Einführung in das Programmieren für Technische Mathematik

Prof. Dr. Dirk Praetorius

Dr. Michele Ruggeri

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

Institut für Analysis
und Scientific Computing

Formalia

- Rechte & Pflichten
- Benotung
- Anwesenheitspflicht
- Literatur

EPROG-Homepages

- http://www.asc.tuwien.ac.at/eprog/
  - alle Regeln & Pflichten
  - Benotungsschema
  - freiwilliges UE-Material (alte Tests!)
  - Evaluation & Notenspiegel

- http://tuwel.tuwien.ac.at/course/view.php?id=19714
  - alle Regeln & Pflichten
  - Benotungsschema
  - freiwilliges UE-Material (alte Tests!)
  - Download der Folien & Übungsblätter
  - Termine der VO und UE

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 TUWEL
- 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 $1 + \log_2 n$ 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 Zeitmessung?
  - 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?)
  - Tut das Programm (bzw. Teilabschnitte) das Richtige? (d.h. Ergebnis korrekt?)
  - Werden unzulässige Parameter erkannt?
  - 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?
- früh mit dem Testen beginnen
  - nach Implementierung jeder Funktion!
  - nicht erst dann, wenn Prg komplett fertig!
- nach Code-Korrektur alle(!) Tests wiederholen
  - deshalb Dokumentation der Tests!
- Ab jetzt in der UE stets: Wie wurde getestet?
  - allerdings nur inhaltlich
  - d.h. ohne Fehleingaben des Nutzers

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

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 [intmin, intmax]
- Überlauf: Ergebnis von Rechnung $>$ intmax
- Unterlauf: Ergebnis von Rechnung $<$ intmin
- Integer-Arithmetik ist i.d.R. Modulo-Arithmetik
  - d.h. Zahlenbereich ist geschlossen
    - intmax $+$ 1 liefert intmin
    - 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 gek. Brüche genau dann als endliche Summe darstellbar, wenn Nenner Zweierpotenz:
  - $\sum_{k=1}^{M} a_k 2^{-k}$ 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   double test;     // Punkte im Test
 8   double kurztest; // Punkte in Kurztests
 9   double 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   double test;     // Punkte im Test
 8   double kurztest; // Punkte in Kurztests
 9   double 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   double test;     // Punkte im Test
12   double kurztest; // Punkte in Kurztests
13   double 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 $x, y \in \mathbb{R}^n$

C++

- Was ist C++?
- Wie erstellt man ein C++ Programm?
- `Hello World!` mit C++

- `main`
- `cout`, `cin`, `endl`
- `using std::`
- Scope-Operator `::`
- Operatoren «, »
- `#include <iostream>`

Was ist C++

- Weiterentwicklung von C
  - Entwicklung ab 1979 bei AT&T
  - Entwickler: Bjarne Stroustrup
- C++ ist abwärtskompatibel zu C
  - keine Syntaxkorrektur
  - aber: stärkere Zugriffskontrolle bei „Strukturen“
    - Datenkapselung
- Compiler:
  - frei verfügbar in Unix/Mac: `g++`
  - Microsoft Visual C++ Compiler
  - Borland C++ Compiler

Objektorientierte Programmiersprache

- C++ ist objektorientiertes C
- Objekt = Zusammenfassung von Daten + Fktn.
  - Funktionalität hängt von Daten ab
  - vgl. Multiplikation für Skalar, Vektor, Matrix
- Befehlsreferenzen
  - http://en.cppreference.com/w/cpp
  - http://www.cplusplus.com

Wie erstellt man ein C++ Prg?

- 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.cpp`
  - Endung `.cpp` ist Kennung für C++ Programm
- Die ersten beiden Punkte kann man auch simultan erledigen mittels `emacs name.cpp &`
- Schreibe *Source-Code* (= C++ Programm)
- Abspeichern mittels `C-x C-s` nicht vergessen
- Compilieren z.B. mit `g++ name.cpp`
- Falls Code fehlerfrei, erhält man *Executable* `a.out`
  - unter Windows: `a.exe`
- Diese wird durch `a.out` bzw. `./a.out` gestartet
- Compilieren mit `g++ name.cpp -o output` erzeugt Executable `output` statt `a.out`

Hello World!

```
1 #include <iostream>
2
3 int main() {
4   std::cout << "Hello World!\n";
5   return 0;
6 }
```

- C++ Bibliothek für Ein- und Ausgabe ist iostream
- main hat zwingend Rückgabewert int
  - int main()
  - int main(int argc, char* argv[])
    - insbesondere return 0; am Programmende
- Scope-Operator :: gibt *Name Space* an
  - alle Fktn. der Standardbibliotheken haben std
- std::cout ist die Standard-Ausgabe (= Shell)
  - Operator << übergibt rechtes Argument an cout

```
1 #include <iostream>
2 using std::cout;
3
4 int main() {
5   cout << "Hello World!\n";
6   return 0;
7 }
```

- using std::cout;
  - cout gehört zum *Name Space* std
  - darf im Folgenden abkürzen cout statt std::cout

Shell-Input für Main

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 int main(int argc, char* argv[]) {
 6   int j = 0;
 7   cout << "This is " << argv[0] << endl;
 8   cout << "got " << argc-1 << " inputs:" << endl;
 9   for (j=1; j<argc; ++j) {
10     cout << j << ": " << argv[j] << endl;
11   }
12   return 0;
13 }
```

- `<<` arbeitet mit verschiedenen Typen
- kann mehrfache Ausgabe machen `<<`
- `endl` ersetzt `"\n"`
- Shell übergibt Input als C-Strings an Programm
  - Parameter jeweils durch Leerzeichen getrennt
  - argc = Anzahl der Parameter
  - argv = Vektor der Input-Strings
  - argv[0] = Programmname
  - d.h. argc−1 echte Input-Parameter
- Output für Shell-Eingabe `./a.out Hello World!`

```
This is ./a.out
got 2 inputs:
1: Hello
2: World!
```

Eingabe / Ausgabe

```
 1 #include <iostream>
 2 using std::cin;
 3 using std::cout;
 4 using std::endl;
 5
 6 int main() {
 7   int x = 0;
 8   double y = 0;
 9   double z = 0;
10
11   cout << "Geben Sie einen Integer ein: ";
12   cin >> x;
13   cout << "Geben Sie zwei Double ein: ";
14   cin >> y >> z;
15
16   cout << x << " * " << y << " / " << z;
17   cout << " = " << x*y/z << endl;
18
19   return 0;
20 }
```

- `std::cin` ist die Standard-Eingabe (= Tastatur)
  - Operator `>>` schreibt Input in Variable rechts
- Beispielhafte Eingabe / Ausgabe:

```
Geben Sie einen Integer ein: 2
Geben Sie zwei Double ein: 3.6 1.3
2 * 3.6 / 1.3 = 5.53846
```

- `cin` / `cout` gleichwertig mit `printf` / `scanf` in C
  - aber leichter zu bedienen
  - keine Platzhalter + Pointer
  - Formatierung, siehe `http://www.cplusplus.com`
    - ⟶ `ostream::operator<<`

Datentyp `bool`

- `bool`
- `true`
- `false`

Datentyp bool

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 int main() {
 5   double var = 0.3;
 6   bool tmp = var;
 7
 8   if (1) {
 9     cout << "1 ist wahr\n";
10   }
11   if (var) {
12     cout << var << " ist auch wahr\n";
13   }
14   if (tmp == true) {
15     cout << tmp << " ist auch wahr\n";
16     cout << "sizeof(bool) = " << sizeof(bool) << "\n";
17   }
18   if (0) {
19     cout << "0 ist wahr\n";
20   }
21
22   return 0;
23 }
```

- C kennt keinen Datentyp für Wahrheitswerte
  - logischer Vergleich liefert 1 für wahr, 0 für falsch
  - jede Zahl ungleich 0 wird als wahr interpretiert
- C++ hat Datentyp `bool` für Wahrheitswerte
  - Wert `true` für wahr, `false` für falsch
  - jede Zahl ungleich 0 wird als wahr interpretiert
- Output:

```
1 ist wahr
0.3 ist auch wahr
1 ist auch wahr
sizeof(bool) = 1
```

Klassen

- Klassen
- Instanzen
- Objekte

- `class`
- `struct`
- `private`, `public`
- `string`
- `#include <cmath>`
- `#include <cstdio>`
- `#include <string>`

Klassen & Objekte

- Klassen sind (benutzerdefinierte) Datentypen
  - erweitern `struct` aus C
  - bestehen aus Daten und Methoden
  - Methoden = Fktn. auf den Daten der Klasse
- Deklaration etc. wie bei Struktur-Datentypen
  - Zugriff auf Members über Punktoperator
  - sofern dieser Zugriff erlaubt ist!
    - Zugriffskontrolle = Datenkapselung
- formale Syntax: `class ClassName{ ... };`
- Objekte = Instanzen einer Klasse
  - entspricht Variablen dieses neuen Datentyps
  - wobei Methoden nur 1x im Speicher liegen
- später: Kann Methoden überladen
  - d.h. Funktionalität einer Methode abhängig von Art des Inputs
- später: Kann Operatoren überladen
  - z.B. $x + y$ für Vektoren
- später: Kann Klassen von Klassen ableiten
  - sog. Vererbung
  - z.B. $\mathbb{C} \supset \mathbb{R} \supset \mathbb{Q} \supset \mathbb{Z} \supset \mathbb{N}$
  - dann: $\mathbb{R}$ erbt Methoden von $\mathbb{C}$ etc.

Zugriffskontrolle

- Klassen (und Objekte) dienen der Abstraktion
  - genaue Implementierung nicht wichtig
- Benutzer soll so wenig wissen wie möglich
  - sogenannte *black-box* Programmierung
  - nur Ein- und Ausgabe müssen bekannt sein
- Richtiger Zugriff muss sichergestellt werden
- Schlüsselwörter `private`, `public` und `protected`
- `private` (Standard)
  - Zugriff nur von Methoden der gleichen Klasse
- `public`
  - erlaubt Zugriff von überall
- `protected`
  - teilweiser Zugriff von außen ($\rightsquigarrow$ Vererbung)

Beispiel 1/2

```
 1 class Triangle {
 2 private:
 3   double x[2];
 4   double y[2];
 5   double z[2];
 6
 7 public:
 8   void setX(double, double);
 9   void setY(double, double);
10   void setZ(double, double);
11   double area();
12 };
```

- Dreieck in $\mathbb{R}^2$ mit Eckpunkten `x`, `y`, `z`
- Benutzer kann Daten `x`,`y`,`z` nicht lesen + schreiben
  - `get`/`set` Funktionen in `public`-Bereich einbauen
- Benutzer kann Methode `area` aufrufen
- Benutzer muss nicht wissen, wie Daten intern verwaltet werden
  - kann interne Datenstruktur später leichter verändern, falls das nötig wird
  - z.B. Dreieck kann auch durch einen Punkt und zwei Vektoren abgespeichert werden
- Zeile 2: `private:` kann weggelassen werden
  - alle Members/Methoden standardmäßig `private`
- Zeile 7: ab `public:` ist Zugriff frei
  - d.h. Zeile 8 und folgende

Beispiel 2/2

```
 1 class Triangle {
 2 private:
 3   double x[2];
 4   double y[2];
 5   double z[2];
 6
 7 public:
 8   void setX(double, double);
 9   void setY(double, double);
10   void setZ(double, double);
11   double getArea();
12 };
13
14 int main() {
15   Triangle tri;
16
17   tri.x[0] = 1.0; // Syntax-Fehler!
18
19   return 0;
20 }
```

- Zeile 8–11: Deklaration von `public`-Methoden
- Zeile 15: Objekt `tri` vom Typ `Triangle` deklarieren
- Zeile 17: Zugriff auf `private`-Member
- Beim Kompilieren tritt Fehler auf

  ```
  triangle2.cpp:17: error: 'x' is a private
  member of 'Triangle'
  triangle2.cpp:3: note: declared private here
  ```

- daher: `get/set`-Funktionen, falls nötig

Methoden implementieren 1/2

```
 1 #include <cmath>
 2
 3 class Triangle {
 4 private:
 5   double x[2];
 6   double y[2];
 7   double z[2];
 8 public:
 9   void setX(double, double);
10   void setY(double, double);
11   void setZ(double, double);
12   double getArea();
13 };
14
15 double Triangle::getArea() {
16   return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
17                    - (z[0]-x[0])*(y[1]-x[1]) );
18 }
```

- Implementierung wie bei anderen Funktionen
  - direkter Zugriff auf Members der Klasse
- Signatur: `type ClassName::fctName(input)`
  - `type` ist Rückgabewert (`void`, `double` etc.)
  - `input` = Übergabeparameter wie in C
- Wichtig: `ClassName::` vor `fctName`
  - d.h. Methode `fctName` gehört zu `ClassName`
- Darf innerhalb von `ClassName::fctName` auf alle Members der Klasse direkt zugreifen (Zeile 16–17)
  - auch auf `private`-Members
- Zeile 1: Einbinden der `math.h` aus C

Methoden implementieren 2/2

```
 1 #include <cmath>
 2
 3 class Triangle {
 4 private:
 5   double x[2];
 6   double y[2];
 7   double z[2];
 8
 9 public:
10   void setX(double, double);
11   void setY(double, double);
12   void setZ(double, double);
13   double getArea();
14 };
15
16 void Triangle::setX(double x0, double x1) {
17   x[0] = x0; x[1] = x1;
18 }
19
20 void Triangle::setY(double y0, double y1) {
21   y[0] = y0; y[1] = y1;
22 }
23
24 void Triangle::setZ(double z0, double z1) {
25   z[0] = z0; z[1] = z1;
26 }
27
28 double Triangle::getArea() {
29   return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
30                    - (z[0]-x[0])*(y[1]-x[1]) );
31 }
```

Methoden aufrufen

```
 1 #include <iostream>
 2 #include "triangle4.cpp" // Code von letzter Folie
 3
 4 using std::cout;
 5 using std::endl;
 6
 7 // void Triangle::setX(double x0, double x1)
 8 // void Triangle::setY(double y0, double y1)
 9 // void Triangle::setZ(double z0, double z1)
10
11 // double Triangle::getArea() {
12 //     return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
13 //                    - (z[0]-x[0])*(y[1]-x[1]) );
14 // }
15
16 int main() {
17   Triangle tri;
18   tri.setX(0.0,0.0);
19   tri.setY(1.0,0.0);
20   tri.setZ(0.0,1.0);
21   cout << "Flaeche = " << tri.getArea() << endl;
22   return 0;
23 }
```

- Aufruf wie Member-Zugriff bei C-Strukturen
  - wäre in C über Funktionspointer analog möglich
- `getArea` agiert auf den Members von `tri`
  - d.h. `x[0]` in Implementierung entspricht `tri.x[0]`
- **Output:** `Flaeche = 0.5`

Methoden direkt implementieren

```
 1 #include <cmath>
 2
 3 class Triangle {
 4 private:
 5   double x[2];
 6   double y[2];
 7   double z[2];
 8
 9 public:
10   void setX(double x0, double x1) {
11     x[0] = x0;
12     x[1] = x1;
13   }
14   void setY(double y0, double y1) {
15     y[0] = y0;
16     y[1] = y1;
17   }
18   void setZ(double z0, double z1) {
19     z[0] = z0;
20     z[1] = z1;
21   }
22   double getArea() {
23     return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
24                      - (z[0]-x[0])*(y[1]-x[1]) );
25   }
26 };
```

- kann Methoden auch in Klasse implementieren
- ist aber unübersichtlicher $\Rightarrow$ besser nicht!

Klasse `string`

```
 1 #include <iostream>
 2 #include <string>
 3 #include <cstdio>
 4 using std::cout;
 5 using std::string;
 6
 7 int main() {
 8   string str1 = "Hallo";
 9   string str2 = "Welt";
10   string str3 = str1 + " " + str2;
11
12   cout << str3 << "! ";
13   str3.replace(6,4, "Peter");
14   cout << str3 << "! ";
15
16   printf("%s?\n",str3.c_str());
17
18   return 0;
19 }
```

- **Output:** `Hallo Welt! Hallo Peter! Hallo Peter?`
- Zeile 3: Einbinden der `stdio.h` aus C
- Wichtig: `string` $\neq$ `char*`, sondern mächtiger!
- liefert eine Reihe nützlicher Methoden
  - '`+`' zum Zusammenfügen
  - `replace` zum Ersetzen von Teilstrings
  - `length` zum Auslesen der Länge u.v.m.
  - `c_str` liefert Pointer auf `char*`
- http://www.cplusplus.com/reference/string/string/

Strukturen

```
 1 struct MyStruct {
 2   double x[2];
 3   double y[2];
 4   double z[2];
 5 };
 6
 7 class MyClass {
 8   double x[2];
 9   double y[2];
10   double z[2];
11 };
12
13 class MyStructClass {
14 public:
15   double x[2];
16   double y[2];
17   double z[2];
18 };
19
20 int main() {
21   MyStruct var1;
22   MyClass var2;
23   MyStructClass var3;
24
25   var1.x[0] = 0;
26   var2.x[0] = 0; // Syntax-Fehler
27   var3.x[0] = 0;
28
29   return 0;
30 }
```

- Strukturen = Klassen, wobei alle Members `public`
  - d.h. `MyStruct` = `MyStructClass`
- besser direkt `class` verwenden

Funktionen

- Default-Parameter & Optionaler Input
- Überladen

Default-Parameter 1/2

```
1  void f(int x, int y, int z = 0);
2  void g(int x, int y = 0, int z = 0);
3  void h(int x = 0, int y = 0, int z = 0);
```

- kann Default-Werte für Input von Fktn. festlegen
  - durch `= wert`
  - der Input-Parameter ist dann optional
  - bekommt Default-Wert, falls nicht übergegeben

- Beispiel: Zeile 1 erlaubt Aufrufe
  - `f(x,y,z)`
  - `f(x,y)` und `z` bekommt implizit den Wert `z = 0`

```
1  void f(int x = 0, int y = 0, int z); // Fehler
2  void g(int x, int y = 0, int z);     // Fehler
3  void h(int x = 0, int y, int z = 0); // Fehler
```

- darf nur für hintere Parameter verwendet werden
  - d.h. nach optionalem Parameter darf kein obligatorischer Parameter mehr folgen

Default-Parameter 2/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 void f(int x, int y = 0);
 5
 6 void f(int x, int y = 0) {
 7     cout << "x=" << x << ", y=" << y << "\n";
 8 }
 9
10 int main() {
11     f(1);
12     f(1,2);
13     return 0;
14 }
```

- Default-Parameter darf nur einmal gegeben werden
- Kompilieren liefert Syntax-Fehler:

  `default_wrong.cpp:6: error: redefinition of default argument`
- d.h. Default-Parameter nur in Zeile 4 definieren!
- Output nach Korrektur:

  `x=1, y=0`

  `x=1, y=2`
- **Konvention:**
  - d.h. Default-Parameter werden in `hpp` festgelegt
- brauche bei Forward Decl. keine Variablennamen
  - `void f(int, int = 0);` in Zeile 4 ist OK

Überladen von Funktionen 1/2

```
1  void f(char*);
2  double f(char*, double);
3  int f(char*, char*, int = 1);
4  int f(char*);                    // Syntax-Fehler
5  double f(char*, int = 0);        // Syntax-Fehler
```

- Mehrere Funktionen gleichen Namens möglich
  - unterscheiden sich durch ihre Signaturen
- Input muss Variante eindeutig festlegen
- bei Aufruf wird die richtige Variante ausgewählt
  - Compiler erkennt dies über Input-Parameter
  - Achtung mit implizitem Type Cast
- Diesen Vorgang nennt man Überladen
- Reihenfolge bei der Deklaration ist unwichtig
  - d.h. kann Zeilen 1–3 beliebig permutieren
- Rückgabewerte können unterschiedlich sein
  - Also: unterschiedliche Output-Parameter und gleiche Input-Parameter geht nicht
    - Zeile 1 + 2 + 3: OK
    - Zeile 4: Syntax-Fehler, da Input gleich zu 1
    - Zeile 5: Syntax-Fehler, da optionaler Input

Überladen von Funktionen 2/2

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 class Car {
 6 public:
 7   void drive();
 8   void drive(int km);
 9   void drive(int km, int h);
10 };
11
12 void Car::drive() {
13   cout << "10 km gefahren" << endl;
14 }
15
16 void Car::drive(int km) {
17   cout << km << " km gefahren" << endl;
18 }
19
20 void Car::drive(int km, int h) {
21   cout << km << " km gefahren in " << h
22     << " Stunde(n)" << endl;
23 }
24
25 int main() {
26   Car TestCar;
27   TestCar.drive();
28   TestCar.drive(35);
29   TestCar.drive(50,1);
30   return 0;
31 }
```

- Ausgabe:
```
10 km gefahren
35 km gefahren
50 km gefahren in 1 Stunde(n)
```

Überladen vs. Default-Parameter

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 class Car {
 6 public:
 7   void drive(int km = 10, int h = 0);
 8 };
 9
10 void Car::drive(int km, int h) {
11   cout << km << " km gefahren";
12   if (h > 0) {
13     cout << " in " << h << " Stunde(n)";
14   }
15   cout << endl;
16 }
17
18 int main() {
19   Car TestCar;
20   TestCar.drive();
21   TestCar.drive(35);
22   TestCar.drive(50,1);
23   return 0;
24 }
```

- Ausgabe:
```
10 km gefahren
35 km gefahren
50 km gefahren in 1 Stunde(n)
```

Naive Fehlerkontrolle

- Wozu Zugriffskontrolle?
- Vermeidung von Laufzeitfehlern!
- bewusster Fehlerabbruch

- `assert`
- `#include <cassert>`

Wozu Zugriffskontrolle?

