zuweisung $\rightsquigarrow$ Zeile 5
  - In jedem Schritt der `for`-Schleife $\rightsquigarrow$ Zeile 6–10
    - 1 Vergleich $\rightsquigarrow$ Zeile 7
    - 1 Zuweisung (worst case!) $\rightsquigarrow$ Zeile 8
- insgesamt Operationen

$$1 + \sum_{i=1}^{n-1} 2 = 1 + 2(n-1) = 2n - 1$$

Landau-Symbol $\mathcal{O}$ (= groß-O)

- oft nur Größenordnung des Aufwands interessant
- Schreibweise $f = \mathcal{O}(g)$ für $x \to x_0$
  - heißt $\limsup_{x \to x_0} \left| \frac{f(x)}{g(x)} \right| < \infty$
  - d.h. $|f(x)| \leq C\,|g(x)|$ für $x \to x_0$.
  - d.h. f wächst höchstens so schnell wie g
- Beispiel: Maximum suchen
  - Aufwand $2n - 1 = \mathcal{O}(n)$ für $n \to \infty$
- häufig entfällt „für $x \to x_0$“
  - dann Grenzwert x_0 kanonisch z.B. $2n-1 = \mathcal{O}(n)$
- Sprechweise:
  - Algorithmus hat linearen Aufwand,
    falls Aufwand $\mathcal{O}(n)$ bei Problemgröße n
    - Maximumssuche hat linearen Aufwand
  - Algorithmus hat fastlinearen Aufwand,
    falls Aufwand $\mathcal{O}(n \log n)$ bei Problemgröße n
  - Algorithmus hat quadratischen Aufwand,
    falls Aufwand $\mathcal{O}(n^2)$ bei Problemgröße n
  - Algorithmus hat kubischen Aufwand,
    falls Aufwand $\mathcal{O}(n^3)$ bei Problemgröße n

Matrix-Vektor Multiplikation

```
 1 void MVM(double A[], double x[], double b[],
 2          int m, int n) {
 3   int i = 0;
 4   int j = 0;
 5
 6   for (j=0; j<m; j=j+1) {
 7     b[j] = 0;
 8     for (k=0; k<n; k=k+1) {
 9       b[j] = b[j] + A[j+k*m]*x[k];
10     }
11   }
12 }
```

- In jedem Schritt der j-Schleife $\rightsquigarrow$ Zeile 6–11
  - 1 Zuweisung $\rightsquigarrow$ Zeile 7
  - In jedem Schritt der k-Schleife $\rightsquigarrow$ Zeile 8–10
    - 1 Multiplikation $\rightsquigarrow$ Zeile 9
    - 1 Addition $\rightsquigarrow$ Zeile 9
    - 1 Zuweisung $\rightsquigarrow$ Zeile 9
- insgesamt Operationen

$$\sum_{j=0}^{m-1} \left(1 + \sum_{k=0}^{n-1} 3\right) = m + 3mn$$

- Aufwand $\mathcal{O}(mn)$
  - bzw. Aufwand $\mathcal{O}(n^2)$ für $m = n$
  - d.h. quadratischer Aufwand für $m = n$
- Indizierung wird i.a. nicht gezählt (Zeile 9)

Suchen im Vektor

```
 1 int search(int vector[], int value, int n) {
 2
 3   int j = 0;
 4
 5   for (j=0; j<n; j=j+1) {
 6     if (vector[j] == value) {
 7       return j;
 8     }
 9   }
10
11   return -1;
12 }
```

- Aufgabe:
  - Suche Index `j` mit `vector[j]` = `value`
  - Rückgabe `-1`, falls nicht ex.
- Achtung bei Gleichheit mit `double` (später!)
- in jedem Schritt der j-Schleife
  - 1 Vergleich
- Insgesamt Operationen

$$\sum_{j=0}^{n-1} 1 = n$$

- Aufwand $\mathcal{O}(n)$

Binäre Suche im sortierten Vektor

```
 1 int binsearch(int vector[], int value, int n) {
 2
 3   int j = 0;
 4   int start = 0;
 5   int end = n-1;
 6
 7   for ( ; start <= end ; ) {
 8     j = 0.5*(end+start);
 9     if (vector[j] == value) {
10       return j;
11     }
12     else if (vector[j] > value) {
13       end = j-1;
14     }
15     else {
16       start = j+1;
17     }
18   }
19
20   return -1;
21 }
```

- Voraussetzung: Vektor ist aufsteigend sortiert
- Modifiziere Idee des Bisektionsverfahrens
  - Betrachte halben Vektor, falls `vector[j]` $\neq$ `value`
- Frage: Wieviele Iterationen hat der Algorithmus?
  - jeder Schritt halbiert Vektor
  - Falls n Zweierpotenz, gilt $n/2^k = 1$
  - dann maximal $k = 1 + \log_2 n$ Schritte
    - je 2 Vergl. + 2 Zuw. + 1 Mult. + 2 Add./Subtr.
- Aufwand $\mathcal{O}(\log_2 n)$, d.h. logarithmischer Aufwand
  - sublinearer Aufwand $\mathcal{O}(\log_2 n) \ll \mathcal{O}(n)$

Minsort

```
 1 void minsort(int vector[], int n) {
 2   int j = 0;
 3   int k = 0;
 4   int argmin = 0;
 5   double tmp = 0;
 6
 7   for (j=0; j<n-1; j=j+1) {
 8     argmin = j;
 9     for (k=j+1; k<n; k=k+1) {
10       if (vector[argmin] > vector[k]) {
11         argmin = k;
12       }
13     }
14     if (argmin > j) {
15       tmp = vector[argmin];
16       vector[argmin] = vector[j];
17       vector[j] = tmp;
18     }
19   }
20 }
```

- In jedem Schritt der j-Schleife
  - 1 Zuweisung
  - In jedem Schritt der k-Schleife
    - 1 Vergleich
    - 1 Zuweisung (worst case!)
  - jeweils 1 Vergleich
  - jeweils 3 Zuweisungen (worst case!)
- quadratischer Aufwand $\mathcal{O}(n^2)$, weil:

$$\sum_{j=0}^{n-2}\left(5+\sum_{k=j+1}^{n-1}2\right)=5(n-1)+\sum_{j=0}^{n-2}\big((n-(j+1)\big)2$$

$$=5(n-1)+2\sum_{k=1}^{n-1}k=5(n-1)+2\,\frac{n(n-1)}{2}$$

Zeitmessung

- Wozu Zeitmesssung?
  - Vergleich von Algorithmen / Implementierungen
  - Überprüfen theoretischer Voraussagen
- theoretische Voraussagen
  - linearer Aufwand
    - Problemgröße $n \Rightarrow Cn$ Operationen
    - Problemgröße $kn \Rightarrow Ckn$ Operationen
    - d.h. $3\times$ Problemgröße $\Rightarrow 3\times$ Rechenzeit
  - quadratischer Aufwand
    - Problemgröße $n \Rightarrow Cn^2$ Operationen
    - Problemgröße $kn \Rightarrow Ck^2n^2$ Operationen
    - d.h. $3\times$ Problemgröße $\Rightarrow 9\times$ Rechenzeit
  - etc.
- BSP. Code braucht 1 Sekunde für $n = 1000$
  - Aufwand $\mathcal{O}(n) \Rightarrow$ 10 Sekunden für $n = 10000$
  - Aufwand $\mathcal{O}(n^2) \Rightarrow$ 100 Sekunden für $n = 10000$
  - Aufwand $\mathcal{O}(n^3) \Rightarrow$ 1000 Sek. für $n = 10000$
- Bibliothek `time.h`
  - Datentyp `clock_t` für Zeitvariablen
    für Ausgabe Typecast nicht vergessen!
  - Funktion `clock()` liefert Rechenzeit
    seit Programmbeginn
  - Konstante `CLOCKS_PER_SEC` zum Umrechnen:
    `Zeitvariable/CLOCKS_PER_SEC` liefert
    Angabe in Sekunden

Beispiel: Zeitmessung

```
 1 #include <stdio.h>
 2 #include <time.h>
 3
 4 #define DIM 1000
 5 #define VAL 500
 6
 7 int search(int vector[], int value, int n);
 8 int binsearch(int vector[], int value, int n);
 9 void minsort(int vector[], int n);
10
11 main() {
12   clock_t t1;
13   clock_t t2;
14   int i = 0;
15   int v[DIM];
16
17   for(i=0; i<DIM; i=i+1) {
18     printf("v[%d]=",i);
19     scanf("%d",&v[i]);
20   }
21
22   t1 = clock();
23   i = search(v,VAL,DIM);
24   t2 = clock();
25
26   printf("search: %f\n", (double)(t2-t1)/CLOCKS_PER_SEC);
27
28   t1 = clock();
29   minsort(v,DIM);
30   t2 = clock();
31
32   printf("minsort: %f\n", (double)(t2-t1)/CLOCKS_PER_SEC);
33
34   t1 = clock();
35   i = binsearch(v,VAL,DIM);
36   t2 = clock();
37
38   printf("binary search: %f\n",
39          (double)(t2-t1)/CLOCKS_PER_SEC);
40 }
```

Vergleich von Laufzeit

	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\log_2 n)$
n	`search`	`minsort`	`binsearch`
1.000	0.00	0.00	0.00
2.000	0.00	0.00	0.00
4.000	0.00	0.01	0.00
8.000	0.00	0.06	0.00
16.000	0.00	0.25	0.00
32.000	0.00	1.03	0.00
64.000	0.00	4.12	0.00
128.000	0.00	16.55	0.00
256.000	0.00	64.31	0.00
512.000	0.00	257.25	0.00
1.024.000	0.00	$\geq$ 18min	0.00
2.048.000	0.01	$\geq$ 72min	0.00
4.096.000	0.01	$\geq 4,5$h	0.00
8.192.000	0.02	$\geq$ 18h	0.00
16.384.000	0.04	$\geq$ 3d	0.00
32.768.000	0.08	$\geq$ 12d	0.00
65.536.000	0.15	$\geq 1,5$m	0.00
131.072.000	0.29	$\geq$ 6m	0.00
262.144.000	0.60	$\geq$ 2y	0.00
524.288.000	1.18	$\geq$ 8y	0.00
1.048.576.000	2.53	$\geq$ 32y	0.00

- log. Aufwand perfekt, denn $2^{30} > 1.048.576.000$
- auch linearer Aufwand liefert sehr gute Rechenzeit
- Quadratischer Aufwand für große n spürbar
- Fazit: Algorithmen sollen kleinen Aufwand haben
  - Ziel der numerischen Mathematik
  - nicht immer möglich

Bedingungsschleifen

- Bedingungsschleife
- kopfgesteuert vs. fußgesteuert
- Operatoren `++` und `--`

- `while`
- `do - while`

Die while-Schleife

- Formal: `while(condition) statement`
  - vgl. `binsearch`: `for( ; condition ; )`
- *Vor* jedem Durchlauf wird `condition` geprüft & Abbruch, falls nicht erfüllt
  - sog. kopfgesteuerte Schleife
- Eventuell also kein einziger Durchlauf!
- `statement` kann Block sein

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int counter = 5;
 5
 6   while (counter > 0) {
 7     printf("%d ",counter);
 8     counter = counter-1;
 9   }
10   printf("\n");
11 }
```

- Output:

```
5 4 3 2 1
```

Operatoren ++

- ++a und a++ sind arithmetisch äquivalent zu a=a+1
- Zusätzlich aber Auswertung von Variable a
- Präinkrement ++a
  - Erst erhöhen, dann auswerten
- Postinkrement a++
  - Erst auswerten, dann erhöhen

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int a = 0;
 5   int b = 43;
 6
 7   printf("1) a=%d, b=%d\n",a,b);
 8
 9   b = a++;
10   printf("2) a=%d, b=%d\n",a,b);
11
12   b = ++a;
13   printf("3) a=%d, b=%d\n",a,b);
14 }
```

- Output:

```
1) a=0, b=43
2) a=1, b=0
3) a=2, b=2
```

Operatoren ++ und --

- Analog zu `a++` und `++a` gibt es
  - Prädekrement `--`
    - Erst verringern, dann auswerten
  - Postdekrement `--`
    - Erst auswerten, dann verringern
- Beachte Unterschied in Bedingungsschleife!

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int counter = 5;
 5
 6   while (--counter>0) {
 7     printf("%d ",counter);
 8   }
 9   printf("\n");
10 }
```

- Output: `4 3 2 1` (für `--counter` in 6)
- Output: `4 3 2 1 0` (für `counter--` in 6)

Bisektionsverfahren (Wh)

- Gegeben: stetiges $f : [a, b] \to \mathbb{R}$ mit $f(a)f(b) \leq 0$
  - Toleranz $\tau > 0$
- Tatsache: Zwischenwertsatz $\Rightarrow$ mind. eine Nst
  - denn $f(a)$ und $f(b)$ haben versch. Vorzeichen
- Gesucht: $x_0 \in [a, b]$ mit folgender Eigenschaft
  - $\exists \widetilde{x}_0 \in [a, b] \quad f(\widetilde{x}_0) = 0$ und $|x_0 - \widetilde{x}_0| \leq \tau$
- Bisektionsverfahren = iterierte Intervallhalbierung
  - Solange Intervallbreite $|b - a| > 2\,\tau$
    - Berechne Intervallmittelpunkt m und $f(m)$
    - Falls $f(a)f(m) \leq 0$, betrachte Intervall $[a, m]$
    - sonst betrachte halbiertes Intervall $[m, b]$
  - $x_0 := m$ ist schließlich gesuchte Approximation
- Verfahren basiert nur auf Zwischenwertsatz
- terminiert nach endlich vielen Schritten, da jeweils Intervall halbiert wird
- Konvergenz gegen Nst. $\widetilde{x}_0$ für $\tau = 0$.

Bisektionsverfahren

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 double f(double x) {
 5   return x*x + exp(x) -2;
 6 }
 7
 8 double bisection(double a, double b, double tol){
 9   double fa = f(a);
10   double m = 0.5*(a+b);
11   double fm = 0;
12
13   while ( b - a > 2*tol ) {
14     m = 0.5*(a+b);
15     fm = f(m);
16     if ( fa*fm <= 0 ) {
17       b = m;
18     }
19     else {
20       a = m;
21       fa = fm;
22     }
23   }
24   return m;
25 }
26
27 main() {
28   double a = 0;
29   double b = 10;
30   double tol = 1e-12;
31   double x = bisection(a,b,tol);
32
33   printf("Nullstelle x=%g\n",x);
34   printf("Funktionswert f(x)=%g\n",f(x));
35 }
```

- Verwendung von Variablen `fa` und `fm` vermeidet doppelte Funktionsauswertung

Euklids Algorithmus

- Gegeben: zwei ganze Zahlen $a, b \in \mathbb{N}$
- Gesucht: größter gemeinsamer Teiler $ggT(a, b) \in \mathbb{N}$

- Euklidischer Algorithmus:
  - Falls $a = b$, gilt $ggT(a, b) = a$
  - Vertausche a und b, falls $a < b$
  - Dann gilt $ggT(a, b) = ggT(a - b, b)$, denn:
    - Sei g Teiler von a, b
    - d.h. $ga_0 = a$ und $gb_0 = b$ mit $a_0, b_0 \in \mathbb{N}$, $g \in \mathbb{N}$
    - also $g(a_0 - b_0) = a - b$ und $a_0 - b_0 \in \mathbb{N}$
    - d.h. g teilt b und $a - b$
    - d.h. $ggT(a, b) \leq ggT(a - b, b)$
    - analog $ggT(a - b, b) \leq ggT(a, b)$
  - Ersetze a durch $a - b$, wiederhole diese Schritte

- Erhalte $ggT(a, b)$ nach endlich vielen Schritten:
  - Falls $a \neq b$, wird also $n := \max\{a, b\} \in \mathbb{N}$ pro Schritt um mindestens 1 kleiner
  - Nach endl. Schritten gilt also nicht mehr $a \neq b$

Euklid-Algorithmus

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int a = 200;
 5   int b = 110;
 6   int tmp = 0;
 7
 8   printf("ggT(%d,%d)=",a,b);
 9
10   while (a != b) {
11     if ( a < b) {
12       tmp = a;
13       a = b;
14       b = tmp;
15     }
16     a = a-b;
17   }
18
19   printf("%d\n",a);
20 }
```

- berechnet ggT von $a, b \in \mathbb{N}$
- basiert auf $ggT(a, b) = ggT(a - b, b)$ für $a > b$
- Für $a = b$ gilt $ggT(a, b) = a = b$

- Output:

```
ggT(200,110)=10
```

Euklid-Algorithmus (verbessert)

- Kernstück des Euklid-Algorithmus

```
10   while (a != b) {
11     if ( a < b) {
12       tmp = a;
13       a = b;
14       b = tmp;
15     }
16     a = a-b;
17   }
```

- Erinnerung: $a\%b$ ist Divsionsrest von a/b
- Euklid-Algorithmus iteriert $a := a - b$ bis $a \leq b$
  - d.h. bis $a = a\%b$
  - falls fertig, gilt $a = 0$ und Ergebnis $b = ggT$

```
10   while (a != 0) {
11     if ( a < b) {
12       tmp = a;
13       a = b;
14       b = tmp;
15     }
16     a = a%b;
17   }
```

- Divisionsrest erfüllt immer $a\%b < b$
  - d.h. es wird immer vertauscht nach Rechnung
  - falls fertig, gilt $b = 0$ und Ergebnis $a = ggT$

```
10   while (b != 0) {
11     tmp = a%b;
12     a = b;
13     b = tmp;
14   }
```

Die do-while-Schleife

- Formal: `do statement while(condition)`
- *Nach* jedem Durchlauf wird `condition` geprüft & Abbruch, falls nicht erfüllt
  - sog. fußgesteuerte Schleife
- Also *mindestens ein* Durchlauf!
- `statement` kann Block sein

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int counter = 5;
 5
 6   do {
 7     printf("%d ",counter);
 8   }
 9   while (--counter>0);
10   printf("\n");
11 }
```

- Output:

  `5 4 3 2 1`

- `counter--` in 9 liefert Output: `5 4 3 2 1 0`

Ein weiteres Beispiel

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int x[2] = {0,1};
 5   int tmp = 0;
 6   int c = 0;
 7
 8   printf("c=");
 9   scanf("%d",&c);
10
11   printf("%d %d ",x[0],x[1]);
12
13   do {
14     tmp = x[0]+x[1];
15     x[0] = x[1];
16     x[1] = tmp;
17     printf("%d ",tmp);
18   }
19   while(tmp<c);
20
21   printf("\n");
22 }
```

- Fibonacci-Folge strebt gegen unendlich
  - $x_0 := 0$, $x_1 := 1$ und $x_{n+1} := x_{n-1} + x_n$ für $n \in \mathbb{N}$
- Ziel: Berechne erstes Folgenglied mit $x_n > c$ für gegebene Schranke $c \in \mathbb{N}$
- für Eingabe $c = 1000$ erhalte Output:

```
c=1000
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

break und continue

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int j = 0;
 5   int k = 0;
 6
 7   for (j=0; j<4; ++j) {
 8     if (j%2 == 0) {
 9       continue;
10     }
11     for (k=0; k < 10; ++k) {
12       printf("j=%d, k=%d\n",j,k);
13       if (k > 1) {
14         break;
15       }
16     }
17   }
18   printf("Ende: j=%d, k=%d\n",j,k);
19
20 }
```

- continue und break im statement von Schleifen
  - continue beendet aktuellen Durchlauf
  - break beendet die aktuelle Schleife
- Code ist schlecht programmiertes Beispiel!
- Output:

```
j=1, k=0
j=1, k=1
j=1, k=2
j=3, k=0
j=3, k=1
j=3, k=2
Ende: j=4, k=2
```

„solange wie“ vs. „solange bis“

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int a = 200;
 5   int b = 110;
 6   int tmp = 0;
 7
 8   printf("ggT(%d,%d)=",a,b);
 9
10   while (1) {
11     if (a == b) {
12       break;
13     }
14     else if ( a < b) {
15       tmp = a;
16       a = b;
17       b = tmp;
18     }
19     a = a-b;
20   }
21
22   printf("%d\n",b);
23 }
```

- `for` und `while` haben Laufbedingung `condition`
  - d.h. Schleife läuft, solange `condition` wahr
- Algorithmen haben idR. Abbruchbedingung `done`
  - d.h. Abbruch, falls `done` wahr
  - d.h. `condition` = Negation von `done`
- einfache Realisierung über Endlosschleife mit `break`
  - Bedingung in Zeile 10 ist immer wahr!
  - Abbruch erfolgt nur durch `break` in Zeile 12

Kommentarzeilen

- wozu Kommentarzeilen?

- `//`
- `/* ... */`

Kommentarzeilen

- werden vom Interpreter/Compiler ausgelassen
- nur für den Leser des Programmcodes
- notwendig, um eigene Programme auch später noch zu begreifen
  - deshalb brauchbar für Übung?
- notwendig, damit andere den Code verstehen
  - soziale Komponente der Übung?
- extrem brauchbar zum debuggen
  - Teile des Source-Code „auskommentieren“, sehen was passiert...
  - vor allem bei Fehlermeldungen des Parser
- Wichtige Regeln:
  - nie dt. Sonderzeichen verwenden
  - nicht zu viel und nicht zu wenig
  - zu Beginn des Source-Codes stets Autor & letzte Änderung kommentieren
    - vermeidet das Arbeiten an alten Versionen...

Kommentarzeilen in C

```
 1 #include <stdio.h>
 2
 3 main() {
 4   // printf("1 ");
 5   printf("2 ");
 6   /*
 7     printf("3");
 8     printf("4");
 9   */
10   printf("5");
11   printf("\n");
12 }
```

- Gibt in C zwei Typen von Kommentaren:
  - einzeiliger Kommentar
    - eingeleitet durch //, geht bis Zeilenende
    - z.B. Zeile 4
    - stammt eigentlich aus C++
  - mehrzeiliger Kommentar
    - alles zwischen /* (Anfang) und */ (Ende)
    - z.B. Zeile 6–9
    - darf nicht geschachtelt werden!
      - d.h. /* ... /* ... */ ... */ ist Syntaxfehler
- Vorschlag
  - Verwende // für echte Kommentare
  - Verwende /* ... */ zum Debuggen
- Output:

  2 5

Beispiel: Euklids Algorithmus

```
 1 // author: Dirk Praetorius
 2 // last modified: 19.03.2013
 3
 4 // Euklids Algorithmus zur Berechnung des ggT
 5 // basiert auf ggT(a,b) = ggT(a-b,b) fuer a>b
 6 // und ggT(a,b) = ggT(b,a)
 7
 8 int euklid(int a, int b) {
 9   int tmp = 0;
10
11   // iteriert Uebergang ggT(a,b) = ggT(a-b,b),
12   // realisiert mittels Divisionsrest, bis
13   // b = 0. Dann war a==b, also ggT = a
14
15   while (b != 0) {
16     tmp = b;
17     b = a%b;
18     a = tmp;
19   }
20
21   return a;
22 }
```

Naive Fehlerkontrolle

- Vermeidung von Laufzeitfehlern!
- bewusster Fehlerabbruch

- `gcc -c`
- `gcc -c -Wall`
- `assert`
- `#include <assert.h>`

Motivation

- Fakt ist: alle Programmierer machen Fehler
  - Code läuft beim ersten Mal nie richtig
- Großteil der Entwicklungszeit geht in Fehlersuche
- „Profis“ unterscheiden sich von „Anfängern“ im Wesentlichen durch effizientere Fehlersuche
- Syntax-Fehler sind leicht einzugrenzen
  - es steht Zeilennummer dabei (Compiler!)
  - Tipp: Verwende während des Programmierens zum Syntax-Test regelmäßig (Details später!)
    - `gcc -c name.c` nur Objekt-Code
    - `gcc -c -Wall name.c` alle Warnungen
- Laufzeitfehler sind viel schwieriger zu finden
  - Programm läuft, tut aber nicht das Richtige
  - manchmal fällt der Fehler ewig nicht auf

    $\Rightarrow$ sehr schlecht

Fehler vermeiden!

- Programmier-Konventionen beachten
  - z.B. bei Namen für Variablen, Funktionen etc.
- Kommentarzeilen dort, wo im Code etwas passiert
  - z.B. Verzweigung mit nicht offensichtlicher Bdg.
  - z.B. Funktionen (Zweck, Input, Output)
- jede Funktion hat nur eine Funktionalität
  - jede Funktion einzeln & sofort testen
  - Wenn später Funktion verwendet wird, kann ein etwaiger Fehler dort nicht mehr sein!
  - d.h. kann Fehler im Prg schneller lokalisieren!
- jede Funktionalität hat eigene Funktion
  - Prg. in überschaubare Funktionen zerlegen!
- nicht alles auf einmal programmieren!
  - **Achtung:** Häufiger Anfängerfehler!
- Möglichst viele Fehler bewusst abfangen!
  - Funktions-Input auf Konsistenz prüfen!
    - Fehler-Abbruch, falls inkonsistent!
  - garantieren, dass Funktions-Output zulässig!