```
 1 class Fraction {
 2 public:
 3   int numerator;
 4   int denominator;
 5 };
 6
 7 int main() {
 8   Fraction x;
 9   x.numerator = -1000;
10   x.denominator = 0;
11
12   return 0;
13 }
```

- Großteil der Entwicklungszeit geht in Fehlersuche von Laufzeitfehlern!
- Möglichst viele Fehler bewusst abfangen!
  - Fkt-Input auf Konsistenz prüfen, ggf. Abbruch
  - garantieren, dass Funktions-Output zulässig!
  - Zugriff kontrollieren mittels `get` und `set`
    - reine Daten sollten immer `private` sein
    - Benutzer kann/darf Daten nicht verpfuschen!
    - in C = soll nicht, in C++ = kann nicht!
- Wie sinnvolle Werte sicherstellen? (Zeile 10)
  - mögliche Fehlerquellen direkt ausschließen
  - Programm bestimmt, was Nutzer darf!
- kontrollierter Abbruch mit C-Bibliothek `assert.h`
  - Einbinden `#include <cassert>`
  - Abbruch mit Ausgabe der Zeile im Source-Code

C-Bibliothek assert.h

```
 1 #include <iostream>
 2 #include <cassert>
 3 using std::cout;
 4
 5 class Fraction {
 6 private:
 7   int numerator;
 8   int denominator;
 9 public:
10   int getNumerator() { return numerator; };
11   int getDenominator() { return denominator; };
12   void setNumerator(int n) { numerator = n; };
13   void setDenominator(int n) {
14     assert(n != 0);
15     if (n > 0) {
16       denominator = n;
17     }
18     else {
19       denominator = -n;
20       numerator = -numerator;
21     }
22   }
23   void print() {
24     cout << numerator << "/" << denominator << "\n";
25   }
26 };
27
28 int main() {
29   Fraction x;
30   x.setNumerator(1);
31   x.setDenominator(3);
32   x.print();
33   x.setDenominator(0);
34   return 0;
35 }
```

- `assert(condition);` bricht ab, falls `condition` falsch
- Output:

```
1/3
Assertion failed: (n>0), function setDenominator,
file assert.cpp, line 14.
```

Konventionen

- Namens-Konventionen
- Deklaration von Variablen
- File-Konventionen

- `for(int j=0; j<dim; ++j) { ... }`

Namens-Konventionen

- lokale Variablen
  - `klein_mit_underscores`
- globale Variablen
  - `klein_mit_underscore_hinten_`
- Präprozessor-Konstanten
  - `GROSS_MIT_UNDERSCORE`
- in Header-Files
  - `_NAME_DER_KLASSE_`
- Funktionen / Methoden
  - `erstesWortKleinKeineUnderscores`
- Strukturen / Klassen
  - `ErstesWortGrossKeineUnderscores`

Variablen-Deklaration

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 int main() {
 6   double sum = 0;
 7
 8   for (int j=1; j<=100; ++j) {
 9     sum = sum + j;
10   }
11
12   cout << sum << endl;
13 }
```

- in C++ überall erlaubt, aber schlechter Stil!
  - wird schnell unübersichtlich!
- Konvention: Deklaration nur am Blockanfang
  - ist übersichtlicher!
- zwei Ausnahmen:
  - Zählvariable bei `for`-Schleife
    - üblicherweise im Schleifen-Kopf deklariert
    - ist lokale Variable, bis Schleife terminiert
  - `assert` vor Deklaration ist OK!
- Beispiel-Code berechnet $\sum_{j=1}^{100} j = 5050$
  - Zählvariable `j` existiert nur in Zeile 8–10

Schlechter Code 1/2

```
 1 #include <stdio.h>
 2
 3 int main() {
 4   int a[2] = {0, 1};
 5   int b[2] = {2, 3};
 6   int c[3] = {4, 5};
 7   int i = 0;
 8
 9   printf("a = (%d,%d), b = (%d,%d), c = (%d,%d), i = %d\n",
10          a[0], a[1], b[0], b[1], c[0], c[1], i);
11
12   a[i] = b[i] = c[i];
13
14   printf("a = (%d,%d), b = (%d,%d), c = (%d,%d), i = %d\n",
15          a[0], a[1], b[0], b[1], c[0], c[1], i);
16
17   c[0] = 9;
18   i = 0;
19
20   a[i] = b[i++] = c[i];
21
22   printf("a = (%d,%d), b = (%d,%d), c = (%d,%d), i = %d\n",
23          a[0], a[1], b[0], b[1], c[0], c[1], i);
24
25   return 0;
26 }
```

- schlecht: Nicht jede Zeile sofort verständlich!
- **Achtung:** Verhalten von `b[i++]` ist undefiniert!

  ```
  warning: unsequenced modification and access
  to 'i'
  ```

- **faktischer Output:**

  ```
  a = (0,1), b = (2,3), c = (4,5), i = 0
  a = (4,1), b = (4,3), c = (4,5), i = 0
  a = (4,9), b = (9,3), c = (9,5), i = 1
  ```

Schlechter Code 2/2

```
 1 #include <cstdlib>
 2 #include <cstdio>
 3 int main(){
 4   int i=0;
 5   int n=5;
 6   int* a=(int*)malloc((n+1)*sizeof(int));
 7   int*b=(int*)malloc((n+1)*sizeof(int));
 8   int *c=(int*)malloc((n+1)*sizeof(int));
 9   int * d=(int*)malloc((n+1)*sizeof(int));
10   while(i<n){
11     a[i]=b[i]=c[i]=d[i]=i++;}
12   printf("a[%d] = %d\n",n-1,n-1);
13 }
```

- Code für menschliches Auge schreiben!
  - Leerzeichen vor/nach Zuweisungen
  - Leerzeichen vor/nach Type-Cast-Operator
  - (manchmal) Leerzeichen vor/nach arithm. Op.
  - (manchmal) Leerzeichen vor/nach Klammern, wenn Klammern geschachtelt werden
  - Leerzeilen dort, wo gedankliche Blöcke
    - Deklarationen / Speicher anlegen / Aktionen
- Guter Code hat nur eine Aktion pro Zeile!
  - Deshalb Mehrfachzuweisungen schlecht, aber dennoch (leider) in C/C++ möglich!
- Zählschleifen, falls Laufzeit klar!
  - auch wenn defacto `for` = `while` in C

Besser lesbar!

```
 1 #include <cstdlib>
 2 #include <cstdio>
 3
 4 int main(){
 5   int n = 5;
 6
 7   int* a = (int*) malloc( (n+1)*sizeof(int) );
 8   int* b = (int*) malloc( (n+1)*sizeof(int) );
 9   int* c = (int*) malloc( (n+1)*sizeof(int) );
10   int* d = (int*) malloc( (n+1)*sizeof(int) );
11
12   for(int i=0; i<n; ++i){
13     a[i] = i - 1;
14     b[i] = i - 1;
15     c[i] = i - 1;
16     d[i] = i - 1;
17   }
18
19   printf("a[%d] = %d\n",n-1,a[n-1]);
20 }
```

- Code für menschliches Auge schreiben!
  - Leerzeichen vor/nach Zuweisungen
  - Leerzeichen vor/nach Type-Cast-Operator
  - (manchmal) Leerzeichen vor/nach arithm. Op.
  - (manchmal) Leerzeichen vor/nach Klammern, wenn Klammern geschachtelt werden
  - Leerzeilen dort, wo gedankliche Blöcke
    - Deklarationen / Speicher anlegen / Aktionen
- Guter Code hat nur eine Aktion pro Zeile!
  - schlecht: `b = ++a;`
  - schlecht: `a = b = c;`
- Zählschleifen, falls Laufzeit klar!
  - auch wenn defacto `for` = `while` in C

File-Konventionen

- Jedes C++ Programm besteht aus mehreren Files
  - C++ File für das Hauptprogramm `main.cpp`
  - Konvention: pro verwendeter Klasse zusätzlich
    - Header-File `myClass.hpp`
    - Source-File `myClass.cpp`
- Header-File `myClass.hpp` besteht aus
  - `#include` aller benötigten Bibliotheken
  - Definition der Klasse
  - nur Signaturen der Methoden (ohne Rumpf)
  - Kommentare zu den Methoden
    - Was tut eine Methode?
    - Was ist Input? Was ist Output?
    - insb. Default-Parameter + optionaler Input
- `myClass.cpp` enthält Source-Code der Methoden
- Warum Code auf mehrere Files aufteilen?
  - Übersichtlichkeit & Verständlichkeit des Codes
  - Anlegen von Bibliotheken
- Header-File beginnt mit

```
#ifndef _MY_CLASS_
#define _MY_CLASS_
```

- Header-File endet mit

```
#endif
```

- Dieses Vorgehen erlaubt mehrfache Einbindung!
- Wichtig: Kein `using` im Header verwenden!
  - insb. auch kein `using std::`...

triangle.hpp

```
 1 #ifndef _TRIANGLE_
 2 #define _TRIANGLE_
 3
 4 #include <cmath>
 5
 6 // The class Triangle stores a triangle in R2
 7
 8 class Triangle {
 9 private:
10   // the coordinates of the nodes
11   double x[2];
12   double y[2];
13   double z[2];
14
15 public:
16   // define or change the nodes of a triangle,
17   // e.g., triangle.setX(x1,x2) writes the
18   // coordinates of the node x of the triangle.
19   void setX(double, double);
20   void setY(double, double);
21   void setZ(double, double);
22
23   // return the area of the triangle
24   double getArea();
25 };
26
27 #endif
```

triangle.cpp

```
 1 #include "triangle.hpp"
 2
 3 void Triangle::setX(double x0, double x1) {
 4     x[0] = x0; x[1] = x1;
 5 }
 6
 7 void Triangle::setY(double y0, double y1) {
 8     y[0] = y0; y[1] = y1;
 9 }
10
11 void Triangle::setZ(double z0, double z1) {
12     z[0] = z0; z[1] = z1;
13 }
14
15 double Triangle::getArea() {
16     return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
17                     - (z[0]-x[0])*(y[1]-x[1]) );
18 }
```

- Erzeuge Objekt-Code aus Source (Option `-c`)
  - `g++ -c triangle.cpp` liefert `triangle.o`
- Kompilieren `g++ triangle.cpp` liefert Fehler
  - Linker `ld` scheitert, da kein `main` vorhanden

```
Undefined symbols for architecture x86_64:
"_main", referenced from:
   implicit entry/start for main executable
ld: symbol(s) not found for architecture x86_64
```

triangle_main.cpp

```
 1 #include <iostream>
 2 #include "triangle.hpp"
 3
 4 using std::cout;
 5 using std::endl;
 6
 7 int main() {
 8   Triangle tri;
 9   tri.setX(0.0,0.0);
10   tri.setY(1.0,0.0);
11   tri.setZ(0.0,1.0);
12   cout << "Flaeche = " << tri.getArea() << endl;
13   return 0;
14 }
```

- Kompilieren mit `g++ triangle_main.cpp triangle.o`
  - erzeugt Objekt-Code aus `triangle_main.cpp`
  - bindet zusätzlichen Objekt-Code `triangle.o` ein
  - linkt den Code inkl. Standardbibliotheken
- oder Verwendung von `make` analog zu C

Konstruktor & Destruktor

- Konstruktor
- Destruktor
- Überladen von Methoden
- optionaler Input & Default-Parameter
- Schachtelung von Klassen

- `this`
- `ClassName(...)`
- `~ClassName()`
- Operator :

Konstruktor & Destruktor

- Konstruktor = Aufruf automatisch bei Deklaration
  - kann Initialisierung übernehmen
  - kann verschiedene Aufrufe haben, z.B.
    - Anlegen eines Vektors der Länge Null
    - Anlegen eines Vektors $x \in \mathbb{R}^N$ und Initialisieren mit Null
    - Anlegen eines Vektors $x \in \mathbb{R}^N$ und Initialisieren mit gegebenem Wert
  - formal: `className(input)`
    - kein Output, eventuell Input
    - versch. Konstruktoren haben versch. Input
    - Standardkonstruktor: `className()`

- Destruktor = Aufruf automat. bei Lifetime-Ende
  - Freigabe von dynamischem Speicher
  - es gibt nur Standarddestruktor: `~className()`
    - kein Input, kein Output

- Methode kann überladen werden, z.B. Konstruktor
  - kein Input $\Rightarrow$ Vektor der Länge Null
  - ein Input `dim` $\Rightarrow$ Null-Vektor der Länge `dim`
  - Input `dim`, `val` $\Rightarrow$ Vektor der Länge `dim` mit Einträgen `val`

Konstruktor: Ein Beispiel

```
 1 #include <iostream>
 2 #include <string>
 3 using std::cout;
 4 using std::string;
 5
 6 class Student {
 7 private:
 8   string lastname;
 9   int student_id;
10 public:
11   Student() {
12     cout << "Student generiert\n";
13   }
14   Student(string name, int id) {
15     lastname = name;
16     student_id = id;
17     cout << "Student (" << lastname << ", ";
18     cout << student_id << ") angemeldet\n";
19   }
20 };
21
22 int main() {
23   Student demo;
24   Student var("Praetorius",12345678);
25   return 0;
26 }
```

- Konstruktor hat keinen Rückgabewert (Z. 11, 14)
  - Name `className(input)`
  - Standardkonstr. `Student()` ohne Input (Z. 11)
- Output

  ```
  Student generiert
  Student (Praetorius, 12345678) angemeldet
  ```

Namenskonflikt & Pointer this

```
 1 #include <iostream>
 2 #include <string>
 3 using std::cout;
 4 using std::string;
 5
 6 class Student {
 7 private:
 8   string lastname;
 9   int student_id;
10 public:
11   Student() {
12     cout << "Student generiert\n";
13   }
14   Student(string lastname, int student_id) {
15     this->lastname = lastname;
16     this->student_id = student_id;
17     cout << "Student (" << lastname << ", ";
18     cout << student_id << ") angemeldet\n";
19   }
20 };
21
22 int main() {
23   Student demo;
24   Student var("Praetorius",12345678);
25   return 0;
26 }
```

- `this` gibt Pointer auf das aktuelle Objekt
  - `this->` gibt Zugriff auf Member des akt. Objekts
- Namenskonflikt in Konstruktor (Zeile 14)
  - Input-Variable heißen wie Members der Klasse
  - Zeile 14–16: Lösen des Konflikts mittels `this->`

Destruktor: Ein Beispiel

```
 1 #include <iostream>
 2 #include <string>
 3 using std::cout;
 4 using std::string;
 5
 6 class Student {
 7 private:
 8   string lastname;
 9   int student_id;
10 public:
11   Student() {
12     cout << "Student generiert\n";
13   }
14   Student(string lastname, int student_id) {
15     this->lastname = lastname;
16     this->student_id = student_id;
17     cout << "Student (" << lastname << ", ";
18     cout << student_id << ") angemeldet\n";
19   }
20   ~Student() {
21     cout << "Student (" << lastname << ", ";
22     cout << student_id << ") abgemeldet\n";
23   }
24 };
25
26 int main() {
27   Student var("Praetorius",12345678);
28   return 0;
29 }
```

- Zeile 20–23: Destruktor (ohne Input + Output)
- Output

  ```
  Student (Praetorius, 12345678) angemeldet
  Student (Praetorius, 12345678) abgemeldet
  ```

Methoden: Kurzschreibweise

```
 1 #include <iostream>
 2 #include <string>
 3 using std::cout;
 4 using std::string;
 5
 6 class Student {
 7 private:
 8   string lastname;
 9   int student_id;
10 public:
11   Student() : lastname("nobody"), student_id(0) {
12     cout << "Student generiert\n";
13   }
14   Student(string name, int id) :
15     lastname(name), student_id(id) {
16       cout << "Student (" << lastname << ", ";
17       cout << student_id << ") angemeldet\n";
18   }
19   ~Student() {
20     cout << "Student (" << lastname << ", ";
21     cout << student_id << ") abgemeldet\n";
22   }
23 };
24
25 int main() {
26   Student test;
27   return 0;
28 }
```

- Zeile 11, 14–15: Kurzschreibweise für Zuweisung
  - ruft entsprechende Konstruktoren auf
  - eher schlecht lesbar
- Output

  ```
  Student generiert
  Student (nobody, 0) abgemeldet
  ```

Noch ein Beispiel

```
 1 #include <iostream>
 2 #include <string>
 3 using std::cout;
 4 using std::string;
 5
 6 class Test {
 7 private:
 8   string name;
 9 public:
10   void print() {
11     cout << "Name " << name << "\n";
12   }
13   Test() : name("Standard") { print(); }
14   Test(string n) : name(n)  { print(); }
15   ~Test() {
16     cout << "Loesche " << name << "\n";
17   }
18 };
19
20 int main() {
21   Test t1("Objekt1");
22   {
23     Test t2;
24     Test t3("Objekt3");
25   }
26   cout << "Blockende" << "\n";
27   return 0;
28 }
```

- Ausgabe:

```
Name Objekt1
Name Standard
Name Objekt3
Loesche Objekt3
Loesche Standard
Blockende
Loesche Objekt1
```

Schachtelung von Klassen

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 class Class1 {
 6 public:
 7   Class1() { cout << "Konstr Class1" << endl; }
 8   ~Class1() { cout << "Destr Class1" << endl; }
 9 };
10
11 class Class2 {
12 private:
13    Class1 obj1;
14 public:
15    Class2() { cout << "Konstr Class2" << endl; }
16    ~Class2() { cout << "Destr Class2" << endl; }
17 };
18
19 int main() {
20   Class2 obj2;
21   return 0;
22 }
```

- Klassen können geschachtelt werden
  - Standardkonstr./-destr. automatisch aufgerufen
  - Konstruktoren der Member zuerst
  - Destruktoren der Member zuletzt
- Ausgabe:

```
Konstr Class1
Konstr Class2
Destr Class2
Destr Class1
```

vector_first.hpp

```
 1 #ifndef _VECTOR_FIRST_
 2 #define _VECTOR_FIRST_
 3
 4 #include <cmath>
 5 #include <cstdlib>
 6 #include <cassert>
 7 #include <iostream>
 8
 9 // The class Vector stores vectors in Rd
10
11 class Vector {
12 private:
13   // dimension of the vector
14   int dim;
15   // dynamic coefficient vector
16   double* coeff;
17
18 public:
19   // constructors and destructor
20   Vector();
21   Vector(int dim, double init = 0);
22   ~Vector();
23
24   // return vector dimension
25   int size();
26
27   // read and write vector coefficients
28   void set(int k, double value);
29   double get(int k);
30
31   // compute Euclidean norm
32   double norm();
33 };
34
35 #endif
```

vector_first.cpp 1/2

```
 1 #include "vector_first.hpp"
 2
 3 Vector::Vector() {
 4   dim = 0;
 5   coeff = (double*) 0;
 6   std::cout << "allocate empty vector" << "\n";
 7 }
 8
 9 Vector::Vector(int dim, double init) {
10   assert(dim>0);
11   this->dim = dim;
12   coeff = (double*) malloc(dim*sizeof(double));
13   assert(coeff != (double*) 0);
14   for (int j=0; j<dim; ++j) {
15     coeff[j] = init;
16   }
17   std::cout << "allocate vector, length " << dim << "\n";
18 }
```

- erstellt drei Konstruktoren (Zeile 3, Zeile 9)
  - Standardkonstruktor (Zeile 3)
  - Deklaration `Vector var(dim,init);`
  - Deklaration `Vector var(dim);` mit `init = 0`
  - opt. Input durch Default-Parameter (Zeile 9)
    - wird in `vector.hpp` angegeben (letzte Folie!)
- **Achtung:** `g++` erfordert expliziten Type Cast bei Pointern, z.B. `malloc` (Zeile 12)
- in C++ darf man Variablen überall deklarieren
  - ist kein guter Stil, da unübersichtlich
    - im ursprünglichen C nur am Blockanfang
    - C-Stil möglichst beibehalten! Code wartbarer!
- **vernünftig:** `for (int j=0; j<dim; ++j) { ... }`
  - für lokale Zählvariablen (in Zeile 14)

vector_first.cpp 2/2

```
 9 Vector::Vector(int dim, double init) {
10   assert(dim>0);
11   this->dim = dim;
12   coeff = (double*) malloc(dim*sizeof(double));
13   assert(coeff != (double*) 0);
14   for (int j=0; j<dim; ++j) {
15     coeff[j] = init;
16   }
17   std::cout << "allocate vector, length " << dim << "\n";
18 }
19
20 Vector::~Vector() {
21   if (dim > 0) {
22     free(coeff);
23   }
24   std::cout << "free vector, length " << dim << "\n";
25 }
26
27 int Vector::size() {
28   return dim;
29 }
30
31 void Vector::set(int k, double value) {
32   assert(k>=0 && k<dim);
33   coeff[k] = value;
34 }
35
36 double Vector::get(int k) {
37   assert(k>=0 && k<dim);
38   return coeff[k];
39 }
40
41 double Vector::norm() {
42   double norm = 0;
43   for (int j=0; j<dim; ++j) {
44     norm = norm + coeff[j]*coeff[j];
45   }
46   return sqrt(norm);
47 }
```

- **ohne Destruktor:** nur Speicher von Pointer frei

main.cpp

```
 1 #include "vector_first.hpp"
 2 #include <iostream>
 3
 4 using std::cout;
 5
 6 int main() {
 7   Vector vector1;
 8   Vector vector2(20);
 9   Vector vector3(100,4);
10   cout << "Norm = " << vector1.norm() << "\n";
11   cout << "Norm = " << vector2.norm() << "\n";
12   cout << "Norm = " << vector3.norm() << "\n";
13
14   return 0;
15 }
```

- Kompilieren mit

```
g++ -c vector_first.cpp
g++ main.cpp vector_first.o
```

- Output:

```
allocate empty vector
allocate vector, length 20
allocate vector, length 100
Norm = 0
Norm = 0
Norm = 40
free vector, length 100
free vector, length 20
free vector, length 0
```

Referenzen

- Definition
- Unterschied zwischen Referenz und Pointer
- direktes Call by Reference
- Referenzen als Funktions-Output

- `type&`

Was ist eine Referenz?

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 int main() {
 6   int var = 5;
 7   int& ref = var;
 8
 9   cout << "var = " << var << endl;
10   cout << "ref = " << ref << endl;
11   ref = 7;
12   cout << "var = " << var << endl;
13   cout << "ref = " << ref << endl;
14
15   return 0;
16 }
```

- Referenzen sind Aliasnamen für Objekte/Variablen
- `type& ref = var;`
  - erzeugt eine Referenz `ref` zu `var`
  - `var` muss vom Datentyp `type` sein
  - Referenz muss bei Definition initialisiert werden!
- nicht verwechselbar mit Address-Of-Operator
  - `type&` ist Referenz
  - `&var` liefert Speicheradresse von `var`
- Output:

```
var = 5
ref = 5
var = 7
ref = 7
```

Address-Of-Operator

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 int main() {
 6   int var = 5;
 7   int& ref = var;
 8
 9   cout << "var = " << var << endl;
10   cout << "ref = " << ref << endl;
11   cout << "Adresse von var = " << &var << endl;
12   cout << "Adresse von ref = " << &ref << endl;
13
14   return 0;
15 }
```

- muss: Deklaration + Init. bei Referenzen (Zeile 7)
  - sind nur Alias-Name für denselben Speicher
  - d.h. `ref` und `var` haben dieselbe Adresse
- Output:

```
var = 5
ref = 5
Adresse von var = 0x7fff532e8b48
Adresse von ref = 0x7fff532e8b48
```

Funktionsargumente als Pointer

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 void swap(int* px, int* py) {
 6   int tmp = *px;
 7   *px = *py;
 8   *py = tmp;
 9 }
10
11 int main() {
12   int x = 5;
13   int y = 10;
14   cout << "x = " << x << ", y = " << y << endl;
15   swap(&x, &y);
16   cout << "x = " << x << ", y = " << y << endl;
17   return 0;
18 }
```

- Output:

  ```
  x = 5, y = 10
  x = 10, y = 5
  ```

- bereits bekannt aus C:
  - übergebe Adressen `&x`, `&y` mit Call-by-Value
  - lokale Variablen `px`, `py` vom Typ `int*`
  - Zugriff auf Speicherbereich von `x` durch Dereferenzieren `*px`
  - analog für `*py`
- Zeile 6–8: Vertauschen der Inhalte von `*px` und `*py`

Funktionsargumente als Referenz

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 void swap(int& rx, int& ry) {
 6   int tmp = rx;
 7   rx = ry;
 8   ry = tmp;
 9 }
10
11 int main() {
12   int x = 5;
13   int y = 10;
14   cout << "x = " << x << ", y = " << y << endl;
15   swap(x, y);
16   cout << "x = " << x << ", y = " << y << endl;
17   return 0;
18 }
```

- Output:

  ```
  x = 5, y = 10
  x = 10, y = 5
  ```

- echtes Call-by-Reference in C++
  - Funktion kriegt als Input Referenzen
  - Syntax: `type fctName( ..., type& ref, ... )`
    - dieser Input wird als Referenz übergeben
- `rx` ist lokaler Name (Zeile 5–9) für den Speicherbereich von `x` (Zeile 12–17)
- analog für `ry` und `y`

Referenzen vs. Pointer

- Referenzen sind Aliasnamen für Variablen
  - müssen bei Deklaration initialisiert werden
  - kann Referenzen nicht nachträglich zuordnen!

- keine vollständige Alternative zu Pointern
  - keine Mehrfachzuweisung
  - kein dynamischer Speicher möglich
  - keine Felder von Referenzen möglich
  - Referenzen dürfen nicht `NULL` sein

- **Achtung:** Syntax verschleiert Programmablauf
  - bei Funktionsaufruf nicht klar, ob Call by Value oder Call by Reference
  - anfällig für Laufzeitfehler, wenn Funktion Daten ändert, aber Hauptprogramm das nicht weiß
  - passiert bei Pointer nicht

- **Wann Call by Reference sinnvoll?**
  - falls Input-Daten umfangreich
    - denn Call by Value kopiert Daten
  - dann Funktionsaufruf billiger

Refs als Funktions-Output 1/3

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 int& fct() {
 6   int x = 4711;
 7   return x;
 8 }
 9
10 int main() {
11   int var = fct();
12   cout << "var = " << var << endl;
13
14   return 0;
15 }
```

- Referenzen können Output von Funktionen sein
  - sinnvoll bei Objekten (später!)
- wie bei Pointern auf Lifetime achten!
  - Referenz wird zurückgegeben (Zeile 7)
  - aber Speicher wird freigegeben, da Blockende!
- Compiler erzeugt Warnung

  ```
  reference_output.cpp:7: warning: reference to
  stack memory associated with local variable
  'x' returned
  ```

Refs als Funktions-Output 2/3

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 class Demo {
 6 private:
 7   int val;
 8 public:
 9   Demo(int input) {
10     val = input;
11   }
12   int getContent() {
13     return val;
14   }
15 };
16
17 int main() {
18   Demo var(10);
19   int x = var.getContent();
20   x = 1;
21   cout << "x = " << x << ", ";
22   cout << "val = " << var.getContent() << endl;
23   return 0;
24 }
```

- Output:

  ```
  x = 1, val = 10
  ```

- Auf Folie nichts Neues!
  - nur Motivation der folgenden Folie

Refs als Funktions-Output 3/3

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 class Demo {
 6 private:
 7   int val;
 8 public:
 9   Demo(int input) {
10     val = input;
11   }
12   int& getContent() {
13     return val;
14   }
15 };
16
17 int main() {
18   Demo var(10);
19   int& x = var.getContent();
20   x = 1;
21   cout << "x = " << x << ", ";
22   cout << "val = " << var.getContent() << endl;
23   return 0;
24 }
```

- Output:

  ```
  x = 1, val = 1
  ```

- Achtung: `private` Member wurde geändert
  - Das will man eigentlich nicht!
  - Das kann Laufzeitfehler produzieren!
- Beachte: Code von `getContent` gleich
  - nur andere Signatur
  - Änderungen nur in Zeile 12, 19

Schlüsselwort `const`

- Konstanten definieren
- read-only Referenzen
- Überladen & `const` bei Variablen
- Überladen & `const` bei Referenzen
- Überladen & `const` bei Methoden

- `const`
- `const int*`, `int const*`, `int* const`
- `const int&`

elementare Konstanten

- möglich über `#define CONST wert`
  - einfache Textersetzung `CONST` durch `wert`
  - fehleranfällig & kryptische Fehlermeldung
    - falls `wert` Syntax-Fehler erzeugt
  - Konvention: Konstantennamen groß schreiben

- besser als konstante Variable
  - z.B. `const int var = wert;`
  - z.B. `int const var = wert;`
    - beide Varianten haben dieselbe Bedeutung!
  - wird als Variable angelegt, aber Compiler verhindert Schreiben
  - zwingend Initialisierung bei Deklaration

- **Achtung** bei Pointern
  - `const int* ptr` ist Pointer auf `const int`
  - `int const* ptr` ist Pointer auf `const int`
    - beide Varianten haben dieselbe Bedeutung!
  - `int* const ptr` ist konstanter Pointer auf `int`

Beispiel 1/2

```
1 int main() {
2   const double var = 5;
3   var = 7;
4   return 0;
5 }
```

- Syntax-Fehler beim Kompilieren:

  ```
  const.cpp:3: error: read-only variable is
  not assignable
  ```

```
1 int main() {
2   const double var = 5;
3   double tmp  = 0;
4   const double* ptr = &var;
5   ptr = &tmp;
6   *ptr = 7;
7   return 0;
8 }
```

- Syntax-Fehler beim Kompilieren:

  ```
  const_pointer.cpp:6: error: read-only
  variable is not assignable
  ```

Beispiel 2/2

```
1 int main() {
2   const double var = 5;
3   double tmp  = 0;
4   double* const ptr = &var;
5   ptr = &tmp;
6   *ptr = 7;
7   return 0;
8 }
```

- Syntax-Fehler beim Kompilieren:

  ```
  const_pointer2.cpp:4: error: cannot
  initialize a variable of type 'double *const'
  with an rvalue of type 'const double *'
  ```

  - Der Pointer `ptr` hat falschen Typ (Zeile 4)

```
1 int main() {
2   const double var = 5;
3   double tmp  = 0;
4   const double* const ptr = &var;
5   ptr = &tmp;
6   *ptr = 7;
7   return 0;
8 }
```

- zwei Syntax-Fehler beim Kompilieren:

  ```
  const_pointer3.cpp:5: error: read-only
  variable is not assignable
  const_pointer3.cpp:6: error: read-only
  variable is not assignable
  ```

  - Zuweisung auf Pointer `ptr` (Zeile 5)
  - Dereferenzieren und Schreiben (Zeile 6)

Read-Only Referenzen

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 int main() {
 6   double var = 5;
 7   double& ref = var;
 8   const double& cref = var;
 9   cout << "var = " << var << ", ";
10   cout << "ref = " << ref << ", ";
11   cout << "cref = " << cref << endl;
12   ref = 7;
13   cout << "var = " << var << ", ";
14   cout << "ref = " << ref << ", ";
15   cout << "cref = " << cref << endl;
16   // cref = 9;
17   return 0;
18 }
```

- `const type& cref`
  - deklariert konstante Referenz auf `type`
    - alternative Syntax: `type const& cref`
  - d.h. `cref` ist wie Variable vom Typ `const type`
  - Zugriff auf Referenz nur **lesend** möglich
- Output:

  ```
  var = 5, ref = 5, cref = 5
  var = 7, ref = 7, cref = 7
  ```

- Zeile `cref = 9;` würde Syntaxfehler liefern

  ```
  error: read-only variable is not assignable
  ```

Read-Only Refs als Output 1/2

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 class Demo {
 6 private:
 7   int val;
 8 public:
 9   Demo(int input) {
10     val = input;
11   }
12   int& getContent() {
13     return val;
14   }
15 };
16
17 int main() {
18   Demo var(10);
19   int& x = var.getContent();
20   x = 1;
21   cout << "x = " << x << ", ";
22   cout << "val = " << var.getContent() << endl;
23   return 0;
24 }
```

- Output:

  ```
  x = 1, val = 1
  ```

- Achtung: `private` Member wurde geändert
- selber Code wie oben (nur Wiederholung!)

Read-Only Refs als Output 2/2

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 class Demo {
 6 private:
 7   int val;
 8 public:
 9   Demo(int input) { val = input; }
10   const int& getContent() { return val; }
11 };
12
13 int main() {
14   Demo var(10);
15   const int& x = var.getContent();
16   // x = 1;
17   cout << "x = " << x << ", ";
18   cout << "val = " << var.getContent() << endl;
19   return 0;
20 }
```

- Output:

  ```
  x = 10, val = 10
  ```

- Zuweisung `x = 1;` würde Syntax-Fehler liefern

  ```
  error: read-only variable is not assignable
  ```

- Deklaration `int& x = var.getContent();` würde Syntax-Fehler liefern

  ```
  error: binding of reference to type 'int' to
  a value of type 'const int' drops qualifiers
  ```

- sinnvoll, falls Read-Only Rückgabe sehr groß ist
  - z.B. Vektor, langer String etc.

Type Casting

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 double square(double& x) {
 6   return x*x;
 7 }
 8
 9 int main() {
10   const double var = 5;
11   cout << "var = " << var << ", ";
12   cout << "var*var = " << square(var) << endl;
13   return 0;
14 }
```

- `const type` ist stärker als `type`
  - kein Type Casting von `const type` auf `type`
- Syntax-Fehler beim Kompilieren:

  ```
  const_typecasting.cpp:12 error: no matching
  function for call to 'square'
  const_typecasting.cpp:5: note: candidate
  function not viable: 1st argument
  ('const double') would lose const qualifier
  ```

- Type Casting von `type` auf `const type` ist aber OK!
- mögliche Lösung: Signatur ändern auf
  - `double square(const double& x)`

Read-Only Refs als Input 1/5

```
 1 #include "vector_first.hpp"
 2 #include <iostream>
 3 #include <cassert>
 4
 5 using std::cout;
 6
 7 double product(const Vector& x, const Vector& y){
 8   double sum = 0;
 9   assert( x.size() == y.size() );
10   for (int j=0; j<x.size(); ++j) {
11     sum = sum + x.get(j)*y.get(j);
12   }
13   return sum;
14 }
15
16 int main() {
17   Vector x(100,1);
18   Vector y(100,2);
19   cout << "norm(x) = " << x.norm() << "\n";
20   cout << "norm(y) = " << y.norm() << "\n";
21   cout << "x.y = " << product(x,y) << "\n";
22   return 0;
23 }
```

- Vorteil: schlanker Daten-Input ohne Kopieren!
  - und: Daten können nicht verändert werden!
- Problem: Syntax-Fehler beim Kompilieren, z.B.

  ```
  const_vector.cpp:9: error: member function
  'size' not viable: 'this' argument has type
  'const Vector', but function is not marked
  const
  ```

  - d.h. Problem mit Methode `size`

Read-Only Refs als Input 2/5

```
 1 #ifndef _VECTOR_NEW_
 2 #define _VECTOR_NEW_
 3
 4 #include <cmath>
 5 #include <cstdlib>
 6 #include <cassert>
 7
 8 // The class Vector stores vectors in Rd
 9
10 class Vector {
11 private:
12   // dimension of the vector
13   int dim;
14   // dynamic coefficient vector
15   double* coeff;
16
17 public:
18   // constructors and destructor
19   Vector();
20   Vector(int, double = 0);
21   ~Vector();
22
23   // return vector dimension
24   int size() const;
25
26   // read and write vector coefficients
27   void set(int k, double value);
28   double get(int k) const;
29
30   // compute Euclidean norm
31   double norm() const;
32 };
33
34 #endif
```

- Read-Only Methoden werden mit `const` markiert
  - `className::fct(... input ...) const { ... }`
  - geht nur bei Methoden, nicht bei allg. Fktn.
- neue Syntax: Zeile 24, 28, 31

Read-Only Refs als Input 3/5

```
 1 #include "vector_new.hpp"
 2 #include <iostream>
 3 using std::cout;
 4
 5 Vector::Vector() {
 6   dim = 0;
 7   coeff = (double*) 0;
 8   cout << "new empty vector" << "\n";
 9 }
10
11 Vector::Vector(int dim, double init) {
12   assert(dim > 0);
13   this->dim = dim;
14   coeff = (double*) malloc(dim*sizeof(double));
15   assert( coeff != (double*) 0);
16   for (int j=0; j<dim; ++j) {
17     coeff[j] = init;
18   }
19   cout << "new vector, length " << dim << "\n";
20 }
21
22 Vector::~Vector() {
23   if (dim > 0) {
24     free(coeff);
25   }
26   cout << "free vector, length " << dim << "\n";
27 }
```

- keine Änderungen!

Read-Only Refs als Input 4/5

```
29 int Vector::size() const {
30   return dim;
31 }
32
33 void Vector::set(int k, double value) {
34   assert(k>=0 && k<dim);
35   coeff[k] = value;
36 }
37
38 double Vector::get(int k) const {
39   assert(k>=0 && k<dim);
40   return coeff[k];
41 }
42
43 double Vector::norm() const {
44   double norm = 0;
45   for (int j=0; j<dim; ++j) {
46     norm = norm + coeff[j]*coeff[j];
47   }
48   return sqrt(norm);
49 }
```

- geändert: Zeile 29, 38, 43

Read-Only Refs als Input 5/5

```
 1 #include "vector_new.hpp"
 2 #include <iostream>
 3 #include <cassert>
 4
 5 using std::cout;
 6
 7 double product(const Vector& x, const Vector& y){
 8   double sum = 0;
 9   assert( x.size() == y.size() );
10   for (int j=0; j<x.size(); ++j) {
11     sum = sum + x.get(j)*y.get(j);
12   }
13   return sum;
14 }
15
16 int main() {
17   Vector x(100,1);
18   Vector y(100,2);
19   cout << "norm(x) = " << x.norm() << "\n";
20   cout << "norm(y) = " << y.norm() << "\n";
21   cout << "x.y = " << product(x,y) << "\n";
22   return 0;
23 }
```

- Vorteil: schlanker Daten-Input ohne Kopieren!
  - und: Daten können nicht verändert werden!
- Output:

```
new vector, length 100
new vector, length 100
norm(x) = 10
norm(y) = 20
x.y = 200
free vector, length 100
free vector, length 100
```

Zusammenfassung Syntax

- bei normalen Datentypen (nicht Pointer, Referenz)
  - `const int var`
  - `int const var`
    - dieselbe Bedeutung = Integer-Konstante
- bei Referenzen
  - `const int& ref` = Referenz auf `const int`
  - `int const& ref` = Referenz auf `const int`
- Achtung bei Pointern
  - `const int* ptr` = Pointer auf `const int`
  - `int const* ptr` = Pointer auf `const int`
  - `int* const ptr` = konstanter Pointer auf `int`
- bei Methoden, die nur Lese-Zugriff brauchen
  - `className::fct(... input ...) const`
  - kann Methode sonst nicht mit `const`-Refs nutzen
- sinnvoll, falls Rückgabe eine Referenz ist
  - `const int& fct(... input ...)`
  - lohnt sich nur bei großer Rückgabe, die nur gelesen wird
  - **Achtung:** Rückgabe muss existieren, sonst Laufzeitfehler!

Überladen und const 1/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 void f(int x) { cout << "int\n"; };
 5 void f(const int x) { cout << "const int\n"; };
 6
 7 int main() {
 8     int x = 0;
 9     const int c = 0;
10     f(x);
11     f(c);
12     return 0;
13 }
```

- const wird bei Input-Variablen nicht berücksichtigt
  - Syntax-Fehler beim Kompilieren:

    ```
    overload_const.cpp:5: error: redefinition
    of 'f'
    ```

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 void f(int& x) { cout << "int\n"; };
 5 void f(const int& x) { cout << "const int\n"; };
 6
 7 int main() {
 8     int x = 0;
 9     const int c = 0;
10     f(x);
11     f(c);
12     return 0;
13 }
```

- const wichtig bei Referenzen als Input
  - Kompilieren OK und Output:

    ```
    int
    const int
    ```

Überladen und const 2/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Demo {
 5 private:
 6     int content;
 7 public:
 8     Demo() { content = 0; }
 9     void f() { cout << "normales Objekt\n"; };
10     void f() const { cout << "const Objekt\n"; };
11 };
12
13 int main() {
14     Demo x;
15     const Demo y;
16     x.f();
17     y.f();
18     return 0;
19 }
```

- kann Methode durch const-Methode überladen
  - const-Meth. wird für const-Objekte verwendet
  - sonst wird „normale" Methode verwendet
- Output:

  ```
  normales Objekt
  const Objekt
  ```

Überladen von Operatoren

- Kopierkonstruktor
- Type Casting
- Zuweisungsoperator
- Unäre und binäre Operatoren

- `operator`

Klasse für Komplexe Zahlen

```
 1 #include <iostream>
 2 #include <cmath>
 3
 4 class Complex {
 5 private:
 6   double re;
 7   double im;
 8 public:
 9   Complex(double=0, double=0);
10   double real() const;
11   double imag() const;
12   double abs() const;
13   void print() const;
14 };
15
16 Complex::Complex(double re, double im) {
17   this->re = re;
18   this->im = im;
19 }
20
21 double Complex::real() const {
22   return re;
23 }
24
25 double Complex::imag() const {
26   return im;
27 }
28
29 double Complex::abs() const {
30     return sqrt(re*re + im*im);
31 }
32
33 void Complex::print() const {
34   std::cout << re << " + " << im << " * i";
35 }
```

- Default-Parameter in der ersten Deklaration
  - Zeile 9: Forward Declaration des Konstruktors
  - Zeile 16–19: Code des Konstruktors

Kopierkonstruktor

```
1 Complex::Complex(const Complex& rhs) {
2   re = rhs.re;
3   im = rhs.im;
4 }
```

- `className::className(const className& rhs)`
- Spezieller Konstruktor für den Aufruf
  - `Complex lhs = rhs;`
  - oder auch `Complex lhs(rhs);`
- erzeugt neues Objekt `lhs`, kopiert Daten von `rhs`
  - also Input als konstante Referenz (read-only)
- wird automatisch erstellt (Shallow Copy), falls nicht explizit programmiert
  - hier formal unnötig, da nur statische Daten
  - wichtig, falls Klasse dynamische Daten enthält

Zuweisungsoperator

```
1 Complex& Complex::operator=(const Complex& rhs) {
2   if (this != &rhs) {
3     re = rhs.re;
4     im = rhs.im;
5   }
6   return *this;
7 }
```

- `className& className::operator=(const className& rhs)`
- Falls `Complex lhs, rhs;` bereits deklariert
  - Zuweisung `lhs = rhs;`
  - keine Deklaration, also Referenz zurückgeben
  - Input als konstante Referenz (read-only)
  - Output als Referenz für Zuweisungsketten
    - z.B. `a = b = c = d;`
    - `=` weist von rechts nach links zu!
    - `a = ...` braucht Auswertung von `b = c = d;`
- Funktionalität:
  - Daten von `lhs` durch `rhs` überschreiben
  - ggf. dynamische Daten von `lhs` vorher freigeben
- `this` is Pointer auf das Objekt selbst
  - d.h. `*this` ist das Objekt selbst
- `if` verhindert Konflikt bei Selbstzuweisung `z = z;`
  - hier formal unnötig, da nur statische Daten
- wird automatisch erstellt (Shallow Copy), falls nicht explizit programmiert
  - hier formal unnötig, da nur statische Daten
  - wichtig, falls Klasse dynamische Daten enthält

Type Casting

```
1 Complex::Complex(double re = 0, double im = 0) {
2   this->re = re;
3   this->im = im;
4 }
```

- Konstruktor gibt Type Cast `double` auf `Complex`
  - d.h. $x \in \mathbb{R}$ entspricht $x + 0i \in \mathbb{C}$

```
1 Complex::operator double() const {
2   return re;
3 }
```

- Type Cast `Complex` auf `double`, z.B. durch Realteil
  - formal: `ClassName::operator type() const`
    - implizite Rückgabe
- Beachte ggf. bekannte Type Casts
  - implizit von `int` auf `double`
  - oder implizit von `double` auf `int`

Unäre Operatoren

- unäre Operatoren = Op. mit einem Argument

```
1 const Complex Complex::operator-() const {
2   return Complex(-re,-im);
3 }
```

- Vorzeichenwechsel - (Minus)
  - const Complex Complex::operator-() const
    - Output ist vom Typ const Complex
    - Methode agiert nur auf aktuellen Members
    - Methode ist read-only auf aktuellen Daten
  - wird Methode der Klasse
- Aufruf später durch -x

```
1 const Complex Complex::operator~() const {
2   return Complex(re,-im);
3 }
```

- Konjugation ~ (Tilde)
  - const Complex Complex::operator~() const
    - Output ist vom Typ const Complex
    - Methode agiert nur auf aktuellen Members
    - Methode ist read-only auf aktuellen Daten
  - wird Methode der Klasse
- Aufruf später durch ~x

complex_part.hpp

```
 1 #ifndef _COMPLEX_PART_
 2 #define _COMPLEX_PART_
 3
 4 #include <iostream>
 5 #include <cmath>
 6
 7 class Complex {
 8 private:
 9   double re;
10   double im;
11 public:
12   Complex(double=0, double=0);
13   Complex(const Complex& rhs);
14   ~Complex();
15   Complex& operator=(const Complex& rhs);
16
17   double real() const;
18   double imag() const;
19   double abs() const;
20   void print() const;
21
22   operator double() const;
23
24   const Complex operator~() const;
25   const Complex operator-() const;
26 };
27
28 #endif
```

- Zeile 12: Forward Declaration mit Default-Input
- Zeile 12 + 22: Type Casts `Complex` vs. `double`

complex_part.cpp 1/2

```
 1 #include "complex_part.hpp"
 2
 3 using std::cout;
 4
 5 Complex::Complex(double re, double im) {
 6   this->re = re;
 7   this->im = im;
 8   cout << "Konstruktor\n";
 9 }
10
11 Complex::Complex(const Complex& rhs) {
12   re = rhs.re;
13   im = rhs.im;
14   cout << "Kopierkonstruktor\n";
15 }
16
17 Complex::~Complex() {
18   cout << "Destruktor\n";
19 }
20
21 Complex& Complex::operator=(const Complex& rhs) {
22   if (this != &rhs) {
23     re = rhs.re;
24     im = rhs.im;
25     cout << "Zuweisung\n";
26   }
27   else {
28     cout << "Selbstzuweisung\n";
29   }
30   return *this;
31 }
```

complex_part.cpp 2/2

```
33 double Complex::real() const {
34   return re;
35 }
36
37 double Complex::imag() const {
38   return im;
39 }
40
41 double Complex::abs() const {
42     return sqrt(re*re + im*im);
43 }
44
45 void Complex::print() const {
46   cout << re << " + " << im << "*i";
47 }
48
49 Complex::operator double() const {
50   cout << "Complex -> double\n";
51   return re;
52 }
53
54 const Complex Complex::operator-() const {
55   return Complex(-re,-im);
56 }
57
58 const Complex Complex::operator~() const {
59   return Complex(re,-im);
60 }
```

Beispiel

```
 1 #include <iostream>
 2 #include "complex_part.hpp"
 3 using std::cout;
 4
 5 int main() {
 6   Complex w(1);
 7   Complex x;
 8   Complex y(1,1);
 9   Complex z = y;
10   x = x;
11   x = ~y;
12   w.print(); cout << "\n";
13   x.print(); cout << "\n";
14   y.print(); cout << "\n";
15   z.print(); cout << "\n";
16   return 0;
17 }
```

- Output:

```
Konstruktor
Konstruktor
Konstruktor
Kopierkonstruktor
Selbstzuweisung
Konstruktor
Zuweisung
Destruktor
1 + 0*i
1 + -1*i
1 + 1*i
1 + 1*i
Destruktor
Destruktor
Destruktor
Destruktor
```

Beispiel: Type Cast

```
 1 #include <iostream>
 2 #include "complex_part.hpp"
 3 using std::cout;
 4
 5 int main() {
 6   Complex z((int) 2.3, (int) 1);
 7   double x = z;
 8   z.print(); cout << "\n";
 9   cout << x << "\n";
10   return 0;
11 }
```

- Konstruktor fordert `double` als Input (Zeile 6)
  - erst expliziter Type Cast `2.3` auf `int`
  - dann impliziter Type Cast auf `double`
- Output:

```
Konstruktor
Complex -> double
2 + 1*i
2
Destruktor
```

Binäre Operatoren

```
 1 const Complex operator+(const Complex& x,const Complex& y){
 2   double xr = x.real();
 3   double xi = x.imag();
 4   double yr = y.real();
 5   double yi = y.imag();
 6   return Complex(xr + yr, xi + yi);
 7 }
 8 const Complex operator-(const Complex& x,const Complex& y){
 9   double xr = x.real();
10   double xi = x.imag();
11   double yr = y.real();
12   double yi = y.imag();
13   return Complex(xr - yr, xi - yi);
14 }
15 const Complex operator*(const Complex& x,const Complex& y){
16   double xr = x.real();
17   double xi = x.imag();
18   double yr = y.real();
19   double yi = y.imag();
20   return Complex(xr*yr - xi*yi, xr*yi + xi*yr);
21 }
22 const Complex operator/(const Complex& x,const double y){
23   assert(y != 0)
24   return Complex(x.real()/y, x.imag()/y);
25 }
26 const Complex operator/(const Complex& x,const Complex& y){
27   double norm = y.abs();
28   assert(norm > 0);
29   return x*~y / (norm*norm);
30 }
```

- binäre Operatoren = Op. mit zwei Argumenten
  - z.B. `+`, `-`, `*`, `/`
- außerhalb der Klassendefinition als Funktion
  - formal: `const type operator+(const type& rhs1, const type& rhs2)`
  - **Achtung:** kein `type::` da kein Teil der Klasse!
- Zeile 22 + 26: beachte $x/y = (x\overline{y})/(y\overline{y}) = x\overline{y}/|y|^2$

Operator <<

```
 1 std::ostream& operator<<(std::ostream& output,
 2                          const Complex& x) {
 3   if (x.imag() == 0) {
 4     return output << x.real();
 5   }
 6   else if (x.real() == 0) {
 7     return output << x.imag() << "i";
 8   }
 9   else {
10     return output << x.real() << " + " << x.imag() << "i";
11   }
12 }
```

- `cout`-Ausgabe erfolgt über Klasse `std::ostream`
- weitere Ausgabe wird einfach angehängt mit `<<`
  - kann insbesondere `for`-Schleife verwenden, um z.B. Vektoren / Matrizen mit `cout` auszugeben

complex.hpp

```
 1 #ifndef _COMPLEX_
 2 #define _COMPLEX_
 3
 4 #include <iostream>
 5 #include <cmath>
 6 #include <cassert>
 7
 8 class Complex {
 9 private:
10   double re;
11   double im;
12 public:
13   Complex(double=0, double=0);
14   Complex(const Complex&);
15   ~Complex();
16   Complex& operator=(const Complex&);
17
18   double real() const;
19   double imag() const;
20   double abs() const;
21
22   operator double() const;
23
24   const Complex operator~() const;
25   const Complex operator-() const;
26
27 };
28
29 std::ostream& operator<<(std::ostream& output,
30                                          const Complex& x);
31 const Complex operator+(const Complex&, const Complex&);
32 const Complex operator-(const Complex&, const Complex&);
33 const Complex operator*(const Complex&, const Complex&);
34 const Complex operator/(const Complex&, const double);
35 const Complex operator/(const Complex&, const Complex&);
36
37 #endif
```

- "vollständige Bibliothek" ohne unnötige cout im folgende cpp Source-Code

complex.cpp 1/3

```
 1 #include "complex.hpp"
 2 using std::ostream;
 3
 4 Complex::Complex(double re, double im) {
 5   this->re = re;
 6   this->im = im;
 7 }
 8
 9 Complex::Complex(const Complex& rhs) {
10   re = rhs.re;
11   im = rhs.im;
12 }
13
14 Complex::~Complex() {
15 }
16
17 Complex& Complex::operator=(const Complex& rhs) {
18   if (this != &rhs) {
19     re = rhs.re;
20     im = rhs.im;
21   }
22   return *this;
23 }
24
25 double Complex::real() const {
26   return re;
27 }
28
29 double Complex::imag() const {
30   return im;
31 }
32
33 double Complex::abs() const {
34   return sqrt(re*re + im*im);
35 }
36
37 Complex::operator double() const {
38     return re;
39 }
```

complex.cpp 2/3

```
41 const Complex Complex::operator-() const {
42   return Complex(-re,-im);
43 }
44
45 const Complex Complex::operator~() const {
46   return Complex(re,-im);
47 }
48
49 const Complex operator+(const Complex& x,const Complex& y){
50   double xr = x.real();
51   double xi = x.imag();
52   double yr = y.real();
53   double yi = y.imag();
54   return Complex(xr + yr, xi + yi);
55 }
56
57 const Complex operator-(const Complex& x,const Complex& y){
58   double xr = x.real();
59   double xi = x.imag();
60   double yr = y.real();
61   double yi = y.imag();
62   return Complex(xr - yr, xi - yi);
63 }
64
65 const Complex operator*(const Complex& x,const Complex& y){
66   double xr = x.real();
67   double xi = x.imag();
68   double yr = y.real();
69   double yi = y.imag();
70   return Complex(xr*yr - xi*yi, xr*yi + xi*yr);
71 }
```

complex.cpp 3/3

```
73 const Complex operator/(const Complex& x, const double y){
74   assert(y != 0);
75   return Complex(x.real()/y, x.imag()/y);
76 }
77
78 const Complex operator/(const Complex& x,const Complex& y){
79   double norm = y.abs();
80   assert(norm > 0);
81   return x*~y / (norm*norm);
82 }
83
84 std::ostream& operator<<(std::ostream& output,
85                          const Complex& x) {
86   if (x.imag() == 0) {
87     return output << x.real();
88   }
89   else if (x.real() == 0) {
90     return output << x.imag() << "i";
91   }
92   else {
93     return output << x.real() << " + " << x.imag() << "i";
94   }
95 }
```

complex_main.cpp

```
 1 #include "complex.hpp"
 2 #include <iostream>
 3 using std::cout;
 4
 5 int main() {
 6   Complex w;
 7   Complex x(1,0);
 8   Complex y(0,1);
 9   Complex z(3,4);
10
11   w = x + y;
12   cout << w << "\n";
13
14   w = x*y;
15   cout << w << "\n";
16
17   w = x/y;
18   cout << w << "\n";
19
20   w = z/(x + y);
21   cout << w << "\n";
22
23   w = w.abs();
24   cout << w << "\n";
25
26   return 0;
27 }
```

- Output:

```
1 + 1i
1i
-1i
3.5 + 0.5i
3.53553
```

Funktionsaufruf & Kopierkonstruktor 1/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Demo {
 5 private:
 6   int data;
 7 public:
 8   Demo(int data = 0) {
 9     cout << "Standardkonstruktor\n";
10     this->data = data;
11   }
12
13   Demo(const Demo& rhs) {
14     cout << "Kopierkonstruktor\n";
15     data = rhs.data;
16   }
17
18   Demo& operator=(const Demo& rhs) {
19     cout << "Zuweisungsoperator\n";
20     data = rhs.data;
21     return *this;
22   }
23
24   ~Demo() {
25     cout << "Destruktor\n";
26
27   }
28 };
```

- Bei Funktionsaufruf werden Daten mittels Kopierkonstruktor an Funktion übergeben

Funktionsaufruf & Kopierkonstruktor 2/2

```
30 void function(Demo input) {
31   cout << "Funktion mit Call by Value\n";
32 }
33
34 void function2(Demo& input) {
35   cout << "Funktion mit Referenz\n";
36 }
37
38 int main() {
39   Demo x;
40   Demo y = x;
41   cout << "*** Funktionsaufruf (Call by Value)\n";
42   function(y);
43   cout << "*** Funktionsaufruf (Call by Reference)\n";
44   function2(x);
45   cout << "*** Programmende\n";
46   return 0;
47 }
```

- Bei Funktionsaufruf werden Daten mittels Kopierkonstruktor an Funktion übergeben
- Output:

```
Standardkonstruktor
Kopierkonstruktor
*** Funktionsaufruf (Call by Value)
Kopierkonstruktor
Funktion mit Call by Value
Destruktor
*** Funktionsaufruf (Call by Reference)
Funktion mit Referenz
*** Programmende
Destruktor
Destruktor
```

Zusammenfassung Syntax

- Konstruktor (= Type Cast auf `Class`)

  ```
  Class::Class( ... input ... )
  ```

- Destruktor

  ```
  Class::~Class()
  ```

- Type Cast von `Class` auf `type`

  ```
  Class::operator type() const
  ```

  - explizit durch Voranstellen `(type)`
  - implizit bei Zuweisung auf Var. vom Typ `type`

- Kopierkonstruktor (Deklaration mit Initialisierung)

  ```
  Class::Class(const Class&)
  ```

  - expliziter Aufruf durch `Class var(rhs);`
    - oder `Class var = rhs;`
  - implizite bei Funktionsaufruf (Call by Value)

- Zuweisungsoperator

  ```
  Class& Class::operator=(const Class&)
  ```

- unäre Operatoren, z.B. Tilde `~` und Vorzeichen `-`

  ```
  const Class Class::operator-() const
  ```

- binäre Operatoren, z.B. `+`, `-`, `*`, `/`

  ```
  const Class operator+(const Class&, const Class&)
  ```

  - außerhalb der Klasse als Funktion

- Ausgabe mittels `cout`

  ```
  std::ostream& operator<<(std::ostream& output,
                           const Class& object)
  ```

Binäre Operatoren in der Klasse

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Complex {
 5 private:
 6   double re;
 7   double im;
 8 public:
 9   Complex(double re=0, double im=0) {
10     this->re = re;
11     this->im = im;
12   }
13   const Complex operator-() const {
14     return Complex(-re,-im);
15   }
16   const Complex operator-(const Complex& y) {
17     return Complex(re-y.re, im-y.im);
18   }
19  void print() const {
20     cout << re << " + " << im << "\n";
21   }
22 };
23
24 int main() {
25   Complex x(1,0);
26   Complex y(0,1);
27   Complex w = x-y;
28   (-y).print();
29   w.print();
30 }
```

- binäre Operatoren `+`, `-`, `*`, `/` als Methode möglich
  - Vorzeichen (unärer Operator): Zeile 13-15
  - Subtraktion (binärer Operator): Zeile 16-18
    - dann erstes Argument = aktuelles Objekt
- statt außerhalb der Klasse als Funktion

```
const Complex operator-(const Complex& x, const Complex& y)
```

Welche Operatoren überladen?

```
+    -    *    /     &      ^       %
|    ~    !    =     <      >       +=
-=   *=   /=   %=    ^=     &=      |=
<<   >>   >>=  <<=   ==     !=      <=
>=   &&   ||   ++    --     ->*     ,
->   []   ()   new   new[]  delete  delete[]
```

- als unäre Operatoren, vorangestellt `++var`
  `const Class Class::operator++()`
- als unäre Operatoren, nachgestellt `var++`
  `const Class Class::operator++(int)`
- als binäre Operatoren
  `const Class operator+(const Class&, const Class&)`
- kann Operatoren auch überladen
  - z.B. Division `Complex/double` vs. `Complex/Complex`
  - z.B. unär und binär (neg. Vorzeichen vs. Minus)
  - unterschiedliche Signatur beachten!
- Man kann keine neuen Operatoren definieren!
- Man kann `.`, `:`, `::`, `sizeof`, `.*` nicht überladen!
- Im Test sind Signaturen für Operator vorgegeben!
  - Ausnahme: Konstruktor, Destruktor!

- https://www.c-plusplus.net/forum/232010-full
- https://en.wikipedia.org/wiki/Operators_in_C_and_C++

Dynamische Speicherverwaltung

- dynamische Speicherverwaltung in C++
- Dreierregel

- `new`, `new ... []`
- `delete`, `delete[]`

new vs. malloc

- `malloc` reserviert nur Speicher
  - **Nachteil:** Konstr. werden nicht aufgerufen
    - d.h. Initialisierung händisch
- ein dynamisches Objekt

```
type* var = (type*) malloc(sizeof(type));
*var = ...;
```

- dynamischer Vektor von Objekten der Länge `N`

```
type* vec = (type*) malloc(N*sizeof(type));
vec[j] = ...;
```

- in C++ ist Type Cast bei `malloc` zwingend!

- `new` reserviert Speicher + ruft Konstruktoren auf
- ein dynamisches Objekt (mit Standardkonstruktor)

```
type* var = new type;
```

- ein dynamisches Objekt (mit Konstruktor)

```
type* var = new type(... input ... );
```

- dyn. Vektor der Länge `N` (mit Standardkonstruktor)

```
type* vec = new type[N];
```

  - Standardkonstruktor für jeden Koeffizienten

- **Konvention:** Immer `new` verwenden!
- Aber: Es gibt keine C++ Variante von `realloc`

delete vs. free

- `free` gibt Speicher von `malloc` frei

```
type* vec = (type*) malloc(N*sizeof(type));
free(vec);
```

  - unabhängig von Objekt / Vektor von Objekten
  - nur auf Output von `malloc` anwenden!

- `delete` ruft Destruktor auf und gibt Speicher von `new` frei

```
type* var = new type(... input ... );
delete var;
```

  - für ein dynamische erzeugtes Objekt
  - nur auf Output von `new` anwenden!

- `delete[]` ruft Destruktor für jeden Koeffizienten auf und gibt Speicher von `new ...[N]` frei

```
type* vec = new type[N];
delete[] vec;
```

  - für einen dynamischen Vektor von Objekten
  - nur auf Output von `new ...[N]` anwenden!

- **Konvention:** Falls Pointer auf keinen dynamischen Speicher zeigt, wird er händisch auf `NULL` gesetzt
  - d.h. nach `free`, `delete`, `delete[]` folgt
    - `vec = (type*) NULL;`
    - in C++ häufiger: `vec = (type*) 0;`

Dreierregel

- auch: Regel der großen Drei
- Wenn Destruktor oder Kopierkonstruktor oder Zuweisungsoperator implementiert ist, so müssen alle drei implementiert werden!
- notwendig, wenn Klasse dynamische Felder enthält
  - anderenfalls macht Compiler automatisch Shallow Copy (OK bei elementaren Typen!)
  - denn Shallow Copy führt sonst auf Laufzeitfehler bei dynamischen Feldern

Missachtung der Dreierregel 1/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Demo {
 5 private:
 6   int n;
 7   double* data;
 8 public:
 9   Demo(int n, double input);
10   ~Demo();
11   int getN() const;
12   const double* getData() const;
13   void set(double input);
14 };
15
16 Demo::Demo(int n, double input) {
17   cout << "constructor, length " << n << "\n";
18   this->n = n;
19   data = new double[n];
20   for (int j=0; j<n; ++j) {
21     data[j] = input;
22   }
23 }
24
25 Demo::~Demo() {
26   cout << "destructor, length " << n << "\n";
27   delete[] data;
28 }
29
30 int Demo::getN() const {
31   return n;
32 }
33
34 const double* Demo::getData() const {
35   return data;
36 }
```

- Destruktor ist vorhanden (dynamischer Speicher!)
- Kopierkonstruktor und Zuweisungsoperator fehlen

Missachtung der Dreierregel 2/2

```
38 void Demo::set(double input) {
39   for (int j=0; j<n; ++j) {
40     data[j] = input;
41   }
42 }
43
44 std::ostream& operator<<(std::ostream& output,
45                          const Demo& object) {
46   const double* data = object.getData();
47   for(int j=0; j<object.getN(); ++j) {
48     output << data[j] << " ";
49   }
50   return output;
51 }
52
53 void print(Demo z) {
54   cout << "print: " << z << "\n";
55 }
56
57 int main() {
58   Demo x(4,2);
59   Demo y = x;
60   cout << "x = " << x << ", y = " << y << "\n";
61   y.set(3);
62   cout << "x = " << x << ", y = " << y << "\n";
63   print(x);
64   x.set(5);
65   cout << "x = " << x << ", y = " << y << "\n";
66   return 0;
67 }
```

- Output:

```
x = 2 2 2 2 , y = 2 2 2 2
x = 3 3 3 3 , y = 3 3 3 3
print: 3 3 3 3
destructor, length 4
x = 5 5 5 5 , y = 5 5 5 5
destructor, length 4
Speicherzugriffsfehler
```

vector.hpp

```
 1 #ifndef _VECTOR_
 2 #define _VECTOR_
 3 #include <cmath>
 4 #include <cassert>
 5
 6 // The class Vector stores vectors in Rd
 7 class Vector {
 8 private:
 9   int dim;
10   double* coeff;
11
12 public:
13   // constructors, destructor, assignment
14   Vector();
15   Vector(int dim, double init=0);
16   Vector(const Vector&);
17   ~Vector();
18   Vector& operator=(const Vector&);
19   // return length of vector
20   int size() const;
21   // read and write entries
22   const double& operator[](int k) const;
23   double& operator[](int k);
24   // compute Euclidean norm
25   double norm() const;
26 };
27
28 // addition of vectors
29 const Vector operator+(const Vector&, const Vector&);
30 // scalar multiplication
31 const Vector operator*(const double, const Vector&);
32 const Vector operator*(const Vector&, const double);
33 // scalar product
34 const double operator*(const Vector&, const Vector&);
35
36 #endif
```

- Überladen von [ ]
  - falls konstantes Objekt, Methode aus Zeile 22
  - falls “normales Objekt”, Methode aus Zeile 23

vector.cpp 1/4

```
 1 #include "vector.hpp"
 2 #include <iostream>
 3 using std::cout;
 4
 5 Vector::Vector() {
 6   dim = 0;
 7   coeff = (double*) 0;
 8   // just for demonstration purposes
 9   cout << "constructor, empty\n";
10 }
11
12 Vector::Vector(int dim, double init) {
13   assert(dim >= 0);
14   this->dim = dim;
15   if (dim == 0) {
16     coeff = (double*) 0;
17   }
18   else {
19     coeff = new double[dim];
20     for (int j=0; j<dim; ++j) {
21       coeff[j] = init;
22     }
23   }
24   // just for demonstration purposes
25   cout << "constructor, length " << dim << "\n";
26 }
27
28 Vector::Vector(const Vector& rhs) {
29   dim = rhs.dim;
30   if (dim == 0) {
31     coeff = (double*) 0;
32   }
33   else {
34     coeff = new double[dim];
35     for (int j=0; j<dim; ++j) {
36       coeff[j] = rhs[j];
37     }
38   }
39   // just for demonstration purposes
40   cout << "copy constructor, length " << dim << "\n";
41 }
```

vector.cpp 2/4

```
43 Vector::~Vector() {
44   if (dim > 0) {
45     delete[] coeff;
46   }
47   // just for demonstration purposes
48   cout << "free vector, length " << dim << "\n";
49 }
50
51 Vector& Vector::operator=(const Vector& rhs) {
52   if (this != &rhs) {
53     if (dim != rhs.dim) {
54       if (dim > 0) {
55         delete[] coeff;
56       }
57       dim = rhs.dim;
58       if (dim > 0) {
59         coeff = new double[dim];
60       }
61       else {
62         coeff = (double*) 0;
63       }
64     }
65     for (int j=0; j<dim; ++j) {
66       coeff[j] = rhs[j];
67     }
68   }
69   // just for demonstration purposes
70   cout << "deep copy, length " << dim << "\n";
71   return *this;
72 }
73
74 int Vector::size() const {
75   return dim;
76 }
```

vector.cpp 3/4

```
 78 const double& Vector::operator[](int k) const {
 79   assert(k>=0 && k<dim);
 80   return coeff[k];
 81 }
 82
 83 double& Vector::operator[](int k) {
 84   assert(k>=0 && k<dim);
 85   return coeff[k];
 86 }
 87
 88 double Vector::norm() const {
 89   double sum = 0;
 90   for (int j=0; j<dim; ++j) {
 91     sum = sum + coeff[j]*coeff[j];
 92   }
 93   return sqrt(sum);
 94 }
 95
 96 const Vector operator+(const Vector& rhs1,
 97                        const Vector& rhs2) {
 98   assert(rhs1.size() == rhs2.size());
 99   Vector result(rhs1);
100   for (int j=0; j<result.size(); ++j) {
101     result[j] += rhs2[j];
102   }
103   return result;
104 }
```

- Zugriff auf Vektor-Koeff. über [ ] (Zeile 81 + 86)

vector.cpp 4/4

```
106 const Vector operator*(const double scalar,
107                          const Vector& input) {
108   Vector result(input);
109   for (int j=0; j<result.size(); ++j) {
110     result[j] *= scalar;
111   }
112   return result;
113 }
114
115 const Vector operator*(const Vector& input,
116                          const double scalar) {
117   return scalar*input;
118 }
119
120 const double operator*(const Vector& rhs1,
121                          const Vector& rhs2) {
122   double scalarproduct = 0;
123   assert(rhs1.size() == rhs2.size());
124   for (int j=0; j<rhs1.size(); ++j) {
125     scalarproduct += rhs1[j]*rhs2[j];
126   }
127   return scalarproduct;
128 }
```

- Zeile 118: Falls man `Vector * double` nicht implementiert, kriegt man kryptischen Laufzeitfehler:
  - impliziter Type Cast `double` auf `int`
  - Aufruf Konstruktor mit einem `int`-Argument
  - vermutlich `assert`-Abbruch in Zeile 126
- Operator `*` dreifach überladen:
  - `Vector * double` Skalarmultiplikation
  - `double * Vector` Skalarmultiplikation
  - `Vector * Vector` Skalarprodukt

Beispiel

```
 1 #include "vector.hpp"
 2 #include <iostream>
 3 using std::cout;
 4
 5 int main() {
 6   Vector vector1;
 7   Vector vector2(100,4);
 8   Vector vector3 = 4*vector2;
 9   cout << "*** Addition\n";
10   vector1 = vector2 + vector2;
11   cout << "Norm1 = " << vector1.norm() << "\n";
12   cout << "Norm2 = " << vector2.norm() << "\n";
13   cout << "Norm3 = " << vector3.norm() << "\n";
14   cout << "Skalarprodukt = " << vector2*vector3 << "\n";
15   cout << "Norm " << (4*vector3).norm() << "\n";
16   return 0;
17 }
```

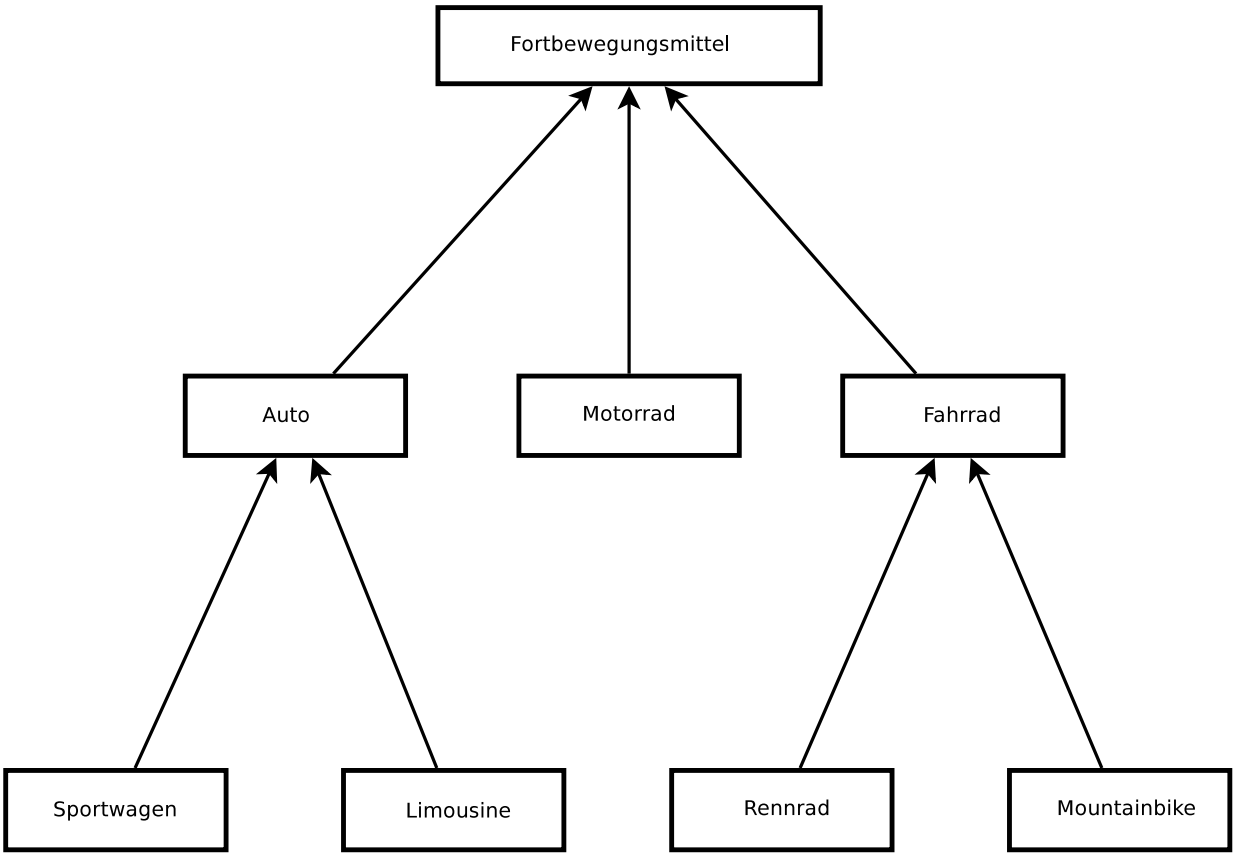
- Output:

```
constructor, empty
constructor, length 100
copy constructor, length 100
*** Addition
copy constructor, length 100
deep copy, length 100
free vector, length 100
Norm1 = 80
Norm2 = 40
Norm3 = 160
Skalarprodukt = 6400
Norm copy constructor, length 100
640
free vector, length 100
free vector, length 100
free vector, length 100
free vector, length 100
```

Vererbung

- Was ist Vererbung?
- Geerbte Felder und Methoden
- Methoden redefinieren
- Aufruf von Basismethoden

Was ist Vererbung?