Bibliothek assert.h

```
 1 #include <stdio.h>
 2 #include <assert.h>
 3
 4 void test(int x, int y) {
 5   assert(x<y);
 6   printf("It holds x < y\n");
 7 }
 8
 9 main() {
10   int x = 0;
11   int y = 0;
12
13   printf("x = ");
14   scanf("%d",&x);
15
16   printf("y = ");
17   scanf("%d",&y);
18
19   test(x,y);
20 }
```

- **Ziel:** Sofortabbruch mit Fehlermeldung, sobald Funktion merkt, dass Input / Output unzulässig
- `#include <assert.h>`
  - `assert(condition);` liefert Fehlerabbruch, falls `condition` falsch
  - mit Ausgabe der Zeilennummer im Source-Code
- **Input:**

  ```
  x = 2
  y = 1
  ```

- **Output:**

  ```
  Assertion failed: (x<y), function test,
  file assert.c, line 5.
  ```

Beispiel: Euklids Algorithmus

```
 1 // author: Dirk Praetorius
 2 // last modified: 30.03.2017
 3
 4 // Euklids Algorithmus zur Berechnung des ggT
 5 // basiert auf ggT(a,b) = ggT(a-b,b) fuer a>b
 6 // und ggT(a,b) = ggT(b,a)
 7
 8 int euklid(int a, int b) {
 9   assert(a>0);
10   assert(b>0);
11   int tmp = 0;
12
13   // iteriert Uebergang ggT(a,b) = ggT(b,a-b),
14   // realisiert mittels Divisionsrest, bis
15   // b = 0. Dann war a==b, also ggT = a
16
17   while (b != 0) {
18     tmp = a%b;
19     a = b;
20     b = tmp;
21   }
22
23   return a;
24 }
```

- assert stellt sicher, dass Input zulässig
  - d.h. $a, b \in \mathbb{N}$ ist notwendig!

Testen

- Motivation
- Qualitätssicherung
- Arten von Tests

Motivation

- Ariane 5 Explosion ('96)
  - Konversion `double` → `int`
  - Schaden ca. 500 Mio. Dollar
- Patriot Missile Fehler, Golfkrieg ('91)
  - Zeitmessung falsch berechnet & Rundungsfehler
- „kleine BUGs, große GAUs"
  - `http://www5.in.tum.de/~huckle/bugs.html`

Qualitätssicherung

- Software entsteht durch menschliche Hand
- Fehler zu machen, ist menschlich!
- Software wird deshalb Fehler enthalten
- Ziel: (Laufzeit-) Fehler finden vor großem Schaden
- Je später Fehler entdeckt werden, desto aufwändiger ist ihre Behebung!
- Schon beim Implementieren auf Qualität achten
  - siehe oben: Fehler vermeiden!
- wünschenswert: je 1/3 Zeit für
  - Programmieren
  - Testen
  - Dokumentieren
- wünschenswert: Dokumentation der Tests!
  - damit reproduzierbar
- In der Praxis meist viel Programmieren, wenig Testen, noch weniger Dokumentieren ;-)

Testen

- Testen ist der Prozess, ein Programm mit der Absicht auszuführen, Fehler zu finden!
  - Glenford Myers: Art of Sofware Design (1979)
- Test ist der Vergleich des Verhaltens eines Prg (Ist) mit dem erwarteten Verhalten eines Systems (Soll)
- Es ist praktisch nicht möglich, alle Prg.funktionen und alle möglichen Werte in den Eingabedaten in allen Kombinationen zu testen.
  - d.h. Tests sind idR. unvollständig!
- Probleme beim unvollständigen Testen
  - Tests erlauben nur das Auffinden von Fehlern
  - Tests können Korrektheit nicht beweisen
  - Fehlerursache ist durch Soll-Ist-Vergleich nicht zwangsläufig klar
  - Testfälle können selbst fehlerhaft sein!
- Vorteile beim unvollständigen Testen
  - Zeitaufwand vertretbar
  - Tests beziehen sich idR. auf „realistischen Input“
  - Tests sind idR. reproduzierbar

Arten von Tests

- strukturelle Tests (für jede Funktion)
  - Werden alle Anweisungen ausgeführt oder gibt es toten Code?
  - Treten Fehler auf, wenn `if ... else` mit *wahr* / *falsch* durchlaufen werden?
  - Treten Fehler auf, wenn `if ... else` in allen Kombinationen mit *wahr*/*falsch* durchlaufen?
- funktionale Tests (für jede Fkt. und Programm)
  - Tut jede Funktion mit zulässigen Parametern das Richtige? (d.h. Ergebnis korrekt?)
  - Werden unzulässige Parameter erkannt?
  - Tut das Programm (bzw. Teilabschnitte) das Richtige? (d.h. Ergebnis korrekt?)
  - Werden Grenzfälle / Sonderfälle korrekt erkannt und liefern das Richtige?
  - Was passiert bei Fehleingaben, d.h. bei Fehler des Benutzers?

Wie testen?

- **Ziel:** Tut Funktion / Programm das Richtige?
- funktionale Tests brauchen Testfälle
  - mit bekanntem Ergebnis / Output!
- Was sind generische Fälle / Parameter?
  - Bei welchen Fällen treten Verzweigungen auf?
  - Möglichst viele Verzweigungen abdecken!
- Welche Fälle sind kritisch?
  - Wo können aufgrund Rechenfehlern oder Rechenungenauigkeiten andere Ergebnisse auftreten?
- Ab jetzt in der UE stets: Wie wurde getestet?
  - allerdings nur inhaltlich
  - d.h. ohne Fehleingaben des Nutzers

Typische Fehler beim Testen

- zu wenig Zeit für ausführliche Tests
- zu spät mit Testen begonnen, z.B.
  - erst nach Fertigstellung der Software
  - sodass einzelne Funktionen nicht mehr getestet
- das „normale“ Verhalten der Nutzer vergessen
- Fehler-Analyse geht nicht tief genug
  - Folgen von Fehlern unterschätzt
  - nach Code-Korrektur alle(!) Tests wiederholen
    - deshalb Dokumentation der Tests!

Pointer

- Variable vs. Pointer
- Dereferenzieren
- Address-of Operator &
- Dereference Operator *
- Call by Reference

Variablen

- Variable = symbolischer Name für Speicherbereich
  - + Information, wie Speicherbereich interpretiert werden muss (Datentyp laut Deklaration)
- Compiler übersetzt Namen in Referenz auf Speicherbereich und merkt sich, wie dieser interpretiert werden muss

Pointer

- Pointer = Variable, die Adresse eines Speicherbereichs enthält
- Dereferenzieren = Zugriff auf den Inhalt eines Speicherbereichs mittels Pointer
  - Beim Dereferenzieren muss Compiler wissen, welcher Var.typ im gegebenen Speicherbereich liegt, d.h. wie Speicherbereich interpretiert werden muss

Pointer in C

- Pointer & Variablen sind in C eng verknüpft:
  - `var` Variable ⇒ `&var` zugehöriger Pointer
  - `ptr` Pointer ⇒ `*ptr` zugehörige Variable
  - insbesondere `*&var` = `var` sowie `&*ptr` = `ptr`
- Bei Deklaration muss Typ des Pointers angegeben werden, da `*ptr` eine Variable sein soll!
  - `int* ptr;` deklariert `ptr` als Pointer auf `int`
- Wie üblich gleichzeitige Initialisierung möglich
  - `int var;` deklariert Variable `var` vom Typ `int`
  - `int* ptr = &var;` deklariert `ptr` und weist Speicheradresse der Variable `var` zu
    - Bei solchen Zuweisungen muss der Typ von Pointer und Variable passen, sonst passiert Unglück!
      - I.a. gibt Compiler eine Warnung aus, z.B. `incompatible pointer type`
- Analog für andere Datentypen, z.B. `double`

Ein elementares Beispiel

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int var = 1;
 5   int* ptr = &var;
 6
 7   printf("a) var = %d, *ptr = %d\n",var,*ptr);
 8
 9   var = 2;
10   printf("b) var = %d, *ptr = %d\n",var,*ptr);
11
12   *ptr = 3;
13   printf("c) var = %d, *ptr = %d\n",var,*ptr);
14
15   var = 47;
16   printf("d) *(&var) = %d,",*(&var));
17   printf("*&var = %d\n",*&var);
18
19   printf("e) &var = %p\n", &var);
20 }
```

- `%p` Platzhalter für `printf` für Adresse
- Output:

```
a) var = 1, *ptr = 1
b) var = 2, *ptr = 2
c) var = 3, *ptr = 3
d) *(&var) = 47,*&var = 47
e) &var = 0x7fff518baba8
```

Call by Reference in C

- Elementare Datentypen werden in C mit *Call by Value* an Funktionen übergeben
  - z.B. `int`, `double`, Pointer
- *Call by Reference* ist über Pointer realisierbar:

```
 1 #include <stdio.h>
 2
 3 void test(int* y) {
 4   printf("a) *y=%d\n", *y);
 5   *y = 43;
 6   printf("b) *y=%d\n", *y);
 7 }
 8
 9
10 main() {
11   int x = 12;
12   printf("c) x=%d\n", x);
13   test(&x);
14   printf("d) x=%d\n", x);
15 }
```

- Output:

  c) x=12

  a) *y=12

  b) *y=43

  d) x=43

Begrifflichkeiten

- Call by Value
  - Funktionen erhalten Werte der Input-Parameter und speichern diese in lokalen Variablen
  - Änderungen an den Input-Parameter wirken sich nicht außerhalb der Funktion aus
- Call by Reference
  - Funktionen erhalten Variablen als Input ggf. unter lokal neuem Namen
  - Änderungen an den Input-Parametern wirken sich außerhalb der Funktion aus

Wiederholung

- Standard in C ist Call by Value
- Kann Call by Reference mittels Pointern realisieren
- Vektoren werden mit Call by Reference übergeben

Warum Call by Reference?

- Funktionen haben in C maximal 1 Rückgabewert
- Falls Fkt mehrere Rückgabewerte haben soll ...

Ein Beispiel

```
 1 #include <stdio.h>
 2 #include <assert.h>
 3 #define DIM 5
 4
 5 void scanVector(double input[], int dim) {
 6   assert(dim > 0);
 7   int j = 0;
 8   for (j=0; j<dim; ++j) {
 9     input[j] = 0;
10     printf("%d: ",j);
11     scanf("%lf",&input[j]);
12   }
13 }
14
15 void determineMinMax(double vector[],int dim,
16                      double* min, double* max) {
17   int j = 0;
18   assert(dim > 0);
19
20   *max = vector[0];
21   *min = vector[0];
22   for (j=1; j<dim; ++j) {
23     if (vector[j] < *min) {
24       *min = vector[j];
25     }
26     else if (vector[j] > *max) {
27       *max = vector[j];
28     }
29   }
30 }
31
32 main() {
33   double x[DIM];
34   double max = 0;
35   double min = 0;
36   scanVector(x,DIM);
37   determineMinMax(x,DIM, &min, &max);
38   printf("min(x) = %f\n",min);
39   printf("max(x) = %f\n",max);
40 }
```

- `determineMinMax` liefert mittels Call by Reference Minimum und Maximum eines Vektors

Anmerkungen zu Pointern

- Standard-Notation zur Deklaration ist anders als meine Sichtweise:
  - `int *pointer` deklariert Pointer auf `int`
- Von den C-Erfindern wurden Pointer *nicht* als Variablen verstanden
- Für das Verständnis scheint mir aber „variable“ Sichtweise einfacher
- Leerzeichen wird vom Compiler ignoriert:
  - `int* pointer`, `int *pointer`, `int*pointer`
- `*` wird nur auf den folgenden Namen bezogen
- ACHTUNG bei Deklaration von Listen:
  - `int* pointer, var;` deklariert Pointer auf `int` und Variable vom Typ `int`
  - `int *pointer1, *pointer2;` deklariert zwei Pointer auf `int`
- ALSO Listen von Pointern vermeiden!
  - auch zwecks Lesbarkeit!

Pointer & Arrays

- Deklaration `int array[N];` generiert intern Pointer `array` vom Typ `int*`
- Dekl. `int array[];` äquivalent zu `int* array;`

Funktionspointer

- Deklaration
- Alles ist Pointer!

Funktionspointer

- Funktionsaufruf ist Sprung an eine Adresse
  - Pointer speichern Adressen
  - kann daher Fkt-Aufruf mit Pointer realisieren
- Deklaration eines Funktionspointers:
  - `<return value> (*pointer)(<input>)`; deklariert Pointer `pointer` für Funktionen mit Parametern `<input>` und Ergebnis vom Typ `<return value>`
- Bei Zuweisung müssen Pointer `pointer` und Funktion denselben Aufbau haben
  - gleicher Return-Value
  - gleiche Input-Parameter-Liste
- Aufruf einer Funktion über Pointer wie bei normalem Funktionsaufruf!

Elementares Beispiel