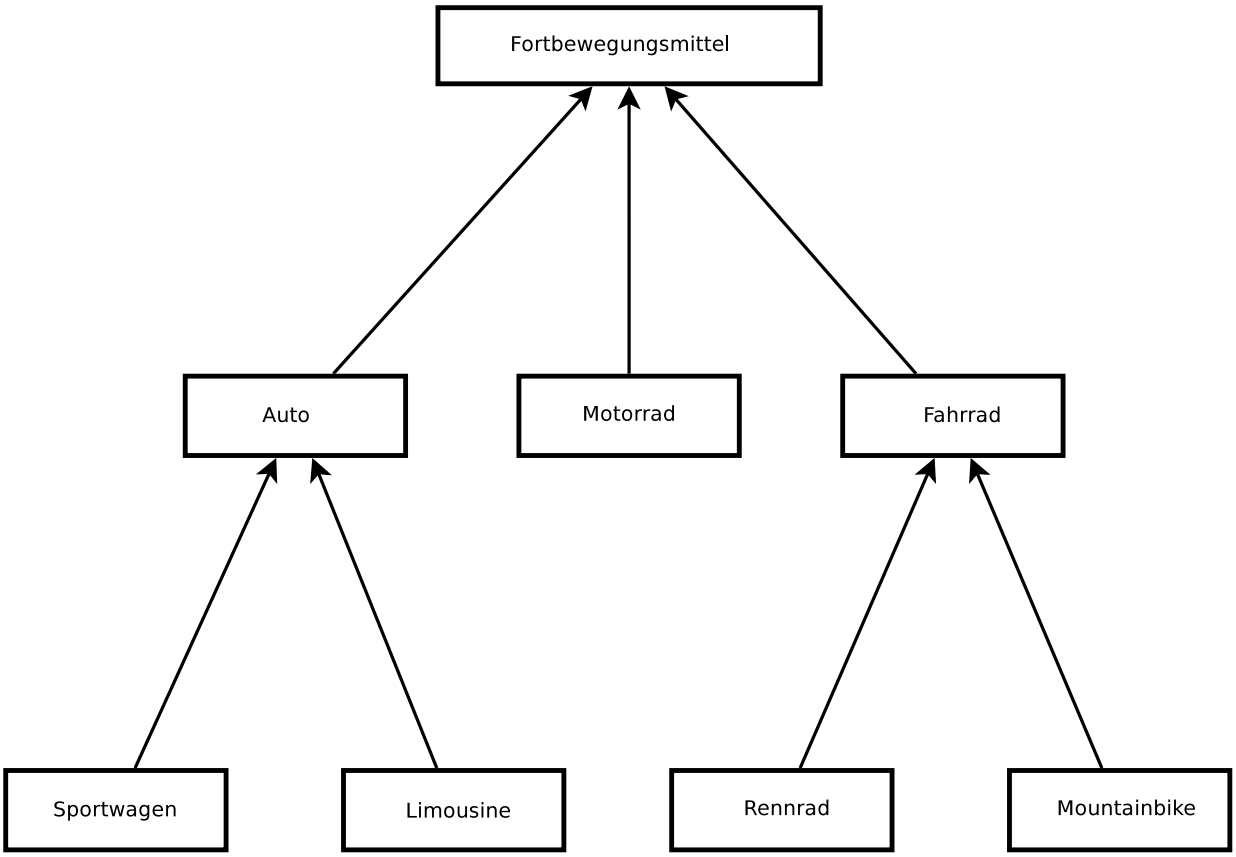


- im Alltag werden Objekte klassifiziert, z.B.
  - Jeder Sportwagen ist ein Auto
    - kann alles, was ein Auto kann, und noch mehr
  - Jedes Auto ist ein Fortbewegungsmittel
    - kann alles, was ein Fbm. kann, und noch mehr
- in C++ mittels Klassen abgebildet
  - Klasse (Fortbewegungsmittel) vererbt alle Members/Methoden an abgeleitete Klasse (Auto)
  - abgeleitete Klasse (Auto) kann zusätzliche Members/Methoden haben
- mathematisches Beispiel: $\mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$

public-Vererbung

- `class Abgeleitet : public Basisklasse { ... };`
  - Klasse `Abgeleitet` erbt alles von `Basisklasse`
    - alle Members + Methoden
  - Qualifier `public` gibt Art der Vererbung an
    - alle `private` Members von `Basisklasse` sind unsichtbare Members von `Abgeleitet`, d.h. nicht im Scope!
    - alle `public` Members von `Basisklasse` sind auch `public` Members von `Abgeleitet`
  - später noch Qualifier `private` und `protected`
  - kann weitere Members + Methoden zusätzlich für `Abgeleitet` im Block `{ ... }` definieren
    - wie bisher!
- Vorteil bei Vererbung:
  - Muss Funktionalität ggf. 1x implementieren!
  - Code wird kürzer (vermeidet Copy'n'Paste)
  - Fehlervermeidung

Formales Beispiel



- `class Fortbewegungsmittel { ... };`
- `class Auto : public Fortbewegungsmittel { ... };`
- `class Sportwagen : public Auto { ... };`
- `class Limousine : public Auto { ... };`
- `class Motorrad : public Fortbewegungsmittel { ... };`
- `class Fahrrad : public Fortbewegungsmittel { ... };`
- `class Rennrad : public Fahrrad { ... };`
- `class Mountainbike : public Fahrrad { ... };`

Ein erstes C++ Beispiel

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 private:
 6   double x;
 7 public:
 8   double getX() const { return x; }
 9   void setX(double input) { x = input; }
10 };
11
12 class Abgeleitet : public Basisklasse {
13 private:
14   double y;
15 public:
16   double getY() const { return y; }
17   void setY(double input) { y = input; }
18 };
19
20 int main() {
21   Basisklasse var1;
22   Abgeleitet var2;
23
24   var1.setX(5);
25   cout << "var1.x = " << var1.getX() << "\n";
26
27   var2.setX(1);
28   var2.setY(2);
29   cout << "var2.x = " << var2.getX() << "\n";
30   cout << "var2.y = " << var2.getY() << "\n";
31   return 0;
32 }
```

- Output:

```
var1.x = 5
var2.x = 1
var2.y = 2
```

private Members vererben 1/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 private:
 6   double x;
 7 public:
 8   Basisklasse() { x = 0; }
 9   Basisklasse(double inx) { x = inx; }
10   double getX() const { return x; }
11   void setX(double inx) { x = inx; }
12 };
13
14 class Abgeleitet : public Basisklasse {
15 private:
16   double y;
17 public:
18   Abgeleitet() { x = 0; y = 0; };
19   Abgeleitet(double inx, double iny) { x = inx; y = iny; };
20   double getY() const { return y; }
21   void setY(double iny) { y = iny; }
22 };
23
24 int main() {
25   Basisklasse var1(5);
26   Abgeleitet var2(1,2);
27
28   cout << "var1.x = " << var1.getX() << ", ";
29   cout << "var2.x = " << var2.getX() << ", ";
30   cout << "var2.y = " << var2.getY() << "\n";
31   return 0;
32 }
```

- derselbe Syntax-Fehler in Zeile 18 + 19:

  ```
  Ableiten2.cpp:18: error: 'x' is a private
  member of 'Basisklasse'
  ```

- Zugriff auf `private` Members nur in eigener Klasse, nicht im Scope bei Objekten abgeleiteter Klassen

private Members vererben 2/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 private:
 6   double x;
 7 public:
 8   Basisklasse() { x = 0; }
 9   Basisklasse(double inx) { x = inx; }
10   double getX() const { return x; }
11   void setX(double inx) { x = inx; }
12 };
13
14 class Abgeleitet : public Basisklasse {
15 private:
16   double y;
17 public:
18   Abgeleitet() { setX(0); y = 0; };
19   Abgeleitet(double inx, double iny) {setX(inx); y = iny;};
20   double getY() const { return y; }
21   void setY(double iny) { y = iny; }
22 };
23
24 int main() {
25   Basisklasse var1(5);
26   Abgeleitet var2(1,2);
27   cout << "var1.x = " << var1.getX() << ", ";
28   cout << "var2.x = " << var2.getX() << ", ";
29   cout << "var2.y = " << var2.getY() << "\n";
30   return 0;
31 }
```

- **Output:** `var1.x = 5, var2.x = 1, var2.y = 2`
- Zeile 18 + 19: Aufruf von `public`-Methoden aus `Basisklasse` erlaubt Zugriff auf `private`-Members von `Basisklasse` auch für Objekte der Klasse `Abgeleitet`
- `x` ist in `Abgeleitet` nicht im Scope, aber existiert!

Konstruktor & Destruktor 1/4

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 private:
 6   double x;
 7 public:
 8   Basisklasse() {
 9     cout << "Basisklasse()\n";
10     x = 0;
11   }
12   Basisklasse(double inx) {
13     cout << "Basisklasse(" << inx << ")\n";
14     x = inx;
15   }
16   ~Basisklasse() {
17     cout << "~Basisklasse()\n";
18   }
19   double getX() const { return x; }
20   void setX(double inx) { x = inx; }
21 };
22
23 class Abgeleitet : public Basisklasse {
24 private:
25   double y;
26 public:
27   Abgeleitet() {
28     cout << "Abgeleitet()\n";
29     setX(0);
30     y = 0;
31   };
32   Abgeleitet(double inx, double iny) {
33     cout << "Abgeleitet(" << inx << "," << iny << ")\n";
34     setX(inx);
35     y = iny;
36   };
37   ~Abgeleitet() {
38     cout << "~Abgeleitet()\n";
39   }
40   double getY() const { return y; }
41   void setY(double iny) { y = iny; }
42 };
```

Konstruktor & Destruktor 2/4

```
44 int main() {
45   Basisklasse var1(5);
46   Abgeleitet var2(1,2);
47   cout << "var1.x = " << var1.getX() << ", ";
48   cout << "var2.x = " << var2.getX() << ", ";
49   cout << "var2.y = " << var2.getY() << "\n";
50   return 0;
51 }
```

- Anlegen eines Objekts vom Typ `Abgeleitet` ruft Konstruktoren von `Basisklasse` und `Abgeleitet` auf
  - automatisch wird Standard-Konstr. aufgerufen!
- Freigabe eines Objekts vom Typ `Abgeleitet` ruft Destruktoren von `Abgeleitet` und `Basisklasse`
- Output:

```
Basisklasse(5)
Basisklasse()
Abgeleitet(1,2)
var1.x = 5, var2.x = 1, var2.y = 2
~Abgeleitet()
~Basisklasse()
~Basisklasse()
```

Konstruktor & Destruktor 3/4

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 private:
 6   double x;
 7 public:
 8   Basisklasse() {
 9     cout << "Basisklasse()\n";
10     x = 0;
11   }
12   Basisklasse(double inx) {
13     cout << "Basisklasse(" << inx << ")\n";
14     x = inx;
15   }
16   ~Basisklasse() {
17     cout << "~Basisklasse()\n";
18   }
19   double getX() const { return x; }
20   void setX(double inx) { x = inx; }
21 };
22
23 class Abgeleitet : public Basisklasse {
24 private:
25   double y;
26 public:
27   Abgeleitet() {
28     cout << "Abgeleitet()\n";
29     setX(0);
30     y = 0;
31   };
32   Abgeleitet(double inx, double iny) : Basisklasse(inx) {
33     cout << "Abgeleitet(" << inx << "," << iny << ")\n";
34     y = iny;
35   };
36   ~Abgeleitet() {
37     cout << "~Abgeleitet()\n";
38   }
39   double getY() const { return y; }
40   void setY(double iny) { y = iny; }
41 };
```

Konstruktor & Destruktor 4/4

```
43 int main() {
44   Basisklasse var1(5);
45   Abgeleitet var2(1,2);
46   cout << "var1.x = " << var1.getX() << ", ";
47   cout << "var2.x = " << var2.getX() << ", ";
48   cout << "var2.y = " << var2.getY() << "\n";
49   return 0;
50 }
```

- kann bewusst Konstruktor von `Basisklasse` wählen wenn Konstruktor von `Abgeleitet` aufgerufen wird
  - `Abgeleitet(...) : Basisklasse(...) {...};`
  - ruft Konstruktor von `Basisklasse`, welcher der Signatur entspricht ($\rightarrow$ Überladen)
- Output:

```
Basisklasse(5)
Basisklasse(1)
Abgeleitet(1,2)
var1.x = 5, var2.x = 1, var2.y = 2
~Abgeleitet()
~Basisklasse()
~Basisklasse()
```

Ein weiteres Beispiel 1/3

```
 1 #ifndef _FORTBEWEGUNGSMITTEL_
 2 #define _FORTBEWEGUNGSMITTEL_
 3
 4 #include <iostream>
 5 #include <string>
 6
 7 class Fortbewegungsmittel {
 8 private:
 9   double speed;
10 public:
11   Fortbewegungsmittel(double = 0);
12   double getSpeed() const;
13   void setSpeed(double);
14   void bewegen() const;
15 };
16
17 class Auto : public Fortbewegungsmittel {
18 private:
19   std::string farbe;
   // zusaetzliche Eigenschaft
20 public:
21   Auto();
22   Auto(double, std::string);
23   std::string getFarbe() const;
24   void setFarbe(std::string);
25   void schalten() const;    // zusaetzliche Faehigkeit
26 };
27
28 class Sportwagen : public Auto {
29 public:
30   Sportwagen();
31   Sportwagen(double, std::string);
32   void kickstart() const;   // zusaetzliche Eigenschaft
33 };
34 #endif
```

Ein weiteres Beispiel 2/3

```
 1 #include "fortbewegungsmittel.hpp"
 2 using std::string;
 3 using std::cout;
 4
 5 Fortbewegungsmittel::Fortbewegungsmittel(double s) {
 6   cout << "Fortbewegungsmittel(" << s << ")\n";
 7   speed = s;
 8 }
 9 double Fortbewegungsmittel::getSpeed() const {
10   return speed;
11 }
12 void Fortbewegungsmittel::setSpeed(double s) {
13   speed = s;
14 }
15 void Fortbewegungsmittel::bewegen() const {
16   cout << "Ich bewege mich mit " << speed << " km/h\n";
17 }
18
19 Auto::Auto() { cout << "Auto()\n"; };
20 Auto::Auto(double s, string f) : Fortbewegungsmittel(s) {
21   cout << "Auto(" << s << "," << f << ")\n";
22   farbe = f;
23 }
24 string Auto::getFarbe() const {
25   return farbe;
26 }
27 void Auto::setFarbe(string f) {
28   farbe = f;
29 }
30 void Auto::schalten() const {
31   cout << "Geschaltet\n";
32 }
33
34 Sportwagen::Sportwagen() { cout << "Sportwagen()\n"; };
35 Sportwagen::Sportwagen(double s, string f) : Auto(s,f) {
36   cout << "Sportwagen(" << s << "," << f << ")\n";
37 }
38 void Sportwagen::kickstart() const {
39   cout << "Roar\n";
40 }
```

Ein weiteres Beispiel 3/3

```
 1 #include "fortbewegungsmittel.hpp"
 2 #include <iostream>
 3
 4 int main() {
 5   Fortbewegungsmittel fahrrad(10);
 6   Auto cabrio(100,"rot");
 7   Sportwagen porsche(230,"schwarz");
 8
 9   fahrrad.bewegen();
10   cabrio.bewegen();
11   porsche.bewegen();
12
13   cabrio.schalten();
14   porsche.kickstart();
15
16   return 0;
17 }
```

- Output:

```
Fortbewegungsmittel(10)
Fortbewegungsmittel(100)
Auto(100,rot)
Fortbewegungsmittel(230)
Auto(230,schwarz)
Sportwagen(230,schwarz)
Ich bewege mich mit 10 km/h
Ich bewege mich mit 100 km/h
Ich bewege mich mit 230 km/h
Geschaltet
Roar
```

private, protected, public 1/2

- `private`, `protected`, `public` sind Qualifier für Members in Klassen
  - kontrollieren, wie auf Members der Klasse zugegriffen werden darf
- `private` (Standard)
  - Zugriff nur von Methoden der gleichen Klasse
- `protected`
  - Zugriff nur von Methoden der gleichen Klasse
  - Unterschied zu `private` nur bei Vererbung
- `public`
  - erlaubt Zugriff von überall
- **Konvention.** Datenfelder sind immer `private`
- `private`, `protected`, `public` sind auch Qualifier für Vererbung, z.B.
  - `class Abgeleitet : public Basisklasse {...};`

Basisklasse	**abgeleitete Klasse**		
	`public`	`protected`	`private`
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`
`private`	`hidden`	`hidden`	`hidden`

- Sichtbarkeit ändert sich durch Art der Vererbung
  - Zugriff kann nur verschärft werden
  - andere außer `public` machen selten Sinn

private, protected, public 2/2

```
 1 class Basisklasse {
 2 private:
 3   int a;
 4 protected:
 5   int b;
 6 public:
 7   int c;
 8 };
 9
10 class Abgeleitet : public Basisklasse {
11 public:
12   void methode() {
13     a = 10; // Nicht OK, da hidden
14     b = 10; // OK, da protected
15     c = 10; // OK, da public
16   }
17 };
18
19 int main() {
20   Basisklasse bas;
21   bas.a = 10; // Nicht OK, da private
22   bas.b = 10; // Nicht OK, da protected
23   bas.c = 10; // OK, da public
24
25   Abgeleitet abg;
26   abg.a = 10; // Nicht OK, da hidden
27   abg.b = 10; // Nicht OK, da protected
28   abg.c = 10; // OK, da public
29
30   return 0;
31 }
```

- Compiler liefert Syntax-Fehler in Zeile 13, 21, 22, 26, 27

```
protected.cpp:13: error: 'a' is a private
member of 'Basisklasse'
protected.cpp:3: note: declared private here
```

Methoden redefinieren 1/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 public:
 6   void print() { cout << "kein Input\n"; }
 7   void print(int x) { cout << "Input = " << x << "\n"; }
 8 };
 9
10 class Abgeleitet : public Basisklasse {
11 public:
12   void print() { cout << "Abgeleitet: kein Input\n"; }
13 };
14
15 int main() {
16   Basisklasse var1;
17   Abgeleitet var2;
18
19   var1.print();
20   var1.print(1);
21   var2.print();
22   var2.print(2);
23   return 0;
24 }
```

- wird in Basisklasse und abgeleiteter Klasse eine Methode gleichen Namens definiert, so steht für Objekte der abgeleiteten Klasse nur diese Methode zur Verfügung, alle Überladungen in der Basisklasse werden überdeckt, sog. Redefinieren
  - Unterscheide Überladen (Zeile 6 + 7)
  - und Redefinieren (Zeile 12)
- Kompilieren liefert Fehlermeldung:

```
redefinieren1.cpp:22: error: too many
arguments to function call, expected 0,
have 1; did you mean 'Basisklasse::print'?
```

Methoden redefinieren 2/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 public:
 6   void print() { cout << "kein Input\n"; }
 7   void print(int x) { cout << "Input = " << x << "\n"; }
 8 };
 9
10 class Abgeleitet : public Basisklasse {
11 public:
12   void print() { cout << "Abgeleitet: kein Input\n"; }
13 };
14
15 int main() {
16   Basisklasse var1;
17   Abgeleitet var2;
18
19   var1.print();
20   var1.print(1);
21   var2.print();
22   var2.Basisklasse::print(2); // nur diese Zeile ist anders
23   return 0;
24 }
```

- `Basisklasse` hat überladene Methode `print`
  - 2 Methoden (Zeile 6 + 7)
- `Abgeleitet` hat nur eine Methode `print` (Zeile 12)
  - `print` aus `Basisklasse` überdeckt (Redefinition)
- Zugriff auf `print` aus Basisklasse über vollständigen Namen möglich (inkl. Klasse als Namensbereich)
- Output:

```
kein Input
Input = 1
Abgeleitet: kein Input
Input = 2
```

Matrizen

- Klasse für Matrizen
- Vektoren als abgeleitete Klasse

Natürliche Matrix-Hierarchie

- für allgemeine Matrix $A \in \mathbb{R}^{m \times n}$
  - Vektoren $x \in \mathbb{R}^m \simeq \mathbb{R}^{m \times 1}$
  - quadratische Matrix $A \in \mathbb{R}^{n \times n}$
    - reguläre Matrix: $\det(A) \neq 0$
    - symmetrische Matrix: $A = A^T$
    - untere Dreiecksmatrix, $A_{jk} = 0$ für $k > j$
    - obere Dreiecksmatrix, $A_{jk} = 0$ für $k < j$
- kann für $A \in \mathbb{R}^{m \times n}$ z.B.
  - Matrix-Matrix-Summe
  - Matrix-Matrix-Produkt
  - Norm berechnen
- kann zusätzlich für quadratische Matrix, z.B.
  - Determinante berechnen
- kann zusätzlich für reguläre Matrix, z.B.
  - Gleichungssystem eindeutig lösen

Koeffizientenzugriff

```
 1 double& Matrix::operator()(int j, int k) {
 2   assert(j>=0 && j<m);
 3   assert(k>=0 && k<n);
 4   return coeff[j+k*m];
 5 }
 6
 7 double& Matrix::operator[](int ell) {
 8     assert(ell>=0 && ell<m*n);
 9     return coeff[ell];
10 }
```

- speichere Matrix $A \in \mathbb{R}^{m \times n}$ spaltenweise als $a \in \mathbb{R}^{mn}$
  - $A_{jk} = a_\ell$ mit $\ell = j + km$ für $j, k = 0, 1, \ldots$
- Operator `[ ]` erlaubt nur ein Argument in C++
  - Syntax `A[j,k]` nicht erlaubt
  - Syntax `A[j][k]` nur möglich mit `double** coeff`
- Nutze Operator `( )`, d.h. Zugriff mittels `A(j,k)`
  - `A(j,k)` liefert A_{jk}
- Nutze Operator `[ ]` für Zugriff auf Speichervektor
  - `A[ell]` liefert a_ℓ

Summe

```
 1 const Matrix operator+(const Matrix& A, const Matrix& B) {
 2   int m = A.size1();
 3   int n = A.size2();
 4   assert(m == B.size1() );
 5   assert(n == B.size2() );
 6   Matrix sum(m,n);
 7   for (int j=0; j<m; ++j) {
 8     for (int k=0; k<n; ++k) {
 9       sum(j,k) = A(j,k) + B(j,k);
10     }
11   }
12   return sum;
13 }
```

- $A+B$ nur definiert für Matrizen gleicher Dimension
  - $(A+B)_{jk} = A_{jk} + B_{jk}$
- könnte auch Speichervektoren addieren, aber...
  - führt auf Fehler, falls zwei Matrizen unterschiedlich gespeichert sind!
    - z.B. quadratische Matrix + untere Δ-Matrix

Produkt

```
 1 const Matrix operator*(const Matrix& A, const Matrix& B) {
 2   int m = A.size1();
 3   int n = A.size2();
 4   int p = B.size2();
 5   double sum = 0;
 6   assert(n == B.size1() );
 7   Matrix product(m,p);
 8   for (int i=0; i<m; ++i) {
 9     for (int k=0; k<p; ++k) {
10       sum = 0;
11       for (int j=0; j<n; ++j) {
12         sum = sum + A(i,j)*B(j,k);
13       }
14       product(i,k) = sum;
15     }
16   }
17   return product;
18 }
```

- $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p} \quad \Rightarrow \quad AB \in \mathbb{R}^{m \times p}$
  - erfordert passende Dimension!
  - $(AB)_{ik} = \sum_{j=0}^{n-1} A_{ij} B_{jk}$

matrix.hpp

```
 1 #ifndef _MATRIX_
 2 #define _MATRIX_
 3 #include <cmath>
 4 #include <cassert>
 5 #include <iostream>
 6
 7 class Matrix {
 8 private:
 9   int m;
10   int n;
11   double* coeff;
12
13 public:
14   // constructors, destructor, assignment
15   Matrix();
16   Matrix(int m, int n, double init=0);
17   Matrix(const Matrix&);
18   ~Matrix();
19   Matrix& operator=(const Matrix&);
20
21   // return size of matrix
22   int size1() const;
23   int size2() const;
24
25   // read and write entries with matrix access A(j,k)
26   const double& operator()(int j, int k) const;
27   double& operator()(int j, int k);
28
29   // read and write storage vector A[ell]
30   const double& operator[](int ell) const;
31   double& operator[](int ell);
32
33   // compute norm
34   double norm() const;
35 };
36
37 // matrix-matrix sum and product
38 const Matrix operator+(const Matrix&, const Matrix&);
39 const Matrix operator*(const Matrix&, const Matrix&);
40 // print matrix via output stream
41 std::ostream& operator<<(std::ostream& output,
42                                        const Matrix&);
43 #endif
```

matrix.cpp 1/4

```
 1 #include "matrix.hpp"
 2 using std::cout;
 3 using std::ostream;
 4
 5 Matrix::Matrix() {
 6   m = 0;
 7   n = 0;
 8   coeff = (double*) 0;
 9   cout << "constructor, empty\n";
10 }
11
12 Matrix::Matrix(int m, int n, double init) {
13   assert(m > 0);
14   assert(n > 0);
15   this->m = m;
16   this->n = n;
17   coeff = new double[m*n];
18   for (int ell=0; ell<m*n; ++ell) {
19     coeff[ell] = init;
20   }
21   cout << "constructor, " << m << " x " << n << "\n";
22 }
23
24 Matrix::Matrix(const Matrix& rhs) {
25   m = rhs.m;
26   n = rhs.n;
27   if (m > 0 && n > 0) {
28     coeff = new double[m*n];
29   }
30   else {
31     coeff = (double*) 0;
32   }
33   for (int ell=0; ell<m*n; ++ell) {
34     coeff[ell] = rhs[ell];
35   }
36   cout << "copy constructor, " << m << " x " << n << "\n";
37 }
```

matrix.cpp 2/4

```
39 Matrix::~Matrix() {
40   if (m > 0 && n > 0) {
41     delete[] coeff;
42   }
43   cout << "destructor, " << m << " x " << n << "\n";
44 }
45
46 Matrix& Matrix::operator=(const Matrix& rhs) {
47   if ( this != &rhs) {
48     if ( (m != rhs.m) || (n != rhs.n) ) {
49       if (m > 0 && n > 0) {
50         delete[] coeff;
51       }
52       m = rhs.m;
53       n = rhs.n;
54       if (m > 0 && n > 0) {
55         coeff = new double[m*n];
56       }
57       else {
58         coeff = (double*) 0;
59       }
60     }
61     for (int ell=0; ell<m*n; ++ell) {
62       coeff[ell] = rhs[ell];
63     }
64   }
65   cout << "deep copy, " << m << " x " << n << "\n";
66   return *this;
67 }
68
69 int Matrix::size1() const {
70   return m;
71 }
72
73 int Matrix::size2() const {
74   return n;
75 }
```

matrix.cpp 3/4

```
 77 const double& Matrix::operator()(int j, int k) const {
 78   assert(j>=0 && j<m);
 79   assert(k>=0 && k<n);
 80   return coeff[j+k*m];
 81 }
 82
 83 double& Matrix::operator()(int j, int k) {
 84   assert(j>=0 && j<m);
 85   assert(k>=0 && k<n);
 86   return coeff[j+k*m];
 87 }
 88
 89 const double& Matrix::operator[](int ell) const {
 90   assert( ell>=0 && ell<m*n );
 91   return coeff[ell];
 92 }
 93
 94 double& Matrix::operator[](int ell) {
 95   assert( ell>=0 && ell<m*n);
 96   return coeff[ell];
 97 }
 98
 99 double Matrix::norm() const {
100   double norm = 0;
101   for (int j=0; j<m; ++j) {
102     for (int k=0; k<n; ++k) {
103       norm = norm + (*this)(j,k) * (*this)(j,k);
104     }
105   }
106   return sqrt(norm);
107 }
```

```
109 const Matrix operator+(const Matrix& A, const Matrix& B) {
110   int m = A.size1();
111   int n = A.size2();
112   assert(m == B.size1() );
113   assert(n == B.size2() );
114   Matrix sum(m,n);
115   for (int j=0; j<m; ++j) {
116     for (int k=0; k<n; ++k) {
117       sum(j,k) = A(j,k) + B(j,k);
118     }
119   }
120   return sum;
121 }
122
123 const Matrix operator*(const Matrix& A, const Matrix& B) {
124   int m = A.size1();
125   int n = A.size2();
126   int p = B.size2();
127   double sum = 0;
128   assert(n == B.size1() );
129   Matrix product(m,p);
130   for (int i=0; i<m; ++i) {
131     for (int k=0; k<p; ++k) {
132       sum = 0;
133       for (int j=0; j<n; ++j) {
134         sum = sum + A(i,j)*B(j,k);
135       }
136       product(i,k) = sum;
137     }
138   }
139   return product;
140 }
141
142 ostream& operator<<(ostream& output, const Matrix& A) {
143   for (int j=0; j<A.size1(); j++) {
144     for (int k=0; k<A.size2(); k++) {
145       output << " " << A(j,k);
146     }
147     output << "\n";
148   }
149   return output;
150 }
```

Testbeispiel

```
 1 #include "matrix.hpp"
 2 #include <iostream>
 3 using std::cout;
 4
 5 int main() {
 6   int m = 2;
 7   int n = 3;
 8   Matrix A(m,n);
 9   for (int j=0; j<m; ++j) {
10     for (int k=0; k<n; ++k) {
11       A(j,k) = j + k*m;
12     }
13   }
14   cout << A;
15   Matrix C;
16   C = A + A;
17   cout << C;
18   return 0;
19 }
```

- Output:

```
constructor, 2 x 3
 0 2 4
 1 3 5
constructor, empty
constructor, 2 x 3
deep copy, 2 x 3
destructor, 2 x 3
 0 4 8
 2 6 10
destructor, 2 x 3
destructor, 2 x 3
```

matrix_vector.hpp

```
 1 #ifndef _VECTOR_
 2 #define _VECTOR_
 3
 4 #include "matrix.hpp"
 5
 6 class Vector : public Matrix {
 7 public:
 8   // constructor and type cast Matrix to Vector
 9   Vector();
10   Vector(int m, double init=0);
11   Vector(const Matrix&);
12
13   // return size of vector
14   int size() const;
15
16   // read and write coefficients with access x(j)
17   const double& operator()(int j) const;
18   double& operator()(int j);
19 };
20 #endif
```

- Identifiziere $x \in \mathbb{R}^n \simeq \mathbb{R}^{n \times 1}$
  - d.h. Klasse `Vector` wird von `Matrix` abgeleitet
- Konstr. `Vector x(n);` und `Vector x(n,init);`
- Type Cast von `Matrix` auf `Vector` schreiben!
  - Type Cast von `Vector` auf `Matrix` automatisch, da `Vector` von `Matrix` abgeleitet
- Zugriff auf Koeffizienten mit `x(j)` oder `x[j]`
- ACHTUNG mit Destr., Kopierkonstr. Zuweisung
  - wenn fehlt, aus Basisklasse genommen
  - hier kein Problem!

matrix_vector.cpp

```
 1 #include "matrix-vector.hpp"
 2 using std::cout;
 3
 4 Vector::Vector() {
 5   cout << "vector constructor, empty\n";
 6 }
 7
 8 Vector::Vector(int m, double init) : Matrix(m,1,init) {
 9   cout << "vector constructor, size " << m << "\n";
10 }
11
12 Vector::Vector(const Matrix& rhs) : Matrix(rhs.size1(),1) {
13   assert(rhs.size2() == 1);
14   for (int j=0; j<rhs.size1(); ++j) {
15     (*this)[j] = rhs(j,0);
16   }
17   cout << "type cast Matrix -> Vector\n";
18 }
19
20 int Vector::size() const {
21   return size1();
22 }
23
24 const double& Vector::operator()(int j) const {
25   assert( j>=0 && j<size() );
26   return (*this)[j];
27 }
28
29 double& Vector::operator()(int j) {
30   assert( j>=0 && j<size() );
31   return (*this)[j];
32 }
```

- Type Cast stellt sicher, dass Input in $\mathbb{R}^{n\times 1}$

Matrix-Vektor-Produkt

- $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p} \quad \Rightarrow \quad AB \in \mathbb{R}^{m \times p}$,
  - $(AB)_{ik} = \sum_{j=0}^{n-1} A_{ij} B_{jk}$
- $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^{n} \quad \Rightarrow \quad Ax \in \mathbb{R}^{m}$,
  - $(Ax)_{i} = \sum_{j=0}^{n-1} A_{ij} x_{j}$
  - d.h. Spezialfall von Matrix-Matrix-Produkt
- Interne Realisierung von `A*x`
  - `x` ist `Vector`, insb. `Matrix` mit Dimension $n \times 1$
  - `A*x` ist `Matrix` mit Dimension $m \times 1$
  - ggf. impliziter Cast auf `Vector`

Testbeispiel

```
 1 #include "matrix.hpp"
 2 #include "matrix-vector.hpp"
 3 using std::cout;
 4
 5 int main() {
 6   int n = 3;
 7   Matrix A(n,n);
 8   for (int j=0; j<n; ++j) {
 9     A(j,j) = j+1;
10   }
11   cout << A;
12   Vector X(n,1);
13   Vector Y = A*X;
14   cout << Y;
15   return 0;
16 }
```

- Output:

```
constructor, 3 x 3
 1 0 0
 0 2 0
 0 0 3
constructor, 3 x 1
vector constructor, size 3
constructor, 3 x 1
constructor, 3 x 1
type cast Matrix -> Vector
destructor, 3 x 1
 1
 2
 3
destructor, 3 x 1
destructor, 3 x 1
destructor, 3 x 3
```

Schlüsselwort `virtual`

- Polymorphie
- Virtuelle Methoden

- `virtual`

Polymorphie

- Jedes Objekt der abgeleiteten Klasse ist auch ein Objekt der Basisklasse
  - Vererbung impliziert immer ist-ein-Beziehung
- Jede Klasse definiert einen Datentyp
  - Objekte können mehrere Typen haben
  - Objekte abgeleiteter Klassen haben mindestens zwei Datentypen:
    - Typ der abgeleiteten Klasse
    - und Typ der Basisklasse
    - **BSP:** im letztem Beispiel ist Vektor vom Typ `Vector` und `Matrix`
- kann den jeweils passenden Typ verwenden
  - Diese Eigenschaft nennt man Polymorphie (*griech.* Vielgestaltigkeit)
- Das hat insbesondere Konsequenzen für Pointer!

Pointer und `virtual` 1/3

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 public:
 6   void print() {cout << "Basisklasse\n";}
 7 };
 8
 9 class Abgeleitet : public Basisklasse {
10 public:
11   void print() {cout << "Abgeleitet\n";}
12 };
13
14 int main() {
15   Abgeleitet a;
16   Abgeleitet* pA = &a;
17   Basisklasse* pB = &a;
18   pA->print();
19   pB->print();
20   return 0;
21 }
```

- Output:

  ```
  Abgeleitet
  Basisklasse
  ```

- Zeile 15: Objekt a vom Typ `Abgeleitet` ist auch vom Typ `Basisklasse`
- Pointer auf `Basisklasse` mit Adresse von `a` möglich
- Zeile 19 ruft `print` aus `Basisklasse` auf
  - i.a. soll `print` aus `Abgeleitet` verwendet werden

Pointer und `virtual` 2/3

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 public:
 6   virtual void print() {cout << "Basisklasse\n";}
 7 };
 8
 9 class Abgeleitet : public Basisklasse {
10 public:
11   void print() {cout << "Abgeleitet\n";}
12 };
13
14 int main() {
15   Abgeleitet a;
16   Abgeleitet* pA = &a;
17   Basisklasse* pB = &a;
18   pA->print();
19   pB->print();
20   return 0;
21 }
```

- Output:

  ```
  Abgeleitet
  Abgeleitet
  ```

- Zeile 6: neues Schlüsselwort `virtual`
  - vor Signatur der Methode `print` (in Basisklasse!)
  - deklariert virtuelle Methode
  - zur Laufzeit wird korrekte Methode aufgerufen
    - **Varianten müssen gleiche Signatur haben**
  - Zeile 19 ruft nun redefinierte Methode `print` auf

Pointer und `virtual` 3/3

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 public:
 6   virtual void print() {cout << "Basisklasse\n";}
 7 };
 8
 9 class Abgeleitet1 : public Basisklasse {
10 public:
11   void print() {cout << "Nummer 1\n";}
12 };
13
14 class Abgeleitet2 : public Basisklasse {
15 public:
16   void print() {cout << "Nummer 2\n";}
17 };
18
19 int main() {
20   Basisklasse* var[2];
21   var[0] = new Abgeleitet1;
22   var[1] = new Abgeleitet2;
23
24   for (int j=0; j<2; ++j) {
25     var[j]->print();
26   }
27   return 0;
28 }
```

- Output:

  ```
  Nummer 1
  Nummer 2
  ```

- `var` ist Vektor mit Objekten verschiedener Typen!

Destruktor und `virtual` 1/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 public:
 6   ~Basisklasse() {
 7     cout << "~Basisklasse()\n";
 8   }
 9 };
10
11 class Abgeleitet : public Basisklasse {
12 public:
13   ~Abgeleitet() {
14     cout << "~Abgeleitet()\n";
15   }
16 };
17
18 int main() {
19   Basisklasse* var = new Abgeleitet;
20   delete var;
21   return 0;
22 }
```

- Output:

  ```
  ~Basisklasse()
  ```

- Destruktor von `Abgeleitet` wird nicht aufgerufen!
  - ggf. entsteht toter Speicher, falls `Abgeleitet` zusätzlichen dynamischen Speicher anlegt
- Destruktoren werden deshalb üblicherweise als virtual deklariert

Destruktor und `virtual` 2/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 public:
 6   virtual ~Basisklasse() {
 7     cout << "~Basisklasse()\n";
 8   }
 9 };
10
11 class Abgeleitet : public Basisklasse {
12 public:
13   ~Abgeleitet() {
14     cout << "~Abgeleitet()\n";
15   }
16 };
17
18 int main() {
19   Basisklasse* var = new Abgeleitet;
20   delete var;
21   return 0;
22 }
```

- Output:

  ```
  ~Abgeleitet()
  ~Basisklasse()
  ```

- Destruktor von `Abgeleitet` wird aufgerufen
  - ruft implizit Destruktor von `Basisklasse` auf

Virtuelle Methoden 1/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 public:
 6   void ego() { cout << "Basisklasse\n"; }
 7   void print() { cout << "Ich bin "; ego(); }
 8 };
 9
10 class Abgeleitet1: public Basisklasse {
11 public:
12   void ego() { cout << "Nummer 1\n"; }
13 };
14
15 class Abgeleitet2: public Basisklasse {};
16
17 int main() {
18   Basisklasse var0;
19   Abgeleitet1 var1;
20   Abgeleitet2 var2;
21   var0.print();
22   var1.print();
23   var2.print();
24   return 0;
25 }
```

- Output:

  ```
  Ich bin Basisklasse
  Ich bin Basisklasse
  Ich bin Basisklasse
  ```

- Obwohl `ego` redefiniert wird für `Abgeleitet1`, bindet `print` immer `ego` von `Basisklasse` ein

Virtuelle Methoden 2/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 public:
 6   virtual void ego() { cout << "Basisklasse\n"; }
 7   void print() { cout << "Ich bin "; ego(); }
 8 };
 9
10 class Abgeleitet1: public Basisklasse {
11 public:
12   void ego() { cout << "Nummer 1\n"; }
13 };
14
15 class Abgeleitet2: public Basisklasse {};
16
17 int main() {
18   Basisklasse var0;
19   Abgeleitet1 var1;
20   Abgeleitet2 var2;
21   var0.print();
22   var1.print();
23   var2.print();
24   return 0;
25 }
```

- Output:

  ```
  Ich bin Basisklasse
  Ich bin Nummer 1
  Ich bin Basisklasse
  ```

- `virtual` (Zeile 6) sorgt für korrekte Einbindung, falls diese für abgeleitete Klasse redefiniert ist

Abstrakte Klassen

- Manchmal werden Klassen nur zur Strukturierung / zum Vererben angelegt, aber Instanzierung ist nicht sinnvoll / nicht gewollt
  - d.h. es soll keine Objekte der Basisklasse geben
  - sog. **abstrakte Klassen**
  - dient nur als Schablone für abgeleitete Klassen

- abstrakte Klassen können nicht instanziert werden
  - Compiler liefert Fehlermeldung!

- In C++ ist eine Klasse abstrakt, falls eine Methode existiert der Form

  ```
  virtual return-type method( ... ) = 0;
  ```

  - Diese sog. **abstrakte Methode** muss in allen abgeleiteten Klassen implementiert werden
    - wird nicht in Basisklasse implementiert

Beispiel zu abstrakten Klassen

```
 1 #include <cmath>
 2
 3 class Figure {
 4 private:
 5   double centerOfMass[2];
 6 public:
 7   virtual double getArea() = 0;
 8 };
 9
10 class Circle : public Figure {
11 private:
12   double radius;
13 public:
14   double getArea() {
15     return radius*radius*3.14159;
16   }
17 };
18
19 class Triangle : public Figure {
20 private:
21   double a[2],b[2],c[2];
22 public:
23   double getArea() {
24     return fabs(0.5*( (b[0]-a[0])*(c[1]-a[1])
25                      -(c[0]-a[0])*(b[1]-a[1])));
26   }
27 };
```

- Abstrakte Klasse `Figure`
  - durch abstrakte Methode `getArea` (Zeile 6)
- abgeleitete Klassen `Circle`, `Triangle`
- `Circle` und `Triangle` redefinieren `getArea`
  - alle abstrakten Meth. müssen redefiniert werden

Beispiel zu `virtual`: Matrizen

- für allgemeine Matrix $A \in \mathbb{R}^{m \times n}$
  - Vektoren $x \in \mathbb{R}^m \simeq \mathbb{R}^{m \times 1}$
  - quadratische Matrix $A \in \mathbb{R}^{n \times n}$
    - reguläre Matrix: $\det(A) \neq 0$
    - symmetrische Matrix: $A = A^T$
    - untere Dreiecksmatrix, $A_{jk} = 0$ für $k > j$
    - obere Dreiecksmatrix, $A_{jk} = 0$ für $k < j$
- symmetrischen Matrizen und Dreiecksmatrizen brauchen generisch weniger Speicher
  - $\frac{n(n+1)}{2}$ statt n^2
- muss Koeffizientenzugriff überladen
  - Koeffizientenzugriff in `Matrix` muss `virtual` sein, damit Methoden für `Matrix` z.B. auch für symmetrische Matrizen anwendbar

matrix.hpp 1/3

```
 1 #ifndef _MATRIX_
 2 #define _MATRIX_
 3 #include <cmath>
 4 #include <cassert>
 5 #include <iostream>
 6
 7 class Matrix {
 8 private:
 9   int m;
10   int n;
11   int storage;
12   double* coeff;
13
14 protected:
15   // methods such that subclasses can access data fields
16   void allocate(int m, int n, int storage, double init);
17   const double* getCoeff() const;
18   double* getCoeff();
19   int getStorage() const;
```

- abgeleitete Klassen, z.B. SymmetricMatrix können auf Datenfelder nicht zugreifen, da hidden nach Vererbung
  - muss Zugriffsfunktionen schaffen
  - protected stellt sicher, dass diese Methoden nur in den abgeleiteten Klassen verwendet werden können (aber nicht von Außen!)
- SymmetricMatrix hat weniger Speicher als Matrix
  - muss Allocation als Methode bereitstellen
    - $m \cdot n$ Speicherplätze für $A \in \mathbb{R}^{m \times n}$
    - nur $\frac{n(n+1)}{2} = \sum_{i=1}^{n} i$ für $A = A^T \in \mathbb{R}^{n \times n}$

matrix.hpp 2/3

```
21 public:
22   // constructors, destructor, assignment
23   Matrix();
24   Matrix(int m, int n, double init=0);
25   Matrix(const Matrix&);
26   ~Matrix();
27   Matrix& operator=(const Matrix&);
28
29   // return size of matrix
30   int size1() const;
31   int size2() const;
32
33   // read and write entries with matrix access A(j,k)
34   virtual const double& operator()(int j, int k) const;
35   virtual double& operator()(int j, int k);
```