```
 1 #include <stdio.h>
 2
 3 void output1(char* string) {
 4   printf("*%s*\n",string);
 5 }
 6
 7 void output2(char* string) {
 8   printf("#%s#\n",string);
 9 }
10
11 main() {
12   char string[] = "Hello World";
13   void (*output)(char* string);
14
15   output = output1;
16   output(string);
17
18   output = output2;
19   output(string);
20 }
```

- Deklaration eines Funktionspointers in Zeile 13
- Zuweisung auf Fkt.pointer in Zeile 15 + 18
- Fkt.aufruf über Pointer in Zeile 16 + 19
- **Output:**

  ```
  *Hello World*
  #Hello World#
  ```

Bisektionsverfahren

```
 1 #include <stdio.h>
 2 #include <assert.h>
 3 #include <math.h>
 4
 5 double bisection(double (*fct)(double x),
 6                  double a, double b, double tol) {
 7   double m = 0;
 8   double fa = 0;
 9   double fm = 0;
10
11   assert(a < b);
12   fa = fct(a);
13   assert(fa*fct(b) <= 0);
14
15   while ( b-a > tol) {
16     m = (a+b)/2;
17     fm = fct(m);
18     if ( fa*fm <= 0 ) {
19       b = m;
20     }
21     else {
22       a = m;
23       fa = fm;
24     }
25   }
26   return m;
27 }
28
29 double f(double x) {
30   return x*x+exp(x)-2;
31 }
32
33 main() {
34   double a = 0;
35   double b = 10;
36   double tol = 1e-12;
37
38   double x = bisection(f,a,b,tol);
39   printf("Nullstelle x=%1.15e\n",x);
40 }
```

- Approximation der Nullstelle von $f(x) = x^2 + e^x - 2$

Elementare Datentypen

- Arrays & Pointer
- `sizeof`

Elementare Datentypen

C kennt folgende elementare Datentypen:

- Datentyp für Zeichen (z.B. Buchstaben)
  - `char`
- Datentypen für Ganzzahlen:
  - `short`
  - `int`
  - `long`
- Datentypen für Gleitkommazahlen:
  - `float`
  - `double`
  - `long double`
- Alle Pointer gelten als elementare Datentypen

Bemerkungen:

- Deklaration und Gebrauch wie bisher
- Man kann Arrays & Pointer bilden
- Für UE nur `char`, `int`, `double` & Pointer
- Genaueres zu den Typen später!

Der Befehl sizeof

```
 1 #include <stdio.h>
 2
 3 void printSizeOf(double vector[]) {
 4   printf("sizeof(vector) = %d\n",sizeof(vector));
 5 }
 6
 7 main() {
 8   int var = 43;
 9   double array[12];
10   double* ptr = array;
11
12   printf("sizeof(var) = %d\n",sizeof(var));
13   printf("sizeof(double) = %d\n",sizeof(double));
14   printf("sizeof(array) = %d\n",sizeof(array));
15   printf("sizeof(ptr) = %d\n",sizeof(ptr));
16   printSizeOf(array);
17 }
```

- Ist `var` eine Variable eines elementaren Datentyps, gibt `sizeof(var)` die Größe der Var. in Bytes zurück
- Ist `type` ein Datentyp, so gibt `sizeof(type)` die Größe einer Variable dieses Typs in Bytes zurück
- Ist `array` ein *lokales statisches Array*, so gibt `sizeof(array)` die Größe des Arrays in Bytes zurück
- Intern sind `ptr` und `array` zwei `double` Pointer und enthalten (= zeigen auf) dieselbe Speicheradresse!
- Output:

```
sizeof(var) = 4
sizeof(double) = 8
sizeof(array) = 96
sizeof(ptr) = 8
sizeof(vector) = 8
```

Funktionen

- Elementare Datentypen werden an Funktionen mit Call by Value übergeben
- Return Value einer Funktion darf nur `void` oder ein elementarer Datentyp sein

Arrays

- Streng genommen, gibt es in C keine Arrays!
  - Deklaration `int array[N];`
    - legt Pointer `array` vom Typ `int*` an
    - organisiert ab der Adresse `array` Speicher, um `N`-mal einen `int` zu speichern
    - d.h. `array` enthält Adresse von `array[0]`
  - Da Pointer als elementare Datentypen mittels Call by Value übergeben werden, werden Arrays augenscheinlich mit Call by Reference übergeben

Laufzeitfehler!

```
 1 #include <stdio.h>
 2 #include <assert.h>
 3
 4 double* scanVector(int length) {
 5   assert(length > 0);
 6   double vector[length];
 7   int j = 0;
 8   for (j=0; j<length; ++j) {
 9     vector[j] = 0;
10     printf("vector[%d] = ",j);
11     scanf("%lf",&vector[j]);
12   }
13   return vector;
14 }
15
16 main() {
17   double* x;
18   int j = 0;
19   int dim = 0;
20
21   printf("dim = ");
22   scanf("%d",&dim);
23
24   x = scanVector(dim);
25
26   for (j=0; j<dim; ++j) {
27     printf("x[%d] = %f\n",j,x[j]);
28   }
29 }
```

- Syntax des Programms ist OK
- Problem: Speicher zu x mit Blockende 14 aufgelöst
  - d.h. Pointer aus 6/13 zeigt auf Irgendwas
- Abhilfe: Call by Reference (vorher!) oder händische Speicherverwaltung (gleich!)

Dynamische Vektoren

- statische & dynamische Vektoren
- Vektoren & Pointer
- dynamische Speicherverwaltung

- `stdlib.h`
- `NULL`
- `malloc`, `realloc`, `free`

- `#ifndef ... #endif`

Statische Vektoren

- `double array[N];` deklariert statischen Vektor `array` der Länge `N` mit `double`-Komponenten
  - Indizierung `array[j]` mit $0 \leq j \leq N-1$
  - `array` ist intern vom Typ `double*`
    - enthält Adr. von `array[0]`, sog. *Base Pointer*
  - Länge `N` kann während Programmablauf nicht verändert werden

- Funktionen können Länge `N` nicht herausfinden
  - Länge `N` als Input-Parameter übergeben

Speicher allokieren

- Nun händische Speicherverwaltung von Arrays
  - dadurch Vektoren dynamischer Länge möglich

- Einbinden der Standard-Bibl: `#include <stdlib.h>`
  - wichtige Befehle `malloc`, `free`, `realloc`

- `pointer = malloc(N*sizeof(type));`
  - allokiert Speicher für Vektor der Länge `N` mit Komponenten vom Typ `type`
    - `malloc` kriegt Angabe in Bytes → `sizeof`
  - `pointer` muss vom Typ `type*` sein
    - Base Pointer `pointer` bekommt Adresse der ersten Komponente `pointer[0]`
  - `pointer` und `N` muss sich Prg merken!

- **Häufiger Laufzeitfehler:** `sizeof` vergessen!
- **Achtung:** Allokierter Speicher ist uninitialisiert!

- **Konvention:** Pointer ohne Speicher bekommen den Wert `NULL` zugewiesen
  - führt sofort auf Speicherzugriffsfehler bei Zugriff

- `malloc` liefert `NULL`, falls Fehler bei Allokation
  - d.h. Speicher konnte nicht allokiert werden

Speicher freigeben

- `free(pointer)`
  - gibt Speicher eines dyn. Vektors frei
  - `pointer` muss Output von `malloc` sein

- **Achtung:** Speicher wird freigegeben, aber `pointer` existiert weiter
  - Erneuter Zugriff führt (irgendwann) auf Laufzeitfehler

- **Achtung:** Speicher freigeben, nicht vergessen!
  - und Pointer auf `NULL` setzen!

Beispiel

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <assert.h>
 4
 5 double* scanVector(int length) {
 6   int j = 0;
 7   double* vector = NULL;
 8   assert(length > 0);
 9
10   vector = malloc(length*sizeof(double));
11   assert(vector != NULL);
12   for (j=0; j<length; ++j) {
13     vector[j] = 0;
14     printf("vector[%d] = ",j);
15     scanf("%lf",&vector[j]);
16   }
17   return vector;
18 }
19
20 void printVector(double* vector, int length) {
21   int j = 0;
22   assert(vector != NULL);
23   assert(length > 0);
24
25   for (j=0; j<length; ++j) {
26     printf("vector[%d] = %f\n",j,vector[j]);
27   }
28 }
29
30 main() {
31   double* x = NULL;
32   int dim = 0;
33
34   printf("dim = ");
35   scanf("%d",&dim);
36
37   x = scanVector(dim);
38   printVector(x,dim);
39
40   free(x);
41   x = NULL;
42 }
```

Dynamische Vektoren

- `pointer = realloc(pointer,Nnew*sizeof(type))`
  - verändert Speicherallokation
    - zusätzliche Allokation für `Nnew` $>$ `N`
    - Speicherbereich kürzen für `Nnew` $<$ `N`
  - Alter Inhalt bleibt (soweit möglich) erhalten
  - Rückgabe `NULL` bei Fehler

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <assert.h>
 4
 5 main() {
 6   int N = 5;
 7   int Nnew = 10;
 8   int j = 0;
 9
10   int* array = malloc(N*sizeof(int));
11   assert(array != NULL);
12   for (j=0; j<N; ++j){
13     array[j] = j;
14   }
15
16   array = realloc(array,Nnew*sizeof(int));
17   assert(array != NULL);
18   for (j=N; j<Nnew; ++j){
19     array[j] = 10*j;
20   }
21
22   for (j=0; j<Nnew; ++j){
23     printf("%d ",array[j]);
24   }
25   printf("\n");
26   free(array);
27   array = NULL;
28 }
```

- Output:

```
0 1 2 3 4 50 60 70 80 90
```

Bemerkungen

- Base Pointer (= Output von `malloc` bzw. `realloc`) merken & nicht verändern
  - notwendig für fehlerfreies `free` und `realloc`
- bei `malloc` und `realloc` nicht `sizeof` vergessen
  - Typ des Base Pointers muss zum `sizeof` passen!
- Output von `malloc` und `realloc` auf `NULL` checken
  - sicherstellen, dass Speicher allokiert wurde!
- allokierter Speicherbereich ist stets uninitialisiert
  - nach Allokation stets initialisieren
- Länge des dynamischen Arrays merken
  - kann Programm nicht herausfinden!
- Nicht mehr benötigten Speicher freigeben
  - insb. vor Blockende `}`, da dann Base Pointer weg
- Pointer auf `NULL` setzen, wenn ohne Speicher
  - Fehlermeldung, falls Programm "aus Versehen" auf Komponente `array[j]` zugreift
- Nie `realloc`, `free` auf statisches Array anwenden
  - Führt auf Laufzeitfehler, da Compiler `free` selbständig hinzugefügt hat!
- Ansonsten gleicher Zugriff auf Komponenten wie bei statischen Arrays
  - Indizierung `array[j]` für $0 \leq j \leq N-1$

Vektor-Bibliothek

- Aufteilen von Source-Code auf mehrere Files
- Precompiler, Compiler, Linker
- Objekt-Code

- `gcc -c`
- `make`

Aufteilen von Source-Code

- längere Source-Codes auf mehrere Files aufteilen
- Vorteil:
  - übersichtlicher
  - Bildung von Bibliotheken
    - Wiederverwendung von alten Codes
    - vermeidet Fehler
- `gcc name1.c name2.c ...`
  - erstellt *ein* Executable aus Source-Codes
  - Reihenfolge der Codes nicht wichtig
  - analog zu `gcc all.c`
    - wenn `all.c` ganzen Source-Code enthält
  - insb. Funktionsnamen müssen eindeutig sein
  - `main()` darf nur 1x vorkommen

Precompiler, Compiler & Linker

- Beim Kompilieren von Source-Code werden mehrere Stufen durchlaufen:

(1) Preprocessor-Befehle ausführen, z.B. `#include`

(2) Compiler erstellt Objekt-Code

(3) Objekt-Code aus Bibliotheken wird hinzugefügt

(4) Linker ersetzt symbolische Namen im Objekt-Code durch Adressen und erzeugt Executable

- Bibliotheken = vorkompilierter Objekt-Code
  - plus zugehöriges Header-File
- Standard-Linker in Unix ist `ld`
- Nur Schritt (3) fertig, d.h. Objekt-Code erzeugen
  - `gcc -c name.c` erzeugt Objekt-Code `name.o`
  - gut zum Debuggen von Syntax-Fehlern!
- Objekt-Code händisch hinzufügen + kompilieren
  - `gcc name.c bib1.o bib2.o ...`
  - `gcc name.o bib1.o bib2.o ...`
  - Reihenfolge + Anzahl der Objekt-Codes ist egal
- Ziel: selbst Bibliotheken erstellen
  - spart ggf. Zeit beim Kompilieren
  - vermeidet Fehler

Eine erste Bibliothek

```
 1 #ifndef _DYNAMICVECTORS_
 2 #define _DYNAMICVECTORS_
 3
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <assert.h>
 7
 8 // allocate + initialize dynamic double vector of length n
 9 double* mallocVector(int n);
10
11 // free a dynamic vector and set the pointer to NULL
12 double* freeVector(double* vector);
13
14 // extend dynamic double vector and initialize new entries
15 double* reallocVector(double* vector, int n, int nnew);
16
17 // allocate dynamic double vector of length n and read
18 // entries from keyboard
19 double* scanVector(int n);
20
21 // print dynamic double vector of length n to shell
22 void printVector(double* vector, int n);
23
24 #endif
```

- Header-File `dynamicvectors.h` zur Bibliothek
  - enthält alle Funktionssignaturen
  - enthält Kommentare zu den Funktionen
- Header-File beginnt mit

  ```
  #ifndef NAME
  #define NAME
  ```

- Header-File ended mit

  ```
  #endif
  ```

- erlaubt mehrfaches Einbinden
  - vermeidet doppelte Deklaration

Source-Code 1/2

```
 1 #include "dynamicvectors.h"
 2
 3 double* mallocVector(int n) {
 4   int j = 0;
 5   double* vector = NULL;
 6   assert(n > 0);
 7
 8   vector = malloc(n*sizeof(double));
 9   assert(vector != NULL);
10
11   for (j=0; j<n; ++j) {
12     vector[j] = 0;
13   }
14   return vector;
15 }
16
17 double* freeVector(double* vector) {
18   free(vector);
19   return NULL;
20 }
21
22 double* reallocVector(double* vector, int n, int nnew) {
23   int j = 0;
24   assert(vector != NULL);
25   assert(n > 0);
26   assert(nnew > 0);
27
28   vector = realloc(vector,nnew*sizeof(double));
29   assert(vector != NULL);
30   for (j=n; j<nnew; ++j) {
31     vector[j] = 0;
32   }
33   return vector;
34 }
```

- Einbinden des Header-Files (Zeile 1)
  - `#include "..."` mit Angabe des Verzeichnis
  - `#include <...>` für Standard-Verzeichnis

Source-Code 2/2

```
36 double* scanVector(int n) {
37   int j = 0;
38   double* vector = NULL;
39   assert(n > 0);
40
41   vector = mallocVector(n);
42   assert(vector != NULL);
43
44   for (j=0; j<n; ++j) {
45     printf("vector[%d] = ",j);
46     scanf("%lf",&vector[j]);
47   }
48   return vector;
49 }
50
51 void printVector(double* vector, int n) {
52   int j = 0;
53   assert(vector != NULL);
54   assert(n > 0);
55
56   for (j=0; j<n; ++j) {
57     printf("%d: %f\n",j,vector[j]);
58   }
59 }
```

Hauptprogramm

```
 1 #include "dynamicvectors.h"
 2
 3 main() {
 4   double* x = NULL;
 5   int n = 10;
 6   int j = 0;
 7   x = mallocVector(n);
 8   for (j=0; j<n; ++j) {
 9     x[j] = j;
10   }
11   x = reallocVector(x,n,2*n);
12   for (j=n; j<2*n; ++j) {
13     x[j] = 10*j;
14   }
15   printVector(x,2*n);
16   x = freeVector(x);
17 }
```

- Hauptprogramm bindet Header der Bibliothek ein
- Kompilieren mittels
  - `gcc -c dynamicvectors.c`
    - erzeugt Object-Code `dynamicvectors.o`
  - `gcc dynamicvectors_main.c dynamicvectors.o`
    - erzeugt Executable `a.out`

Statische Bibliotheken und make

```
1 exe : main.o dynamicvectors.o
2         gcc -o exe main.o dynamicvectors.o
3
4 main.o : dynamicvectors_main.c dynamicvectors.h
5         gcc -c dynamicvectors_main.c -o main.o
6
7 dynamicvectors.o : dynamicvectors.c dynamicvectors.h
8         gcc -c dynamicvectors.c
```

- UNIX-Befehl make erlaubt Abhängigkeiten von Code automatisch zu handeln
  - Automatisierung spart Zeit für Kompilieren
  - Nur wenn Source-Code geändert wurde, wird neuer Objekt-Code erzeugt und abhängiger Code wird neu kompiliert
- Aufruf make befolgt Steuerdatei Makefile
- Aufruf make -f filename befolgt filename
- Datei zeigt Abhängigkeiten und Befehle, z.B.
  - Zeile 1 = Abhängigkeit (nicht eingerückt!)
    - Datei exe hängt ab von ...
  - Zeile 2 = Befehl (eine Tabulator-Einrückung!)
    - Falls exe älter ist als Abhängigkeiten, wird Befehl ausgeführt (und nur dann!)
- mehr zu make in Schmaranz C-Buch, Kapitel 15

Dynamische Matrizen

- Pointer höherer Ordnung
- dynamische Matrizen
- Matrix-Matrix-Multiplikation

Statische Matrizen

- Pointer sind Datentypen $\Rightarrow \exists$ Pointer auf Pointer

- `double array[M][N];` deklariert statische Matrix `array` der Dimension $M \times N$ mit `double`-Koeffizienten
  - Indizierung mittels `array[j][k]` mit $0 \leq j \leq M-1$ und $0 \leq k \leq N-1$
  - Dimensionen `M`, `N` können während Programmablauf nicht verändert werden
  - Funktionen können `M`, `N` nicht herausfinden, d.h. stets als Input-Parameter übergeben

- Speicherung statischer Matrizen ist zeilenweise
  - eventuell nicht allokierbar, falls Speicher fragmentiert

- Formal: Zeile `array[j]` ist Vektor der Länge `N` mit Koeffizienten vom Typ `double`
  - also `array[j]` intern vom Typ `double*`

- Formal: `array` Vektor der Länge `M` mit Koeffizienten vom Typ `double*`
  - also `array` intern vom Typ `double**`

Dynamische Matrizen

- statische Matrix `double array[M][N];`
  - `array` ist `double**` [`double*`-Vektor der Länge `M`]
  - `array[j]` ist `double*` [`double`-Vektor der Länge `N`]

```
 3 double** mallocMatrix(int m, int n) {
 4   int j = 0;
 5   int k = 0;
 6   double** matrix = NULL;
 7   assert(m > 0);
 8   assert(n > 0);
 9
10   matrix = malloc(m*sizeof(double*));
11   assert(matrix != NULL);
12   for (j=0; j<m; ++j) {
13     matrix[j] = malloc(n*sizeof(double));
14     assert(matrix[j] != NULL);
15     for (k=0; k<n; ++k) {
16       matrix[j][k] = 0;
17     }
18   }
19   return matrix;
20 }
```

- Mit Hilfe der Bibliothek für dyn. Vektoren gilt

```
 3 double** mallocMatrix(int m, int n) {
 4   int j = 0;
 5   int k = 0;
 6   double** matrix = NULL;
 7   assert(m > 0);
 8   assert(n > 0);
 9
10   matrix = malloc(m*sizeof(double*));
11   assert(matrix != NULL);
12   for (j=0; j<m; ++j) {
13     matrix[j] = mallocVector(n);
14   }
15   return matrix;
16 }
```

Freigeben dynamischer Matrizen

- Freigeben einer dynamischen Matrix in umgekehrter Reihenfolge:
  - erst die Zeilenvektoren `matrix[j]` freigeben
  - dann Spaltenvektor `matrix` freigeben
- Funktion muss wissen, wie viele Zeilen Matrix hat

```
18 double** freeMatrix(double** matrix, int m) {
19   int j = 0;
20   assert(matrix != NULL);
21   assert(m > 0);
22
23   for (j=0; j<m; ++j) {
24     free(matrix[j]);
25   }
26   free(matrix);
27   return NULL;
28 }
```

- An dieser Stelle kein Gewinn durch Bibliothek für dynamische Vektoren

Re-Allokation 1/4

- Größe `M` × `N` soll auf `Mnew` × `Nnew` geändert werden
  - Funktion soll möglichst wenig Speicher brauchen
- Falls `Mnew` < `M`
  - Speicher von überflüssigen `matrix[j]` freigeben
  - Pointer-Vektor `matrix` mit `realloc` kürzen
  - Alle gebliebenen `matrix[j]` mit `realloc` kürzen oder verlängern, neue Einträge initialisieren

```
34 double** reallocMatrix(double** matrix, int m, int n,
35                        int mnew, int nnew) {
36
37   int j = 0;
38   int k = 0;
39   assert(matrix != NULL);
40   assert(m > 0);
41   assert(n > 0);
42   assert(mnew > 0);
43   assert(nnew > 0);
44
45   if (mnew < m) {
46     for (j=mnew; j<m; ++j) {
47       free(matrix[j]);
48     }
49     matrix = realloc(matrix,mnew*sizeof(double*));
50     assert(matrix != NULL);
51     for (j=0; j<mnew; ++j) {
52       matrix[j] = realloc(matrix[j], nnew*sizeof(double));
53       assert(matrix[j] != NULL);
54       for (k=n; k<nnew; ++k) {
55         matrix[j][k] = 0;
56       }
57     }
58   }
```

Re-Allokation 2/4

- wesentlicher Code von letzter Folie:

```
45   if (mnew < m) {
46     for (j=mnew; j<m; ++j) {
47       free(matrix[j]);
48     }
49     matrix = realloc(matrix,mnew*sizeof(double*));
50     assert(matrix != NULL);
51     for (j=0; j<mnew; ++j) {
52       matrix[j] = realloc(matrix[j], nnew*sizeof(double));
53       assert(matrix[j] != NULL);
54       for (k=n; k<nnew; ++k) {
55         matrix[j][k] = 0;
56       }
57     }
58   }
```

- Realisierung mittels Bibliothek für dyn. Vektoren:

```
41   if (mnew < m) {
42     for (j=mnew; j<m; ++j) {
43       free(matrix[j]);
44     }
45     matrix = realloc(matrix,mnew*sizeof(double*));
46     assert(matrix != NULL);
47     for (j=0; j<mnew; ++j) {
48       matrix[j] = reallocVector(matrix[j],n,nnew);
49     }
50   }
```

Re-Allokation 3/4

- Falls `Mnew` $\geq$ `M`
  - Alle vorhandenen `matrix[j]` mit `realloc` kürzen oder verlängern, neue Einträge initialisieren
  - Pointer-Vektor `matrix` mit `realloc` verlängern
  - Neue Zeilen `matrix[j]` allokieren & initialisieren

```
59   else {
60     for (j=0; j<m; ++j) {
61       matrix[j] = realloc(matrix[j],nnew*sizeof(double));
62       assert(matrix[j] != NULL);
63       for (k=n; k<nnew; ++k) {
64         matrix[j][k] = 0;
65       }
66     }
67     matrix = realloc(matrix,mnew*sizeof(double*));
68     assert(matrix != NULL);
69     for (j=m; j<mnew; ++j) {
70       matrix[j] = malloc(nnew*sizeof(double));
71       assert(matrix[j] != NULL);
72       for (k=0; k<nnew; ++k) {
73         matrix[j][k] = 0;
74       }
75     }
76   }
```

- Realisierung mittels Bibliothek für dyn. Vektoren:

```
51   else {
52     for (j=0; j<m; ++j) {
53       matrix[j] = reallocVector(matrix[j],n,nnew);
54     }
55     matrix = realloc(matrix,mnew*sizeof(double*));
56     assert(matrix != NULL);
57     for (j=m; j<mnew; ++j) {
58       matrix[j] = mallocVector(nnew);
59     }
60   }
```

Re-Allokation 4/4