- Destruktor nicht virtuell, da abgeleitete Klassen keinen dynamischen Speicher haben
- Koeffizienten-Zugriff muss `virtual` sein, da z.B. symmetrische Matrizen anders gespeichert
  - `virtual` nur in Klassendefinition, d.h. generisch im Header-File
- Funktionalität wird mittels Koeff.-Zugriff `A(j,k)` realisiert, z.B. `operator+`, `operator*`
  - kann alles auch für symm. Matrizen nutzen
  - nur 1x für Basisklasse implementieren
    - manchmal ist Redefinition sinnvoll für effizientere Lösung

matrix.hpp 3/3

```
37   // read and write storage vector A[ell]
38   const double& operator[](int ell) const;
39   double& operator[](int ell);
40
41   // compute norm
42   double norm() const;
43 };
44
45 // print matrix via output stream
46 std::ostream& operator<<(std::ostream& output,
47                                        const Matrix&);
48
49 // matrix-matrix sum and product
50 const Matrix operator+(const Matrix&, const Matrix&);
51 const Matrix operator*(const Matrix&, const Matrix&);
52
53 #endif
```

- Operator [ ] für effiziente Implementierung
  - z.B. Addition bei gleichem Matrix-Typ
  - d.h. bei gleicher interner Speicherung
    - muss nur Speichervektoren addieren
- Implementierung von norm, operator+, operator* mittels Koeffizienten-Zugriff A(j,k)
  - direkt für abgeleitete Klassen anwendbar

matrix.cpp 1/5

```
 1 #include "matrix.hpp"
 2 using std::cout;
 3 using std::ostream;
 4
 5 void Matrix::allocate(int m, int n, int storage,
 6                                                 double init) {
 7   assert(m>=0);
 8   assert(n>=0);
 9   assert(storage>=0 && storage<=m*n);
10   this->m = m;
11   this->n = n;
12   this->storage = storage;
13   if (storage>0) {
14     coeff = new double[storage];
15     for (int ell=0; ell<storage; ++ell) {
16       coeff[ell] = init;
17     }
18   }
19   else {
20     coeff = (double*) 0;
21   }
22 }
23
24 const double* Matrix::getCoeff() const {
25   return coeff;
26 }
27
28 double* Matrix::getCoeff() {
29   return coeff;
30 }
31
32 int Matrix::getStorage() const {
33   return storage;
34 }
```

matrix.cpp 2/5

```
36 Matrix::Matrix() {
37   m = 0;
38   n = 0;
39   storage = 0;
40   coeff = (double*) 0;
41   cout << "Matrix: empty constructor\n";
42 }
43
44 Matrix::Matrix(int m, int n, double init) {
45   allocate(m,n,m*n,init);
46   cout << "Matrix: constructor, "
47        << m << " x " << n << "\n";
48 }
49
50 Matrix::Matrix(const Matrix& rhs) {
51   m = rhs.m;
52   n = rhs.n;
53   storage = m*n;
54   if (storage > 0) {
55     coeff = new double[storage];
56     for (int j=0; j<m; ++j) {
57       for (int k=0; k<n; ++k) {
58         (*this)(j,k) = rhs(j,k);
59       }
60     }
61   }
62   else {
63     coeff = (double*) 0;
64   }
65   cout << "Matrix: copy constructor, "
66        << m << " x " << n << "\n";
67 }
68
69 Matrix::~Matrix() {
70   if (storage > 0) {
71     delete[] coeff;
72   }
73   cout << "Matrix: destructor, "
74        << m << " x " << n << "\n";
75 }
```

matrix.cpp 3/5

```
 77 Matrix& Matrix::operator=(const Matrix& rhs) {
 78   if (this != &rhs) {
 79     if ( (m != rhs.m) || (n != rhs.n) ) {
 80       if (storage > 0) {
 81         delete[] coeff;
 82       }
 83       m = rhs.m;
 84       n = rhs.n;
 85       storage = m*n;
 86       if (storage > 0) {
 87         coeff = new double[storage];
 88       }
 89       else {
 90         coeff = (double*) 0;
 91       }
 92     }
 93     for (int j=0; j<m; ++j) {
 94       for (int k=0; k<n; ++k) {
 95         (*this)(j,k) = rhs(j,k);
 96       }
 97     }
 98     cout << "Matrix: deep copy, "
 99          << m << " x " << n << "\n";
100   }
101   return *this;
102 }
103
104 int Matrix::size1() const {
105   return m;
106 }
107
108 int Matrix::size2() const {
109   return n;
110 }
```

- Zeile 78: Code sicher Selbst-Zuweisung `A = A`
  - keine Aktion, nur `return *this;`

matrix.cpp 4/5

```
112 const double& Matrix::operator()(int j, int k) const {
113   assert(j>=0 && j<m);
114   assert(k>=0 && k<n);
115   return coeff[j+k*m];
116 }
117
118 double& Matrix::operator()(int j, int k) {
119   assert(j>=0 && j<m);
120   assert(k>=0 && k<n);
121   return coeff[j+k*m];
122 }
123
124 const double& Matrix::operator[](int ell) const {
125   assert( ell>=0 && ell<storage );
126   return coeff[ell];
127 }
128
129 double& Matrix::operator[](int ell) {
130   assert( ell>=0 && ell<storage );
131   return coeff[ell];
132 }
133
134 double Matrix::norm() const {
135   double norm = 0;
136   for (int j=0; j<m; ++j) {
137     for (int k=0; k<n; ++k) {
138       norm = norm + (*this)(j,k) * (*this)(j,k);
139     }
140   }
141   return sqrt(norm);
142 }
143
144 ostream& operator<<(ostream& output, const Matrix& A) {
145   output << "\n";
146   for (int j=0; j<A.size1(); j++) {
147     for (int k=0; k<A.size2(); k++) {
148       output << " " << A(j,k);
149     }
150     output << "\n";
151   }
152   return output;
```

matrix.cpp 5/5

```
155 const Matrix operator+(const Matrix& A, const Matrix& B) {
156   int m = A.size1();
157   int n = A.size2();
158   assert(m == B.size1() );
159   assert(n == B.size2() );
160   Matrix sum(m,n);
161   for (int j=0; j<m; ++j) {
162     for (int k=0; k<n; ++k) {
163       sum(j,k) = A(j,k) + B(j,k);
164     }
165   }
166   return sum;
167 }
168
169 const Matrix operator*(const Matrix& A, const Matrix& B) {
170   int m = A.size1();
171   int n = A.size2();
172   int p = B.size2();
173   double sum = 0;
174   assert(n == B.size1() );
175   Matrix product(m,p);
176   for (int i=0; i<m; ++i) {
177     for (int k=0; k<p; ++k) {
178       sum = 0;
179       for (int j=0; j<n; ++j) {
180         sum = sum + A(i,j)*B(j,k);
181       }
182       product(i,k) = sum;
183     }
184   }
185   return product;
186 }
```

- Addition: $A + B \in \mathbb{R}^{m \times n}$, $(A + B)_{j\ell} = A_{j\ell} + B_{j\ell}$
- Multiplikation: $AB \in \mathbb{R}^{m \times p}$, $(AB)_{j\ell} = \sum_{k=1}^{n} A_{jk} B_{k\ell}$
  - $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$

Bemerkungen

- da Matrizen spaltenweise gespeichert sind, sollte man eigentlich bei der Reihenfolge der Schleifen beachten, z.B.

```
for (int j=0; j<m; ++j) {
  for (int k=0; k<n; ++k) {
    (*this)(j,k) = rhs(j,k);
  }
}
```

  besser ersetzen durch

```
for (int k=0; k<n; ++k) {
  for (int j=0; j<m; ++j) {
    (*this)(j,k) = rhs(j,k);
  }
}
```

- Speicherzugriff ist dann schneller,
  - es wird nicht im Speicher herumgesprungen
- weitere Funktionen zu `Matrix` sind denkbar
  - Vorzeichen
  - Skalarmultiplikation
  - Matrizen subtrahieren
- weitere Methoden zu `Matrix` sind denkbar
  - Matrix transponieren

squareMatrix.hpp

```
 1 #ifndef _SQUAREMATRIX_
 2 #define _SQUAREMATRIX_
 3 #include "matrix.hpp"
 4 #include <cassert>
 5 #include <iostream>
 6
 7 class SquareMatrix : public Matrix {
 8 public:
 9   // constructors, destructor, type cast from Matrix
10   SquareMatrix();
11   SquareMatrix(int n,double init=0);
12   SquareMatrix(const SquareMatrix&);
13   ~SquareMatrix();
14   SquareMatrix(const Matrix&);
15
16   // further members
17   int size() const;
18 };
19
20 #endif
```

- Jede quadratische Matrix $A \in \mathbb{R}^{n \times n}$ ist insb. eine allgemeine Matrix $A \in \mathbb{R}^{m \times n}$
  - zusätzliche Funktion: z.B. det(A) berechnen
  - hier wird nur `SquareMatrix` von `Matrix` abgeleitet
  - keine zusätzliche Funktionalität, nur
    - Standardkonstruktor und Konstruktor
    - Kopierkonstruktor
    - Destruktor
    - Type Cast `Matrix` auf `SquareMatrix`
    - `size` als Vereinfachung von `size1`, `size2`

squareMatrix.cpp

```
 1 #include "squareMatrix.hpp"
 2 using std::cout;
 3
 4 SquareMatrix::SquareMatrix() {
 5   cout << "SquareMatrix: empy constructor\n";
 6 }
 7
 8 SquareMatrix::SquareMatrix(int n, double init) :
 9                                          Matrix(n,n,init) {
10   cout << "SquareMatrix: constructor, " << size() << "\n";
11 };
12
13 SquareMatrix::SquareMatrix(const SquareMatrix& rhs) :
14                                                Matrix(rhs) {
15   cout << "SquareMatrix: copy constructor, "
16        << size() << "\n";
17 }
18
19 SquareMatrix::~SquareMatrix() {
20   cout << "SquareMatrix: destructor, " << size() << "\n";
21 }
22
23 SquareMatrix::SquareMatrix(const Matrix& rhs) :
24                                                Matrix(rhs) {
25   assert(size1() == size2());
26   cout << "type cast Matrix -> SquareMatrix\n";
27 }
28
29 int SquareMatrix::size() const {
30   return size1();
31 }
```

- Type Cast garantiert, dass `rhs` $\in \mathbb{R}^{m \times n}$ mit $m = n$
  - d.h. Konversion auf `SquareMatrix` ohne Verlust
- theoretisch auch Cast durch Abschneiden sinnvoll
  - hier aber anders!

Demo zu squareMatrix 1/5

```
 1 #include "matrix.hpp"
 2 #include "squareMatrix.hpp"
 3
 4 using std::cout;
 5
 6 int main() {
 7   int n = 3;
 8
 9   cout << "*** init A\n";
10   SquareMatrix A(n);
11   for (int ell=0; ell<n*n; ++ell) {
12     A[ell] = ell;
13   }
14   cout << "A =" << A;
15
16   cout << "*** init B\n";
17   Matrix B = A;
18   cout << "B =" << B;
19
20   cout << "*** init C\n";
21   SquareMatrix C = A;
22   cout << "C = " << C;
23
24   cout << "*** C = A + A\n";
25   C = A + A;
26   cout << "C = " << C;
27
28   cout << "*** init D\n";
29   SquareMatrix D = A + B;
30   cout << "D =" << C;
31
32   cout << "*** terminate\n";
33   return 0;
34 }
```

- erwartetes Resultat:
  - $A = \begin{pmatrix} 0 & 3 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \end{pmatrix} = B, \quad C = \begin{pmatrix} 0 & 6 & 12 \\ 2 & 8 & 14 \\ 4 & 10 & 16 \end{pmatrix} = D,$

Demo zu squareMatrix 2/5

```
 1 #include "matrix.hpp"
 2 #include "squareMatrix.hpp"
 3
 4 using std::cout;
 5
 6 int main() {
 7   int n = 3;
 8
 9   cout << "*** init A\n";
10   SquareMatrix A(n);
11   for (int ell=0; ell<n*n; ++ell) {
12     A[ell] = ell;
13   }
14   cout << "A =" << A;
```

- Output:

```
*** init A
Matrix: constructor, 3 x 3
SquareMatrix: constructor, 3
A =
 0 3 6
 1 4 7
 2 5 8
```

- man sieht spaltenweise Speicherung von A

Demo zu squareMatrix 3/5

```
16   cout << "*** init B\n";
17   Matrix B = A;
18   cout << "B =" << B;
19
20   cout << "*** init C\n";
21   SquareMatrix C = A;
22   cout << "C = " << C;
```

- Output:

```
*** init B
Matrix: copy constructor, 3 x 3
B =
 0 3 6
 1 4 7
 2 5 8
*** init C
Matrix: copy constructor, 3 x 3
SquareMatrix: copy constructor, 3
C =
 0 3 6
 1 4 7
 2 5 8
```

Demo zu squareMatrix 4/5

```
24   cout << "*** C = A + A\n";
25   C = A + A;
26   cout << "C = " << C;
```

- Output:

```
*** C = A + A
Matrix: constructor, 3 x 3
Matrix: copy constructor, 3 x 3
type cast Matrix -> SquareMatrix
Matrix: deep copy, 3 x 3
SquareMatrix: destructor, 3
Matrix: destructor, 3 x 3
Matrix: destructor, 3 x 3
C =
 0 6 12
 2 8 14
 4 10 16
```

Demo zu squareMatrix 5/5

```
28   cout << "*** init D\n";
29   SquareMatrix D = A + B;
30   cout << "D =" << C;
31
32   cout << "*** terminate\n";
33   return 0;
34 }
```

- Output:

```
*** init D
Matrix: constructor, 3 x 3
Matrix: copy constructor, 3 x 3
type cast Matrix -> SquareMatrix
Matrix: destructor, 3 x 3
D =
 0 6 12
 2 8 14
 4 10 16
*** terminate
SquareMatrix: destructor, 3
Matrix: destructor, 3 x 3
SquareMatrix: destructor, 3
Matrix: destructor, 3 x 3
Matrix: destructor, 3 x 3
SquareMatrix: destructor, 3
Matrix: destructor, 3 x 3
```

lowerTriangularMatrix.hpp

```
 1 #ifndef _LOWERTRIANGULARMATRIX_
 2 #define _LOWERTRIANGULARMATRIX_
 3 #include "squareMatrix.hpp"
 4 #include <cassert>
 5 #include <iostream>
 6
 7 class LowerTriangularMatrix : public SquareMatrix {
 8 private:
 9   double zero;
10   double const_zero;
11
12 public:
13   // constructors, destructor, type cast from Matrix
14   LowerTriangularMatrix();
15   LowerTriangularMatrix(int n, double init=0);
16   LowerTriangularMatrix(const LowerTriangularMatrix&);
17   ~LowerTriangularMatrix();
18   LowerTriangularMatrix(const Matrix&);
19
20   // assignment operator
21   LowerTriangularMatrix& operator=(
22                            const LowerTriangularMatrix&);
23
24   // read and write entries with matrix access A(j,k)
25   virtual const double& operator()(int j, int k) const;
26   virtual double& operator()(int j, int k);
27 };
28 #endif
```

- eine Matrix $A \in \mathbb{R}^{n \times n}$ ist untere Dreiecksmatrix, falls $A_{jk} = 0$ für $k > j$
  - d.h. $A = \begin{pmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$ für $n = 3$
- muss nur $\frac{n(n+1)}{2} = \sum_{i=1}^{n} i$ Einträge speichern
- zeilenweise Speicherung: $A_{jk} = a_\ell$ mit $\ell = \frac{j(j+1)}{2} + k$
  - muss Matrix-Zugriff ( ) redefinieren

lowerTriangularMatrix.cpp 1/4

```
 1 #include "lowerTriangularMatrix.hpp"
 2 using std::cout;
 3
 4 LowerTriangularMatrix::LowerTriangularMatrix() {
 5   cout << "LowerTriangular: empty constructor\n";
 6 }
 7
 8 LowerTriangularMatrix::LowerTriangularMatrix(int n,
 9                                               double init) {
10   zero = 0;
11   const_zero = 0;
12   allocate(n, n, n*(n+1)/2, init);
13   cout << "LowerTriangular: constructor, " << n << "\n";
14 }
15
16 LowerTriangularMatrix::LowerTriangularMatrix(
17                          const LowerTriangularMatrix& rhs) {
18   int n = rhs.size();
19   allocate(n, n, n*(n+1)/2, 0);
20   zero = 0;
21   const_zero = 0;
22   for (int ell=0; ell<n*(n+1)/2; ++ell) {
23     (*this)[ell] = rhs[ell];
24   }
25   cout << "LowerTriangular: copy constructor, "
26        << n << "\n";
27 }
```

- `private` Member `zero`, `const_zero` haben Wert 0
  - dienen für Zugriff auf $A_{jk} = 0$ für $k > j$
- Kopierkonstruktor für Objekte der eigenen Klasse
  - sonst würde Kopierkonstruktor der Basisklasse `SquareMatrix` verwendet!

lowerTriangularMatrix.cpp 2/4

```
29 LowerTriangularMatrix::LowerTriangularMatrix(
30                                          const Matrix& rhs) {
31   int n = rhs.size1();
32   assert (n == rhs.size2());
33   allocate(n, n, n*(n+1)/2, 0);
34   zero = 0;
35   const_zero = 0;
36   for (int j=0; j<n; ++j) {
37     for (int k=0; k<=j; ++k) {
38       (*this)(j,k) = rhs(j,k);
39     }
40     for (int k=j+1; k<n; ++k) {
41       assert( rhs(j,k) == 0);
42     }
43   }
44   cout << "type cast Matrix -> LowerTriangular\n";
45 }
46
47 LowerTriangularMatrix::~LowerTriangularMatrix() {
48   cout << "LowerTriangular: destructor, "
49        << size() << "\n";
50 }
```

- Type Cast kontrolliert, dass `rhs` $\in \mathbb{R}^{n \times n}$ eine untere Dreiecksmatrix ist
  - wird verwendet, falls `rhs` kein Objekt der Klasse `LowerTriangularMatrix`
- beachte unterschiedliche `( )` in Zeile 38

lowerTriangularMatrix.cpp 3/4

```
52 LowerTriangularMatrix& LowerTriangularMatrix::operator=(
53                       const LowerTriangularMatrix& rhs) {
54
55   if (this != &rhs) {
56     int n = rhs.size();
57     if (size() != n) {
58       if (size() > 0) {
59         delete[] getCoeff();
60       }
61       allocate(n, n, n*(n+1)/2, 0);
62     }
63     for (int ell=0; ell<n*(n+1)/2; ++ell) {
64       (*this)[ell] = rhs[ell];
65     }
66     cout << "LowerTriangular: deep copy "
67          << size() << "\n";
68   }
69   return *this;
70 }
```

- Redefinition des Zuweisungsoperators nötig, da sonst geerbt von `Matrix`
  - Speichervektor von `LowerTriangularMatrix` ist anders als der von `Matrix`
- analog zu Kopierkonstruktor

lowerTriangularMatrix.cpp 4/4

```
72 const double& LowerTriangularMatrix::operator()(
73                                         int j, int k) const {
74   assert( j>=0 && j<size() );
75   assert( k>=0 && k<size() );
76   if ( j < k ) {
77     return const_zero;
78   }
79   else {
80     const double* coeff = getCoeff();
81     return coeff[j*(j+1)/2+k];
82   }
83 }
84
85 double& LowerTriangularMatrix::operator()(int j, int k) {
86   assert( j>=0 && j<size() );
87   assert( k>=0 && k<size() );
88   if ( j < k ) {
89     zero = 0;
90     return zero;
91   }
92   else {
93     double* coeff = getCoeff();
94     return coeff[j*(j+1)/2+k];
95   }
96 }
```

- Jedes Objekt der Klasse `LowerTriangularMatrix` ist auch Objekt der Klassen `SquareMatrix` und `Matrix`
- Redefinition von Matrix-Zugriff `A(j,k)`
- Garantiere $A_{jk} = 0$ für $k > j$, damit Methoden aus `Matrix` genutzt werden können
  - `const_zero` hat stets Wert 0 (durch Konstruktor)
    - Benutzer kann nicht schreibend zugreifen
  - `zero` wird explizit immer auf 0 gesetzt
    - Benutzer könnte auch schreibend zugreifen

Demo zu lowerTriangularMatrix 1/7

```
 1 #include "matrix.hpp"
 2 #include "squareMatrix.hpp"
 3 #include "lowerTriangularMatrix.hpp"
 4
 5 using std::cout;
 6
 7 int main() {
 8   int n = 3;
 9
10   cout << "*** init A\n";
11   SquareMatrix A(n,1);
12   cout << "A =" << A;
```

- Output:

```
*** init A
Matrix: constructor, 3 x 3
SquareMatrix: constructor, 3
A =
 1 1 1
 1 1 1
 1 1 1
```

Demo zu lowerTriangularMatrix 2/7

```
14   cout << "*** init B\n";
15   LowerTriangularMatrix B(n);
16   for (int ell=0; ell<n*(n+1)/2; ++ell) {
17     B[ell] = 2;
18   }
19   B(0,n-1) = 10; //*** hat keinen Effekt!
20   cout << "B =" << B;
```

- Output:

```
*** init B
Matrix: empty constructor
SquareMatrix: empy constructor
LowerTriangular: constructor, 3
B =
 2 0 0
 2 2 0
 2 2 2
```

Demo zu lowerTriangularMatrix 3/7

```
22   cout << "*** init C\n";
23   Matrix C = A + B;
24   cout << "C =" << C;
25
26   cout << "*** init D\n";
27   LowerTriangularMatrix D(n);
28   for (int ell=0; ell<n*(n+1)/2; ++ell) {
29     D[ell] = ell;
30   }
31   cout << "D =" << D;
```

- Output:

```
*** init C
Matrix: constructor, 3 x 3
C =
 3 1 1
 3 3 1
 3 3 3
*** init D
Matrix: empty constructor
SquareMatrix: empy constructor
LowerTriangular: constructor, 3
D =
 0 0 0
 1 2 0
 3 4 5
```

- Erinnerung: $A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$, $B = \begin{pmatrix} 2 & 0 & 0 \\ 2 & 2 & 0 \\ 2 & 2 & 2 \end{pmatrix}$

Demo zu lowerTriangularMatrix 4/7

```
32   cout << "--\n";
33   D = D + B;
34   cout << "--\n";
35   cout << "D =" << D;
```

- Output:

```
--
Matrix: constructor, 3 x 3
Matrix: empty constructor
SquareMatrix: empy constructor
type cast Matrix -> LowerTriangular
LowerTriangular: deep copy 3
LowerTriangular: destructor, 3
SquareMatrix: destructor, 3
Matrix: destructor, 3 x 3
Matrix: destructor, 3 x 3
--
D =
 2 0 0
 3 4 0
 5 6 7
```

- Erinnerung: $B = \begin{pmatrix} 2 & 0 & 0 \\ 2 & 2 & 0 \\ 2 & 2 & 2 \end{pmatrix}$, $D = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 2 & 0 \\ 3 & 4 & 5 \end{pmatrix}$