```
30 double** reallocMatrix(double** matrix, int m, int n,
31                        int mnew, int nnew) {
32
33   int j = 0;
34   int k = 0;
35   assert(matrix != NULL);
36   assert(m > 0);
37   assert(n > 0);
38   assert(mnew > 0);
39   assert(nnew > 0);
40
41   if (mnew < m) {
42     for (j=mnew; j<m; ++j) {
43       free(matrix[j]);
44     }
45     matrix = realloc(matrix,mnew*sizeof(double*));
46     assert(matrix != NULL);
47     for (j=0; j<mnew; ++j) {
48       matrix[j] = reallocVector(matrix[j],n,nnew);
49     }
50   }
51   else {
52     for (j=0; j<m; ++j) {
53       matrix[j] = reallocVector(matrix[j],n,nnew);
54     }
55     matrix = realloc(matrix,mnew*sizeof(double*));
56     assert(matrix != NULL);
57     for (j=m; j<mnew; ++j) {
58       matrix[j] = mallocVector(nnew);
59     }
60   }
61   return matrix;
62 }
```

Bemerkungen

- `sizeof` bei `malloc`/`realloc` nicht vergessen
- Typ des Pointers muss passen zum Typ in `sizeof`
- Größe `M` $\times$ `N` einer Matrix muss man sich merken
- Base Pointer `matrix` darf man weder verlieren noch verändern!
- Den Vektor `matrix` darf man nur kürzen, wenn vorher der Speicher der Komponenten `matrix[j]` freigegeben wurde
- Freigeben des Vektors `matrix` gibt nicht den Speicher der Zeilenvektoren frei
  - ggf. entsteht toter Speicherbereich, der nicht mehr ansprechbar ist, bis Programm terminiert
- Nacheinander allokierte Speicherbereiche liegen nicht notwendig hintereinander im Speicher
  - jede Zeile `matrix[j]` liegt zusammenhängend im Speicher
  - Gesamtmatrix kann verstreut im Speicher liegen

Strings

- statische & dynamische Strings
- `"..."` vs. `'...'`

- `string.h`

Strings (= Zeichenketten)

- Strings = `char`-Arrays, also 2 Definitionen möglich
  - statisch: `char array[N];`
    - `N` = statische Länge
    - Deklaration & Initialisierung möglich
      `char array[] = "text";`
  - dynamisch (wie oben, Typ: `char*`)

- Fixe Strings in Anführungszeichen `"..."`

- Zugriff auf einzelnes Zeichen mittels `'...'`

- Zugriff auf Teil-Strings nicht möglich!

- Achtung bei dynamischen Strings:
  - als Standard enden alle Strings mit Null-Byte `\0`
    - Länge eines Strings dadurch bestimmen!
  - Bei statischen Arrays geschieht das automatisch (also wirkliche Länge `N+1` und `array[N]='\0'`)
    - Bei dyn. Strings also 1 Byte mehr reservieren!
    - und `\0` nicht vergessen

- An Funktionen können auch fixe Strings (in Anführungszeichen) übergeben werden
  - z.B. `printf("Hello World!\n");`

Funktionen zur String-Manipulation

- Wichtigste Funktionen in `stdio.h`
  - `sprintf`: konvertiert Variable → String
  - `sscanf`: konvertiert String → Variable
- zahlreiche Funktionen in `stdlib.h`, z.B.
  - `atof`: konvertiert String → `double`
  - `atoi`: konvertiert String → `int`
- oder in `string.h`, z.B.
  - `strchr`, `memchr`: Suche `char` innerhalb String
  - `strcmp`, `memcmp`: Vergleiche zwei Strings
  - `strcpy`, `memcpy`: Kopieren von Strings
  - `strlen`: Länge eines Strings (ohne Null-Byte)
- Header-Files mit `#include <name>` einbinden!
- Gute Referenz mit allen Befehlen & Erklärungen
  `http://www.acm.uiuc.edu/webmonkeys/book/c_guide/`
- Details zu den Befehlen mit `man 3 befehl`
- ACHTUNG mit String-Befehlen: Befehle können nicht wissen, ob für den Output-String genügend Speicher allokiert ist (→ Laufzeitfehler!)

Beispiel

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <assert.h>
 5
 6 char* stringCopy(char* source) {
 7   int length = 0;
 8   char* result = NULL;
 9   assert(source != NULL);
10
11   length = strlen(source);
12   result = malloc((length+1)*sizeof(char));
13   strcpy(result,source);
14   return result;
15 }
16
17 main() {
18   char* string1 = "Hello World?";
19   char* string2 = stringCopy(string1);
20   string2[11] = '!';
21   printf("%s %s\n",string1,string2);
22 }
```

- Output:

  ```
  Hello World? Hello World!
  ```

- Fixe Strings in Anführungszeichen `"..."` (Z. 18)
  - erzeugt statisches Array mit zusätzlichem Null-Byte am Ende
- Zugriff auf einzelne Zeichen eines Strings mit einfachen Hochkommata `'...'` (Zeile 20)
- Platzhalter für Strings in `printf` ist `%s` (Zeile 21)

Ganzzahlen

- Bits, Bytes etc.

- `short`, `int`, `long`
- `unsigned`

Speichereinheiten

- 1 Bit = 1 b = kleinste Einheit, speichert 0 oder 1
- 1 Byte = 1 B = Zusammenfassung von 8 Bit
- 1 Kilobyte = 1 KB = 1024 Byte
- 1 Megabyte = 1 MB = 1024 KB
- 1 Gigabyte = 1 GB = 1024 MB
- 1 Terabyte = 1 TB = 1024 GB

Speicherung von Zahlen

- Zur Speicherung von Zahlen wird je nach Datentyp fixe Anzahl an Bytes verwendet
- Konsequenz:
  - pro Datentyp gibt es nur endlich viele Zahlen
    - es gibt jeweils größte und kleinste Zahl!

Ganzzahlen

- Mit n Bits kann man 2^n Ganzzahlen darstellen
- Standardmäßig betrachtet man
  - entweder alle ganzen Zahlen in $[0, 2^n - 1]$
  - oder alle ganzen Zahlen in $[-2^{n-1}, 2^{n-1} - 1]$

Integer-Arithmetik

- exakte Arithmetik innerhalb $[\texttt{intmin}, \texttt{intmax}]$
- Überlauf: Ergebnis von Rechnung $> \texttt{intmax}$
- Unterlauf: Ergebnis von Rechnung $< \texttt{intmin}$
- Integer-Arithmetik ist i.d.R. Modulo-Arithmetik
  - d.h. Zahlenbereich ist geschlossen
    - $\texttt{intmax} + 1$ liefert intmin
    - $\texttt{intmin} - 1$ liefert intmax
  - nicht im C-Standard festgelegt!

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int j = 0;
 5   int n = 8*sizeof(int); // number bits per int
 6   int min = 1;
 7
 8   // compute 2^(n-1)
 9   for (j=1; j<n; ++j) {
10     min = 2*min;
11   }
12   printf("n=%d, min=%d, max=%d\n",n,min,min-1);
13 }
```

- man beobachtet $[-2^{n-1}, 2^{n-1} - 1]$ mit $n = 32$
- Output:

```
n=32, min=-2147483648, max=2147483647
```

2 Milliarden sind nicht viel!

```
 1 #include <stdio.h>
 2
 3 main() {
 4   int n = 1;
 5   int factorial = 1;
 6
 7   do {
 8     ++n;
 9     factorial = n*factorial;
10     printf("n=%d, n!=%d\n",n,factorial);
11   } while (factorial < n*factorial);
12
13   printf("n=%d, n!>%d\n",n+1,n*factorial);
14 }
```

- Output:

```
n=2, n!=2
n=3, n!=6
n=4, n!=24
n=5, n!=120
n=6, n!=720
n=7, n!=5040
n=8, n!=40320
n=9, n!=362880
n=10, n!=3628800
n=11, n!=39916800
n=12, n!=479001600
n=13, n!=1932053504
n=14, n!>-653108224
```

Variablentypen short, int, long

- n Bits $\Rightarrow$ 2^n Ganzzahlen
- In C sind `short`, `int`, `long` mit Vorzeichen
  - d.h. ganze Zahlen in $[-2^{n-1}, 2^{n-1} - 1]$
- Ganzzahlen ≥ 0 durch zusätzliches `unsigned`
  - d.h. ganze Zahlen in $[0, 2^n - 1]$
  - z.B. `unsigned int var1 = 0;`
- Es gilt stets `short` $\leq$ `int` $\leq$ `long`
  - Standardlängen: 2 Byte (`short`), 4 Byte (`int`)
  - Häufig gilt `int` = `long`
  - Für die UE nur `int` (und `short`) verwenden
- Platzhalter in `printf` und `scanf`

Datentyp	printf	scanf
short	%d	
int	%d	%d
unsigned short	%u	
unsigned int	%u	%u

Variablentypen char

- `char` ist Ganzzahl-Typ, idR. 1 Byte
- Zeichen sind intern Ganzzahlen zugeordnet
  - idR. ASCII-Code
  - siehe z.B. http://www.asciitable.com/
- ASCII-Code eines Buchstabens erhält man durch einfache Hochkommata
  - Deklaration `char var = 'A';` weist `var` ASCII-Code des Buchstabens `A` zu
- Platzhalter eines Zeichens für `printf` und `scanf`
  - `%c` als Zeichen
  - `%d` als Ganzzahl

```
1 #include <stdio.h>
2
3 main() {
4   char var = 'A';
5
6   printf("sizeof(var) = %d\n", sizeof(var));
7   printf("%c %d\n",var,var);
8 }
```

- Output:

```
sizeof(var) = 1
A 65
```

Gleitkommazahlen

- analytische Binärdarstellung
- Gleitkomma-Zahlsystem $\mathbb{F}(2, M, e_{\text{min}}, e_{\text{max}})$
- schlecht gestellte Probleme
- Rechenfehler und Gleichheit

- `float`, `double`

Gleitkommadarstellung 1/2

- SATZ: Zu $x \in \mathbb{R}$ existieren
  - Vorzeichen $\sigma \in \{\pm 1\}$
  - Ziffern $a_k \in \{0, 1\}$
  - Exponent $e \in \mathbb{Z}$

  sodass gilt $x = \sigma \left(\sum_{k=1}^{\infty} a_k 2^{-k} \right) 2^e$

- Darstellung ist nicht eindeutig, da z.B. $1 = \sum_{k=1}^{\infty} 2^{-k}$

Bemerkungen

- Satz gilt für jede Basis $b \in \mathbb{N}_{\geq 2}$
  - Ziffern dann $a_j \in \{0, 1, \ldots, b-1\}$
- Dezimalsystem $b = 10$ ist übliches System
  - $47.11 = (4 \cdot 10^{-1} + 7 \cdot 10^{-2} + 1 \cdot 10^{-3} + 1 \cdot 10^{-4}) \cdot 10^2$
    - $a_1 = 4$, $a_2 = 7$, $a_3 = 1$, $a_4 = 1$, $e = 2$
- Mit $b = 2$ sind Brüche genau dann als endliche Summe darstellbar, wenn Nenner Zweierpotenz:
  - $\sum_{k=1}^{M} 2^{-k}$ hat Nenner mit Zweierpotenz
  - Eindeutigkeit der Primfaktorzerlegung
- z.B. keine exakte Darstellung für 1/10 für $b = 2$

Gleitkommadarstellung 2/2

- SATZ: Zu $x \in \mathbb{R}$ existieren
  - Vorzeichen $\sigma \in \{\pm 1\}$
  - Ziffern $a_k \in \{0, 1\}$
  - Exponent $e \in \mathbb{Z}$

  sodass gilt $x = \sigma \left(\sum_{k=1}^{\infty} a_k 2^{-k} \right) 2^e$

- Darstellung ist nicht eindeutig, da z.B. $1 = \sum_{k=1}^{\infty} 2^{-k}$

Gleitkommazahlen

- Gleitkommazahlsystem $\mathbb{F}(2, M, e_{\min}, e_{\max}) \subset \mathbb{Q}$
  - Mantissenlänge $M \in \mathbb{N}$
  - Exponentialschranken $e_{\min} < 0 < e_{\max}$

- $x \in \mathbb{F}$ hat Darstellung $x = \sigma \left(\sum_{k=1}^{M} a_k 2^{-k} \right) 2^e$ mit
  - Vorzeichen $\sigma \in \{\pm 1\}$
  - Ziffern $a_j \in \{0, 1\}$ mit $a_1 = 1$
    - sog. normalisierte Gleitkommazahl
  - Exponent $e \in \mathbb{Z}$ mit $e_{\min} \leq e \leq e_{\max}$

- Darstellung von $x \in \mathbb{F}$ ist eindeutig (Übung!)
- Ziffer a_1 muss nicht gespeichert werden
  - implizites erstes Bit

Beweis von Satz

- o.B.d.A. $x \geq 0$ — Multipliziere ggf. mit $\sigma = -1$.
- Sei $e \in \mathbb{N}_0$ mit $0 \leq x < 2^e$
- o.B.d.A. $x < 1$ — Teile durch 2^e
- Konstruktion der Ziffern a_j durch Bisektion:

- Induktionsbehauptung: Ex. Ziffern $a_j \in \{0, 1\}$
  - sodass $x_n := \sum_{k=1}^{n} a_k 2^{-k}$ erfüllt $x \in [x_n, x_n + 2^{-n})$
- Induktionsanfang: Es gilt $x \in [0, 1)$
  - falls $x \in [0, 1/2)$, wähle $a_1 = 0$, d.h. $x_1 = 0$
  - falls $x \in [1/2, 1)$, wähle $a_1 = 1$, d.h. $x_1 = 1/2$
    - $x_1 = a_1/2 \leq x$
    - $x < (a_1 + 1)/2 = x_1 + 2^{-1}$
- Induktionsschritt: Es gilt $x \in [x_n, x_n + 2^{-n})$
  - falls $x \in [x_n, x_n + 2^{-(n+1)})$, wähle $a_{n+1} = 0$, d.h. $x_{n+1} = x_n$
  - falls $x \in [x_n + 2^{-(n+1)}, x_n + 2^{-n})$, wähle $a_{n+1} = 1$
    - $x_{n+1} = x_n + a_{n+1} 2^{-(n+1)} \leq x$
    - $x < x_n + (a_{n+1}+1)2^{-(n+1)} = x_{n+1} + 2^{-(n+1)}$
- Es folgt $|x_n - x| \leq 2^{-n}$, also $x = \sum_{k=1}^{\infty} a_k 2^{-k}$

Arithmetik für Gleitkommazahlen

- Ergebnis `Inf`, `-Inf` bei Überlauf (oder `1./0.`)
- Ergebnis `NaN`, falls nicht definiert (z.B. `0./0.`)
- Arithmetik ist approximativ, nicht exakt

Schlechte Kondition

- Eine Aufgabe ist numerisch schlecht gestellt, falls kleine Änderungen der Daten auf große Änderungen im Ergebnis führen
  - z.B. hat Dreieck mit gegebenen Seitenlängen einen rechten Winkel?
  - z.B. liegt gegebener Punkt auf Kreisrand?
- Implementierung sinnlos, weil Ergebnis zufällig!

Rechenfehler

- Aufgrund von Rechenfehlern darf man Gleitkommazahlen *nie* auf Gleichheit überprüfen
  - Statt $x = y$ prüfen, ob Fehler $|x - y|$ klein ist
  - z.B. $|x - y| \leq \varepsilon \cdot \max\{|x|, |y|\}$ mit $\varepsilon = 10^{-13}$

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 main() {
 5   double x = (116./100.)*100.;
 6
 7   printf("x=%f\n",x);
 8   printf("floor(x)=%f\n",floor(x));
 9
10   if (x==116.) {
11     printf("There holds x==116\n");
12   }
13   else {
14     printf("Surprise, surprise!\n");
15   }
16 }
```

- Output:

```
x=116.000000
floor(x)=115.000000
Surprise, surprise!
```

Variablentypen float, double

```
1 #include <stdio.h>
2 main() {
3   double x = 2./3.;
4   float y = 2./3.;
5   printf("%f, %1.16e\n", x, x);
6   printf("%f, %1.7e\n",y, y);
7 }
```

- Gleitkommazahlen sind endliche Teilmenge von $\mathbb{Q}$
- `float` ist idR. einfache Genauigkeit nach IEEE-754-Standard
  - $\mathbb{F}(2, 24, -126, 127)$ $\rightarrow$ 4 Byte
  - sog. *single precision*
  - ca. 7 signifikante Dezimalstellen
- `double` ist idR. doppelte Genauigkeit nach IEEE-754-Standard
  - $\mathbb{F}(2, 53, -1022, 1023)$ $\rightarrow$ 8 Byte
  - sog. *double precision*
  - ca. 16 signifikante Dezimalstellen
- Platzhalter in `printf` und `scanf`

Datentyp	printf	scanf
float	%f	%f
double	%f	%lf

  - Platzhalter `%1.16e` für Gleitkommadarstellung
  - siehe `man 3 printf`
- Output:

```
0.666667, 6.6666666666666663e-01
0.666667, 6.6666669e-01
```

Strukturen

- Warum Strukturen?
- Members
- Punktoperator `.`
- Pfeiloperator `->`
- Shallow Copy vs. Deep Copy

- `struct`
- `typedef`

Deklaration von Strukturen

- **Funktionen**
  - Zusammenfassung von versch. Befehlen, um Abstraktionsebenen zu schaffen
- **Strukturen**
  - Zusammenfassung von Variablen versch. Typs zu einem neuen Datentyp
  - Abstraktionsebenen bei Daten
- **Beispiel:** Verwaltung der EPROG-Teilnehmer
  - pro Student jeweils denselben Datensatz

```
 1 // Declaration of structure
 2 struct _Student_ {
 3   char* firstname; // Vorname
 4   char* lastname;  // Nachname
 5   int studentID;   // Matrikelnummer
 6   int studiesID;   // Studienkennzahl
 7   int test;        // Punkte im Test
 8   int kurztest;    // Punkte in Kurztests
 9   int uebung;      // Punkte in der Uebung
10 };
11
12 // Declaration of corresponding data type
13 typedef struct _Student_ Student;
```

- Semikolon nach Struktur-Deklarations-Block
- erzeugt neuen Variablen-Typ `Student`

Strukturen & Members

- Datentypen einer Struktur heißen Members
- Zugriff auf Members mit Punkt-Operator
  - `var` Variable vom Typ `Student`
  - z.B. Member `var.firstname`

```
 1 // Declaration of structure
 2 struct _Student_ {
 3   char* firstname; // Vorname
 4   char* lastname;  // Nachname
 5   int studentID;   // Matrikelnummer
 6   int studiesID;   // Studienkennzahl
 7   int test;        // Punkte im Test
 8   int kurztest;    // Punkte in Kurztests
 9   int uebung;      // Punkte in der Uebung
10 };
11
12 // Declaration of corresponding data type
13 typedef struct _Student_ Student;
14
15 main() {
16   Student var;
17   var.firstname = "Dirk";
18   var.lastname = "Praetorius";
19   var.studentID = 0;
20   var.studiesID = 680;
21   var.test = 25;
22   var.kurztest = 30;
23   var.uebung = 35;
24 }
```

Bemerkungen zu Strukturen

- laut erstem C-Standard verboten:
  - Struktur als Input-Parameter einer Funktion
  - Struktur als Output-Parameter einer Funktion
  - Zuweisungsoperator (=) für gesamte Struktur

- in der Zwischenzeit erlaubt, aber trotzdem:
  - idR. Strukturen dynamisch über Pointer
  - Zuweisung (= Kopieren) selbst schreiben
  - Zuweisung (=) macht sog. *shallow copy*

- Shallow copy:
  - nur die oberste Ebene wird kopiert
  - d.h. Werte bei elementaren Variablen
  - d.h. Adressen bei Pointern
  - also: Kopie hat (physisch!) dieselben dynamischen Daten

- Deep copy:
  - alle Ebenen der Struktur werden kopiert
  - d.h. alle Werte bei elementaren Variablen
  - plus Kopie der dynamischen Inhalte (d.h. durch Pointer adressierter Speicher)

Strukturen: Speicher allokieren

- Also Funktionen anlegen
  - `newStudent`: Allokieren und Initialisieren
  - `freeStudent`: Freigeben des Speichers
  - `cloneStudent`: Vollständige Kopie der Struktur inkl. dyn. Felder, z.B. Member `firstname` (sog. *deep copy*)
  - `copyStudent`: Kopie der obersten Ebene exkl. dynamischer Felder (sog. *shallow copy*)

```
 1 Student* newStudent() {
 2   Student* pointer = malloc(sizeof(Student));
 3   assert( pointer != NULL);
 4
 5   (*pointer).firstname = NULL;
 6   (*pointer).lastname = NULL;
 7   (*pointer).studentID = 0;
 8   (*pointer).studiesID = 0;
 9   (*pointer).test = 0;
10   (*pointer).kurztest = 0;
11   (*pointer).uebung = 0;
12
13   return pointer;
14 }
```

Strukturen & Pfeiloperator

- Im Programm ist `pointer` vom Typ `Student*`
- Zugriff auf Members, z.B. `(*pointer).firstname`
  - Bessere Schreibweise dafür `pointer->firstname`
- Strukturen nie statisch, sondern stets dynamisch
  - Verwende gleich `student` für Typ `Student*`
- Funktion `newStudent` lautet besser wie folgt

```
 5 // Declaration of structure
 6 struct _Student_ {
 7   char* firstname; // Vorname
 8   char* lastname;  // Nachname
 9   int studentID;   // Matrikelnummer
10   int studiesID;   // Studienkennzahl
11   int test;        // Punkte im Test
12   int kurztest;    // Punkte in Kurztests
13   int uebung;      // Punkte in der Uebung
14 };
15
16 // Declaration of corresponding data type
17 typedef struct _Student_ Student;
18
19 // allocate and initialize new student
20 Student* newStudent() {
21   Student* student = malloc(sizeof(Student));
22   assert(student != NULL);
23
24   student->firstname = NULL;
25   student->lastname = NULL;
26   student->studentID = 0;
27   student->studiesID = 0;
28   student->test = 0;
29   student->kurztest = 0;
30   student->uebung = 0;
31
32   return student;
33 }
```

Strukturen: Speicher freigeben

- Freigeben einer dynamisch erzeugten Struktur-Variable vom Typ Student
- Achtung: Zugewiesenen dynamischen Speicher vor Freigabe des Strukturpointers freigeben

```
35 // free memory allocation
36 Student* delStudent(Student* student) {
37   assert(student != NULL);
38
39   if (student->firstname != NULL) {
40     free(student->firstname);
41   }
42
43   if (student->lastname != NULL) {
44     free(student->lastname);
45   }
46
47   free(student);
48   return NULL;
49 }
```

Shallow Copy

- Kopieren einer dynamisch erzeugten Struktur-Variable vom Typ `Student`
  - Kopieren der obersten Ebene einer Struktur exklusive dynamischen Speicher (Members!)

```
51 // shallow copy of student
52 Student* copyStudent(Student* student) {
53   Student* copy = newStudent();
54   assert(student != NULL);
55
56   // ACHTUNG: Pointer!
57   copy->firstname = student->firstname;
58   copy->lastname = student->lastname;
59
60   // Kopieren der harmlosen Daten
61   copy->studentID = student->studentID;
62   copy->studiesID = student->studiesID;
63   copy->test = student->test;
64   copy->kurztest = student->kurztest;
65   copy->uebung = student->uebung;
66
67   return copy;
68 }
```

Deep Copy

- Kopieren einer dynamisch erzeugten Struktur-Variable vom Typ Student
- Vollständige Kopie, inkl. dynamischem Speicher
- Achtung: Zugewiesenen dynamischen Speicher mitkopieren

```
70 // deep copy of student
71 Student* cloneStudent(Student* student) {
72   Student* copy = newStudent();
73   int length = 0;
74   assert( student != NULL);
75
76   if (student->firstname != NULL) {
77     length = strlen(student->firstname)+1;
78     copy->firstname = malloc(length*sizeof(char));
79     assert(copy->firstname != NULL);
80     strcpy(copy->firstname, student->firstname);
81   }
82
83
84   if (student->lastname != NULL) {
85     length = strlen(student->lastname)+1;
86     copy->lastname = malloc(length*sizeof(char));
87     assert(copy->lastname != NULL);
88     strcpy(copy->lastname, student->lastname);
89   }
90
91   copy->studentID = student->studentID;
92   copy->studiesID = student->studiesID;
93   copy->test = student->test;
94   copy->kurztest = student->kurztest;
95   copy->uebung = student->uebung;
96
97   return copy;
98 }
```

Arrays von Strukturen

- Ziel: Array mit Teilnehmern von EPROG erstellen
- keine statischen Arrays verwenden,
  sondern dynamische Arrays
  - Studenten-Daten sind vom Typ `Student`
  - also intern verwaltet mittels Typ `Student*`
  - also Array vom Typ `Student**`

```
1 // Declare array
2 Student** participant=malloc(N*sizeof(Student*));
3
4 // Allocate memory for participants
5 for (j=0; j<N; ++j){
6   participant[j] = newStudent();
7 }
```

- Zugriff auf Members wie vorher
  - `participant[j]` ist vom Typ `Student*`
  - also z.B. `participant[j]->firstname`

Schachtelung von Strukturen

```
 1 struct _Address_ {
 2   char* street;
 3   char* number;
 4   char* city;
 5   char* zip;
 6 };
 7 typedef struct _Address_ Address;
 8
 9 struct _Employee_ {
10   char* firstname;
11   char* lastname;
12   char* title;
13   Address* home;
14   Address* office;
15 };
16 typedef struct _Employee_ Employee;
```

- Mitarbeiterdaten strukturieren
  - Name, Wohnadresse, Büroadresse
- Für `employee` vom Typ `Employee*`
  - `employee->home` Pointer auf `Address`
  - also z.B. `employee->home->city`
- Achtung beim Allokieren, Freigeben, Kopieren

Strukturen & Math

- Strukturen für mathematische Objekte:
  - allgemeine Vektoren
  - Matrizen

Strukturen und Vektoren

```
 1 #ifndef _STRUCT_VECTOR_
 2 #define _STRUCT_VECTOR_
 3
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <assert.h>
 7 #include <math.h>
 8
 9 // declaration of new data type Vector
10 typedef struct _Vector_ {
11   int n;          // Dimension
12   double* entry;  // Vector coefficients
13 } Vector;
14
15 // Allocate and initialize new vector of length n
16 Vector* newVector(int n);
17
18 // free storage of allocated vector and return NULL
19 Vector* delVector(Vector* X);
20
21 // return length of a vector
22 int getVectorN(Vector* X);
23
24 // return coefficient Xi of vector X
25 double getVectorEntry(Vector* X, int i);
26
27 // assign new value to coefficient Xi of vector X
28 void setVectorEntry(Vector* X, int i, double Xi);
29
30 // some example functions...
31 Vector* inputVector();
32 double normVector(Vector* X);
33 double productVector(Vector* X, Vector* Y);
34
35 #endif
```

- Datentyp zur Speicherung von $x \in \mathbb{R}^n$
  - Dimension n vom Typ `int`
  - Datenfeld x_j zur Speicherung von `double`

Allokieren eines Vektors

- Funktion bekommt Länge $n \in \mathbb{N}$ des Vektors
- allokiert Struktur, weist Dimension n zu
- allokiert und initialisiert Datenfeld

```
 3 Vector* newVector(int n) {
 4   int i = 0;
 5   Vector* X = NULL;
 6
 7   assert(n > 0);
 8
 9   X = malloc(sizeof(Vector));
10   assert(X != NULL);
11
12   X->n = n;
13   X->entry = malloc(n*sizeof(double));
14   assert(X->entry != NULL);
15
16   for (i=0; i<n; ++i) {
17     X->entry[i] = 0;
18   }
19   return X;
20 }
```

Freigeben eines Vektors

- Datenfeld freigeben
- Struktur freigeben
- NULL zurückgeben

```
22 Vector* delVector(Vector* X) {
23   assert(X != NULL);
24   free(X->entry);
25   free(X);
26
27   return NULL;
28 }
```

Zugriff auf Strukturen

- Es ist guter (aber seltener) Programmierstil, auf Members einer Struktur nicht direkt zuzugreifen
- Stattdessen lieber
  - für jeden Member `set` und `get` schreiben

```
30 int getVectorN(Vector* X) {
31   assert(X != NULL);
32   return X->n;
33 }
34
35 double getVectorEntry(Vector* X, int i) {
36   assert(X != NULL);
37   assert((i >= 0) && (i < X->n));
38   return X->entry[i];
39 }
40
41 void setVectorEntry(Vector* X, int i, double Xi){
42   assert(X != NULL);
43   assert((i >= 0) && (i < X->n));
44   X->entry[i] = Xi;
45 }
```

- Wenn kein `set`, dann Schreiben nicht erlaubt!
- Wenn kein `get`, dann Lesen nicht erlaubt!
- Dieses Vorgehen erlaubt leichte Umstellung der Datenstruktur bei späteren Modifikationen
- Dieses Vorgehen vermeidet Inkonsistenzen der Daten und insbesondere Laufzeitfehler

Beispiel: Vektor einlesen

```
47 Vector* inputVector() {
48
49   Vector* X = NULL;
50   int i = 0;
51   int n = 0;
52   double input = 0;
53
54   printf("Dimension des Vektors n=");
55   scanf("%d",&n);
56   assert(n > 0);
57
58   X = newVector(n);
59   assert(X != NULL);
60
61   for (i=0; i<n; ++i) {
62     input = 0;
63     printf("x[%d]=",i);
64     scanf("%lf",&input);
65     setVectorEntry(X,i,input);
66   }
67
68   return X;
69 }
```

- Einlesen von $n \in \mathbb{N}$ und eines Vektors $x \in \mathbb{R}^n$

Beispiel: Euklidische Norm

```
71 double normVector(Vector* X) {
72
73   double Xi = 0;
74   double norm = 0;
75   int n = 0;
76   int i = 0;
77
78   assert(X != NULL);
79
80   n = getVectorN(X);
81
82   for (i=0; i<n; ++i) {
83     Xi = getVectorEntry(X,i);
84     norm = norm + Xi*Xi;
85   }
86   norm = sqrt(norm);
87
88   return norm;
89 }
```

- Berechne $\|x\| := \left(\sum_{j=1}^{n} x_j^2 \right)^{1/2}$ für $x \in \mathbb{R}^n$

Beispiel: Skalarprodukt

```
 91 double productVector(Vector* X, Vector* Y) {
 92
 93   double Xi = 0;
 94   double Yi = 0;
 95   double product = 0;
 96   int n = 0;
 97   int i = 0;
 98
 99   assert(X != NULL);
100   assert(Y != NULL);
101
102   n = getVectorN(X);
103   assert(n == getVectorN(Y));
104
105   for (i=0; i<n; ++i) {
106     Xi = getVectorEntry(X,i);
107     Yi = getVectorEntry(Y,i);
108     product = product + Xi*Yi;
109   }
110
111   return product;
112 }
```

- Berechne $x \cdot y := \sum_{j=1}^{n} x_j y_j$ für $x, y \in \mathbb{R}^n$

Strukturen und Matrizen

- Datentyp zur Speicherung von $A \in \mathbb{R}^{m \times n}$
  - Dimensionen m, n vom Typ `int`
  - Datenfeld A_{ij} zur Speicherung von `double`

```
 1 #ifndef _STRUCT_MATRIX_
 2 #define _STRUCT_MATRIX_
 3
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <assert.h>
 7 #include <math.h>
 8 #include "struct_vector.h"
 9
10 typedef struct _Matrix_ {
11   int m;          // Dimension
12   int n;
13   double** entry; // Matrix entries
14 } Matrix;
15
16 // allocate and initialize m x n matrix
17 Matrix* newMatrix(int m, int n);
18
19 // free storage of allocated matrix and return NULL
20 Matrix* delMatrix(Matrix* A);
21
22 // set and get functions
23 int getMatrixM(Matrix* A);
24 int getMatrixN(Matrix* A);
25 double getMatrixEntry(Matrix* A, int i, int j);
26 void setMatrixEntry(Matrix* A, int i, int j, double Aij);
27
28 // compute matrix-vector multiplication
29 Vector* matrixVectorProduct(Matrix* A, Vector* X);
30
31 // compute row-sum norm of a matrix
32 double normMatrix(Matrix* A);
33
34 #endif
```

Allokieren einer Matrix

- Wir speichern die Einträge der Matrix als `double**`
  - Allokation der Einträge wie oben besprochen

```
 3 Matrix* newMatrix(int m, int n) {
 4   int i = 0;
 5   int j = 0;
 6   Matrix* A = NULL;
 7
 8   assert(m > 0);
 9   assert(n > 0);
10
11   A = malloc(sizeof(Matrix));
12   assert(A != NULL);
13
14   A->m = m;
15   A->n = n;
16   A->entry = malloc(m*sizeof(double*));
17   assert(A->entry != NULL);
18
19   for (i=0; i<m; ++i) {
20     A->entry[i] = malloc(n*sizeof(double));
21     assert(A->entry[i] != NULL);
22     for (j=0; j<n; ++j) {
23       A->entry[i][j] = 0;
24     }
25   }
26
27   return A;
28 }
```

Freigeben einer Matrix

- Erst Datenfeld `A->entry` freigeben
  - erst Zeilenvektoren freigeben
  - dann Spaltenvektor freigeben
- Dann Struktur freigeben

```
30 Matrix* delMatrix(Matrix* A) {
31   int i = 0;
32   assert(A != NULL);
33   assert(A->entry != NULL);
34
35   for (i=0; i<A->m; ++i) {
36     free(A->entry[i]);
37   }
38
39   free(A->entry);
40   free(A);
41
42   return NULL;
43 }
```

Zugriffsfunktionen

```
45 int getMatrixM(Matrix* A) {
46   assert(A != NULL);
47   return A->m;
48 }
49
50 int getMatrixN(Matrix* A) {
51   assert(A != NULL);
52   return A->n;
53 }
54
55 double getMatrixEntry(Matrix* A, int i, int j) {
56   assert(A != NULL);
57   assert((i >= 0) && (i < A->m));
58   assert((j >= 0) && (j < A->n));
59   return A->entry[i][j];
60 }
61
62 void setMatrixEntry(Matrix* A, int i, int j, double Aij) {
63   assert(A != NULL);
64   assert((i >= 0) && (i < A->m));
65   assert((j >= 0) && (j < A->n));
66   A->entry[i][j] = Aij;
67 }
```

Beispiel: Matrix-Vektor-Produkt

```
 69 Vector* matrixVectorProduct(Matrix* A, Vector* X) {
 70   double Aij = 0;
 71   double Xj = 0;
 72   double Bi = 0;
 73   int m = 0;
 74   int n = 0;
 75   int i = 0;
 76   int j = 0;
 77   Vector* B = NULL;
 78
 79   assert(A != NULL);
 80   assert(X != NULL);
 81
 82   m = getMatrixM(A);
 83   n = getMatrixN(A);
 84   assert(n == getVectorN(X));
 85
 86   B = newVector(m);
 87   assert(B != NULL);
 88
 89   for (i=0; i<m; ++i) {
 90     Bi = 0;
 91     for (j=0; j<n; ++j) {
 92       Aij = getMatrixEntry(A,i,j);
 93       Xj = getVectorEntry(X,j);
 94       Bi = Bi + Aij*Xj;
 95     }
 96     setVectorEntry(B,i,Bi);
 97   }
 98
 99   return B;
100 }
```

- Gegeben $A \in \mathbb{R}^{m \times n}$ und $x \in \mathbb{R}^n$
- Berechne $b \in \mathbb{R}^m$ mit $b_i = \sum_{j=1}^{n} A_{ij} x_j$

Beispiel: Zeilensummennorm

```
102 double normMatrix(Matrix* A) {
103   int m = 0;
104   int n = 0;
105   double Aij = 0;
106   double max = 0;
107   double sum = 0;
108   int i = 0;
109   int j = 0;
110
111   assert(A != NULL);
112
113   m = getMatrixM(A);
114   n = getMatrixN(A);
115
116   for (i=0; i<m; ++i) {
117     sum = 0;
118     for (j=0; j<n; ++j) {
119       Aij = getMatrixEntry(A,i,j);
120       sum = sum + fabs(Aij);
121     }
122     if (sum > max) {
123       max = sum;
124     }
125   }
126
127   return max;
128 }
```

- Gegeben $A \in \mathbb{R}^{m \times n}$
- Berechne $\|A\|_Z := \max_{i=1,\ldots,m} \sum_{j=1}^{n} |A_{ij}|$

Strukturen und Matrizen, v2

- Manchmal Fortran-Bib nötig, z.B. LAPACK
  - will auf `A->entry` Fortran-Routinen anwenden!
- Fortran speichert $A \in \mathbb{R}^{m \times n}$ spaltenweise in Vektor der Länge mn
  - A_{ij} entspricht `A[i+j*m]`, wenn $A \in \mathbb{R}^{m \times n}$

```
 7 typedef struct _Matrix_ {
 8   int m;
 9   int n;
10   double* entry;
11 } Matrix;
```

- Allokieren der neuen Matrix-Struktur

```
13 Matrix* newMatrix(int m, int n) {
14   int i = 0;
15   Matrix* A = NULL;
16
17   assert(m > 0);
18   assert(n > 0);
19
20   A = malloc(sizeof(Matrix));
21   assert(A != NULL);
22
23   A->m = m;
24   A->n = n;
25   A->entry = malloc(m*n*sizeof(double));
26   assert(A->entry != NULL);
27
28   for (i=0; i<m*n; ++i) {
29     A->entry[i] = 0;
30   }
31
32   return A;
33 }
```

Noch einmal free, set, get

- Freigeben der neuen Matrix-Struktur

```
35 Matrix* delMatrix(Matrix* A) {
36   assert(A != NULL);
37   assert(A->entry != NULL);
38
39   free(A->entry);
40   free(A);
41   return NULL;
42 }
```

- set und get für Matrix-Einträge

```
44 double getMatrixEntry(Matrix* A, int i, int j) {
45   assert(A != NULL);
46   assert((i >= 0) && (i < A->m));
47   assert((j >= 0) && (j < A->n));
48
49   return A->entry[i+j*A->m];
50 }
51
52 void setMatrixEntry(Matrix* A, int i, int j, double Aij) {
53   assert(A != NULL);
54   assert((i >= 0) && (i < A->m));
55   assert((j >= 0) && (j < A->n));
56
57   A->entry[i+j*A->m] = Aij;
58 }
```

- alle anderen Funktionen bleiben unverändert!
- Plötzlich werden set und get eine gute Idee
  - verhindert Fehler!
  - macht Programm im Nachhinein flexibel, z.B. bei Änderungen an Datenstruktur