Demo zu lowerTriangularMatrix 5/7

```
37   cout << "*** init E\n";
38   LowerTriangularMatrix E = D;
39   cout << "E = " << E;
40
41   cout << "*** A = D\n";
42   A = D;
43   cout << "A =" << A;
```

- Output:

```
*** init E
Matrix: empty constructor
SquareMatrix: empy constructor
LowerTriangular: copy constructor, 3
E =
 2 0 0
 3 4 0
 5 6 7
*** A = D
Matrix: deep copy, 3 x 3
A =
 2 0 0
 3 4 0
 5 6 7
```

- Erinnerung: $D = \begin{pmatrix} 2 & 0 & 0 \\ 3 & 4 & 0 \\ 5 & 6 & 7 \end{pmatrix}$
- A ist SquareMatrix
- D ist LowerTriangularMatrix

Demo zu lowerTriangularMatrix 6/7

```
45   cout << "*** B = D\n";
46   B = D;
47   cout << "B =" << B;
48
49   cout << "*** B = A\n";
50   B = A;
51   cout << "B =" << B;
```

- Output:

```
*** B = D
LowerTriangular: deep copy 3
B =
 2 0 0
 3 4 0
 5 6 7
*** B = A
Matrix: empty constructor
SquareMatrix: empy constructor
type cast Matrix -> LowerTriangular
LowerTriangular: deep copy 3
LowerTriangular: destructor, 3
SquareMatrix: destructor, 3
Matrix: destructor, 3 x 3
B =
 2 0 0
 3 4 0
 5 6 7
```

- Erinnerung: A ist SquareMatrix
- B und D sind LowerTriangularMatrix

Demo zu lowerTriangularMatrix 7/7

```
53   cout << "*** terminate\n";
54   return 0;
55 }
```

- Output:

```
*** terminate
LowerTriangular: destructor, 3
SquareMatrix: destructor, 3
Matrix: destructor, 3 x 3
LowerTriangular: destructor, 3
SquareMatrix: destructor, 3
Matrix: destructor, 3 x 3
Matrix: destructor, 3 x 3
LowerTriangular: destructor, 3
SquareMatrix: destructor, 3
Matrix: destructor, 3 x 3
SquareMatrix: destructor, 3
Matrix: destructor, 3 x 3
```

- Erinnerung: Allokationsreihenfolge
  - A ist SquareMatrix
  - B ist LowerTriangularMatrix
  - C ist Matrix
  - D ist LowerTriangularMatrix
  - E ist LowerTriangularMatrix

Templates

- Was sind Templates?
- Funktionentemplates
- Klassentemplates

- `template`

Generische Programmierung

- Wieso Umstieg auf höhere Programmiersprache?
  - Mehr Funktionalität (Wiederverwendbarkeit/Wartbarkeit)
  - haben wir bei Vererbung ausgenutzt
- Ziele:
  - möglichst wenig Code selbst schreiben
  - Gemeinsamkeiten wiederverwenden
  - nur Modifikationen implementieren
- Oftmals ähnlicher Code für verschiedene Dinge
- Vererbung bietet sich oft nicht an
  - es liegt nicht immer Ist-Ein-Beziehung vor
- Idee: Code unabhängig vom Datentyp entwickeln
- Führt auf generische Programmierung

Beispiel: Maximum / Quadrieren

```
 1 int max(int a, int b) {
 2   if (a < b)
 3     return b;
 4   else
 5     return a;
 6 }
 7
 8 double max(double a, double b) {
 9   if (a < b)
10     return b;
11   else
12     return a;
13 }
14
15 int square(int a) {
16   return a*a;
17 }
18
19 double square(double a) {
20   return a*a;
21 }
```

- Ziel: Maximum berechnen / quadrieren
- Gleicher Code für viele Probleme
  - Vererbung bietet sich hier nicht an
- **Lösung:** Templates

Funktionstemplate 1/2

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 template <typename Type>
 6 Type square(const Type& var) {
 7   return var*var;
 8 }
 9
10 int main() {
11   cout << square<double>(1.5) << endl;
12   cout << square(1.5) << endl;
13   cout << square<int>(1.5) << endl;
14 }
```

- `template <typename Type> RetType fct(input)`
  - analog zu normaler Funktionsdeklaration
  - `Type` ist dann variabler Input/Output-Datentyp
  - Referenzen und Pointer auf `Type` möglich
- theoretisch mehrere variable Datentypen möglich
  - `template <typename Type1, typename Type2> ...`
- Funktion `square` kann aufgerufen werden, falls
  - `var` Objekt vom Typ `Type`
  - Datentyp `Type` hat Multiplikation `*`
- bei Aufruf Datentyp in spitzen Klammern (Z. 11)
  - oder implizit (Zeile 12)
- Output:

  2.25

  2.25

  1

Funktionstemplate 2/2

- Was passiert eigentlich bei folgendem Code?

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 template <typename Type>
 6 Type square(const Type& t) {
 7   return t*t;
 8 }
 9
10 int main() {
11   int x = 2;
12   double y = 4.7;
13   cout << square(x) << endl;
14   cout << square(y) << endl;
15 }
```

- Compiler erkennt dass Fkt `square` einmal für Typ `int` und einmal für Typ `double` benötigt wird
- Compiler erzeugt (“programmiert”) und kompiliert anhand von dieser Information, zwei(!) Funktionen mit der Signatur
  - `double square(double)`
  - `int square(int)`
- d.h. `square` automatisch durch Template generiert
  - also nur für die Typen, die wirklich benötigt

Klassentemplate 1/3

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4 using std::string;
 5
 6 template <typename Type>
 7 class Pointer {
 8 private:
 9   Type* ptr;
10   //Die Klasse soll nicht kopierbar sein.
11   Pointer(const Pointer&);
12   Pointer& operator=(const Pointer&);
13 public:
14   Pointer(Type* ptr);
15   ~Pointer();
16   Type& operator*();
17   Type* operator->();
18 };
```

- kann auch Templates für Klassen machen
- z.B. automatische Speicherverwaltung bei Pointern, sog. *smart pointer*
- Idee: Speicher automatisch freigeben
  - verhindert Speicherlecks
  - sog. *Garbage Collection*
- def. Klasse `Pointer<Type>` für beliebigen Typ `Type`
- Um zu verhindern, dass Objekt der Klasse kopiert wird, schreibt man Kopierkonstruktor und Zuweisungsoperator in `private` Bereich (Zeile 11,12)
  - `Pointer pointer(ptr);` ruft Konstruktor
  - `Pointer pointer = ptr;` liefert Syntaxfehler

Klassentemplate 2/3

```
20 template <typename Type>
21 Pointer<Type>::Pointer(Type* ptr) {
22   this->ptr = ptr;
23   cout << "Konstruktor" << endl;
24 }
25
26 template <typename Type>
27 Pointer<Type>::~Pointer() {
28   if ( ptr != (Type*) 0 ) {
29     delete ptr;
30   }
31   cout << "Destruktor" << endl;
32 }
33
34 template <typename Type>
35 Type& Pointer<Type>::operator*() {
36   return *ptr;
37 }
38
39 template <typename Type>
40 Type* Pointer<Type>::operator->() {
41   return ptr;
42 }
```

- Methoden der Klasse `Pointer<Type>`
  - voranstellen von `template <typename Type>`
- Implementierung wie gehabt
- Wichtig: dyn. Objekt wurde mit `new Type` erzeugt
  - sonst scheitert `delete` in Zeile 29
- Dereferenzieren (Z. 34-37) und Pfeil (Z. 39-42) werden auf gespeicherten Pointer weitergereicht
  - d.h. `*object` liefert `*(object.ptr)`
  - d.h. `object->` liefert `object.ptr->`

Klassentemplate 3/3

```
20 template <typename Type>
21 Pointer<Type>::Pointer(Type* ptr) {
22   this->ptr = ptr;
23   cout << "Konstruktor" << endl;
24 }
25
26 template <typename Type>
27 Pointer<Type>::~Pointer() {
28   if ( ptr != (Type*) 0 ) {
29     delete ptr;
30   }
31   cout << "Destruktor" << endl;
32 }
33
34 template <typename Type>
35 Type& Pointer<Type>::operator*() {
36   return *ptr;
37 }
38
39 template <typename Type>
40 Type* Pointer<Type>::operator->() {
41   return ptr;
42 }
43
44 int main() {
45   Pointer<string> pointer(new string("Hallo"));
46   cout << *pointer << endl;
47   cout << "Laenge = " << pointer->length() << endl;
48 }
```

- Output

  ```
  Konstruktor
  Hallo
  Laenge = 5
  Destruktor
  ```

- Destruktor gibt dynamischen Speicher wieder frei

Warum Zuweisung `private`?

- Probleme bei Kopien von Smartpointern:

```
1 int main() {
2   Pointer<string> p(new string("blub"));
3   p->length();
4   {
5     Pointer<string> q = p;
6     q->length();
7   }
8   p->length();
9 }
```

- Pointer wird kopiert (Zeile 5)
  - Hier: nicht möglich, da Zuweisung `private`
- Speicher von `q` wird freigegeben (Zeile 7)
  - Problem: also Speicher von `p` freigegeben
  - Zugriffsfehler in Zeile 8
- **Mögliche Lösung**: Kopien zählen
  - Speicher nur freigeben wenn kein Zugriff mehr
  - wird hier nicht vertieft

C++ Standardcontainer

- C++ hat viele vordefinierte Klassen-Templates
  - `list` (verkettete Listen)
  - `queue` (first-in-first-out)
  - `stack` (last-in-first-out)
  - `deque` (*double ended queue*)
  - `set`
  - `multiset`
  - `map`
  - `multimap`
  - `vector`
- Weitere C++ Bibliotheken
  - Boost Library: Große Sammlung an Bib.
  - `http://www.boost.org`

vector Template

vector Template

```
 1 #include <iostream>
 2 #include <string>
 3 #include <vector>
 4 using std::vector;
 5 using std::string;
 6 using std::cout;
 7
 8 class Eintrag {
 9 public:
10   string name;
11 };
12
13 int main() {
14   vector<Eintrag> telbuch(2);
15   telbuch[0].name = "Peter Pan";
16   telbuch[1].name = "Wolverine";
17   cout << telbuch[1].name << "\n";
18   return 0;
19 }
```

- vector ist C++ Standardcontainer
  - kann beliebige Datentypen verwenden
  - dienen zum Verwalten von Datenmengen
- Zeile 12: Anlegen eines Vektors der Länge 2 mit Einträgen vom Typ Eintrag
- Anlegen `vector<type> name(size);`
- Achtung, nicht verwechseln:
  - 1000 Einträge: `vector<Eintrag> buch(1000);`
  - 1000 Vektoren: `vector<Eintrag> buch[1000];`
- Zugriff auf j-tes Element wie bei Arrays
  - `telbuch[j]` (Zeile 13–14)

Vektoren mittels vector 1/3

```
 1 #include <iostream>
 2 #include <vector>
 3 #include <cassert>
 4
 5 using std::vector;
 6 using std::cout;
 7 using std::ostream;
 8
 9 class Vector {
10 private:
11   vector<double> coeff;
12
13 public:
14   Vector(int dim=0, double init=0);
15   int size() const;
16   const double& operator()(int k) const;
17   double& operator()(int k);
18   double norm() const;
19 };
```

- kein dynamischer Speicher, d.h. automatisch OK:
  - Kopierkonstruktur
  - Zuweisung
  - Destruktor

Vektoren mittels vector 2/3

```
21 Vector::Vector(int dim, double init) : coeff(dim,init) {}
22
23 int Vector::size() const {
24   return coeff.size();
25 }
26
27 const double& Vector::operator()(int k) const {
28   assert(k>=0 && k<size());
29   return coeff[k];
30 }
31
32 double& Vector::operator()(int k) {
33   assert(k>=0 && k<size());
34   return coeff[k];
35 }
36
37 ostream& operator<<(ostream& output, const Vector& x) {
38   output << "\n";
39   if (x.size()==0) {
40     output << " empty vector";
41   }
42   else {
43     for (int j=0; j<x.size(); ++j) {
44       output << " " << x(j);
45     }
46   }
47   output << "\n";
48   return output;
49 }
```

- vector Template hat Methode size (Zeile 24)
- wird genutzt für Methode size für Klasse Vector

Vektoren mittels vector 3/3

```
37 ostream& operator<<(ostream& output, const Vector& x) {
38   output << "\n";
39   if (x.size()==0) {
40     output << " empty vector";
41   }
42   else {
43     for (int j=0; j<x.size(); ++j) {
44       output << " " << x(j);
45     }
46   }
47   output << "\n";
48   return output;
49 }
50
51 int main() {
52   Vector x(5,2);
53   Vector y;
54   cout << "x = " << x;
55   cout << "y = " << y;
56   y = x;
57   cout << "y = " << y;
58   return 0;
59 }
```

- Zuweisung funktioniert (implizit generiert!)
- Output:

```
x =
 2 2 2 2 2
y =
 empty vector
y =
 2 2 2 2 2
```

Matrizen mittels vector 1/2

```
 1 #include <iostream>
 2 #include <vector>
 3 #include <cassert>
 4
 5 using std::vector;
 6 using std::cout;
 7 using std::ostream;
 8
 9 class SquareMatrix {
10 private:
11   vector<vector<double> > coeff;
12
13 public:
14   SquareMatrix(int dim=0, double init=0);
15   int size() const;
16   const double& operator()(int j, int k) const;
17   double& operator()(int j, int k);
18   double norm() const;
19 };
20
21 SquareMatrix::SquareMatrix(int dim, double init) :
22 coeff(dim,vector<double>(dim,init)) {}
23
24 int SquareMatrix::size() const {
25   return coeff.size();
26 }
27
28 const double& SquareMatrix::operator()(int j,int k) const {
29   assert(j>=0 && j<size());
30   assert(k>=0 && k<size());
31   return coeff[j][k];
32 }
33
34 double& SquareMatrix::operator()(int j, int k) {
35   assert(j>=0 && j<size());
36   assert(k>=0 && k<size());
37   return coeff[j][k];
38 }
```

- Beachte `vector<vector<double> >` in Zeile 11
  - `>>` statt `> >` wäre Operator für Input-Stream

Matrizen mittels vector 2/2

```
40 ostream& operator<<(ostream& output,
41                       const SquareMatrix& A) {
42   output << "\n";
43   int n = A.size();
44   if (n == 0) {
45     output << " empty matrix";
46   }
47   else {
48     for (int j=0; j<n; ++j) {
49       for (int k=0; k<n; ++k) {
50         output << " " << A(j,k);
51       }
52       output << "\n";
53     }
54   }
55   output << "\n";
56   return output;
57 }
58
59 int main() {
60   SquareMatrix A(3,5);
61   SquareMatrix B;
62   cout << "B = " << B;
63   A(1,1) = 0;
64   B = A;
65   cout << "B = " << B;
66   return 0;
67 }
```

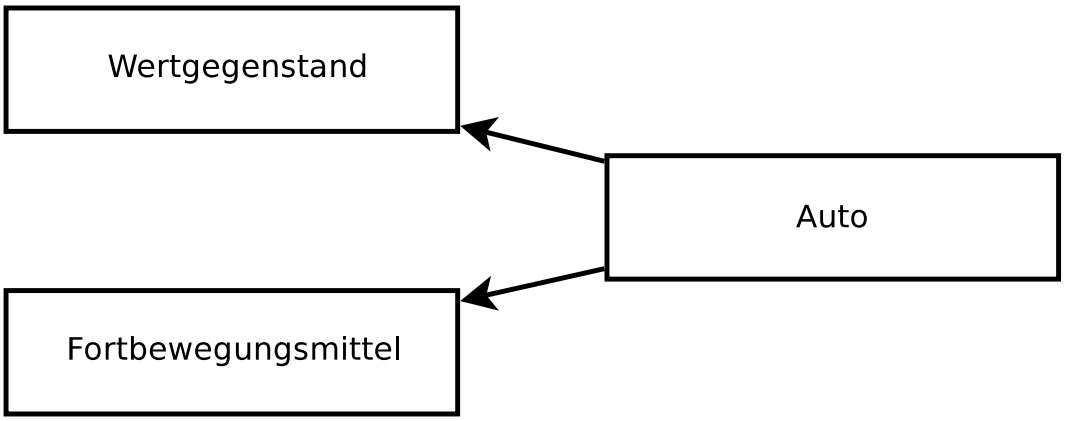
- Output:

```
B =
 empty matrix
B =
 5 5 5
 5 0 5
 5 5 5
```

Mehrfachvererbung

Mehrfachvererbung

- C++ erlaubt Vererbung mit multiplen Basisklassen



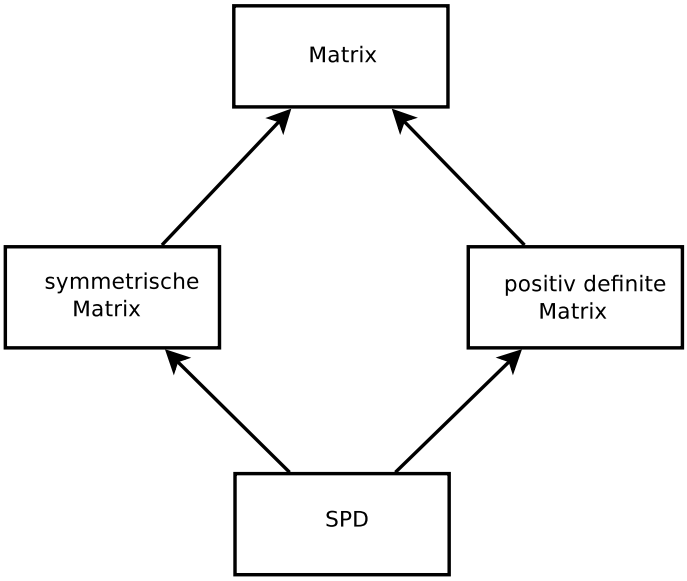
- Syntax:

```
class Auto : public Wertgegenstand, public Fortbew {...}
```

- Vertieft Konzept der Objektorientierung
  - erhöht Wiederverwendbarkeit von Code
- Problem: Mehrdeutigkeiten (nächste Folie)

Diamantvererbung 1/5

- Es könnte eine gemeinsame Oberklasse geben



- SPD = symmetrisch positiv definite Matrix
  - **symmetrisch:** $A = A^T \in \mathbb{R}^{n \times n}$
  - **positiv definit:** $Ax \cdot x > 0$ für alle $x \in \mathbb{R}^n \backslash \{0\}$
    - äquivalent: alle Eigenwerte sind strikt positiv

- Führt zu Mehrdeutigkeit
  - Felder und Methoden sind mehrfach vorhanden
  - Unklar worauf zugegriffen werden soll
  - Speicherverschwendung
  - Schlimmstenfalls: Objekte inkonsistent

Diamantvererbung 2/5

```
 5 class Matrix{
 6 private:
 7   int n;
 8 public:
 9   void set(int n) {this->n = n;}
10   int get() {return n;}
11 };
12
13 class SMatrix : public Matrix {};
14
15 class PDMatrix : public Matrix {};
16
17 class SPDMatrix : public SMatrix, public PDMatrix {};
```

- Klasse `Matrix` hat Member `int n`
- beide abgeleiteten Klassen erben `int n`
- `SPDMatrix` erbt von zwei Klassen
  - `SPDMatrix` hat `int n` doppelt

- **naive Lösung:** Zugriff mittels vollem Namen
  - `SMatrix::n` bzw. `PDMatrix::n`
- unschön, da Speicher dennoch doppelt
  - unübersichtlich
  - fehleranfällig

- **bessere Lösung:** virtuelle Vererbung
  - z.B. `class SMatrix : virtual public Matrix`
- virtuelle Basisklasse wird nur einmal eingebunden

Diamantvererbung 3/5

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 class Matrix{
 6 private:
 7   int n;
 8 public:
 9   void set(int n) {this->n = n;}
10   int get() {return n;}
11 };
12
13 class SMatrix : public Matrix {};
14
15 class PDMatrix : public Matrix {};
16
17 class SPDMatrix : public SMatrix, public PDMatrix {};
18
19 int main() {
20   SPDMatrix A;
21   A.set(1);
22   cout << "n = " << A.get() << endl;
23
24   return 0;
25 }
```

- Kompilieren liefert Fehler

```
diamant1.cpp:21: error: non-static member 'set' found
in multiple base-class subobjects of type 'Matrix':
class SPDMatrix -> class SMatrix -> class Matrix
class SPDMatrix -> class PDMatrix -> class Matrix
```

- alle Datenfelder und Methoden sind doppelt!
  - Zugriff über vollständigen Namen möglich
  - z.B. `SMatrix::set` schreibt `SMatrix::n`

Diamantvererbung 4/5

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 class Matrix{
 6 private:
 7   int n;
 8 public:
 9   void set(int n) {this->n = n;}
10   int get() {return n;}
11 };
12
13 class SMatrix : public Matrix {};
14
15 class PDMatrix : public Matrix {};
16
17 class SPDMatrix : public SMatrix, public PDMatrix {};
18
19 int main() {
20   SPDMatrix A;
21   A.SMatrix::set(1);
22   A.PDMatrix::set(2);
23   cout << "n = " << A.SMatrix::get() << endl;
24   cout << "n = " << A.PDMatrix::get() << endl;
25
26   return 0;
27 }
```

- `SMatrix::n` und `PDMatrix::n` können verschiedene Werte haben
  - fehleranfällig!
- Output:

  ```
  n = 1
  n = 2
  ```

Diamantvererbung 5/5

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::endl;
 4
 5 class Matrix{
 6 private:
 7   int n;
 8 public:
 9   void set(int n) {this->n = n;}
10   int get() {return n;}
11 };
12
13 class SMatrix : virtual public Matrix {};
14
15 class PDMatrix : virtual public Matrix {};
16
17 class SPDMatrix : public SMatrix, public PDMatrix {};
18
19 int main() {
20   SPDMatrix A;
21   A.set(1);
22   A.SMatrix::set(2);
23   A.PDMatrix::set(3);
24   cout << "n = " << A.get() << endl;
25   cout << "n = " << A.SMatrix::get() << endl;
26
27   return 0;
28 }
```

- Vererbung `virtual` der Basisklasse (Zeile 13 + 15)
  - Members werden nur 1x an abg. Klassen vererbt
- Syntaxfehler, falls nur eine der Klassen `virtual`!
- Output:

```
n = 3
n = 3
```

Exception Handling

- `try`
- `catch`
- `throw`
- `bad_alloc`

Exception Handling 1/2

- Was tun bei falscher Benutzereingabe?
  - Programm sofort beenden?
  - Benutzer informieren und Programm beenden?
  - eingreifen, ohne Benutzer zu informieren?
  - dem Benutzer helfen, den Fehler zu korrigieren?
- bisher: sofort beenden mittels `assert`
- Exceptions sind Ausnahmezustände
- Gründe für die Entstehung
  - falsche Eingabe durch Benutzer
  - kein Speicher mehr verfügbar
  - Dateizugriffsfehler
  - Division durch Null
  - etc.
- können diese „Fehler“ nicht verhindern
  - können sie aber antizipieren
  - an zentraler Stelle behandeln
- sofortiges Beenden ist i.a. keine Option!
  - Verlust eventuell korrekt berechneter Daten

Exception Handling 2/2

- Konzept des Exception Handling
  - Trennung von normalen Programmfluss und Behandlung von Fehlern
  - Fehler die in einem Teil auftauchen, werden zentral von aufrufender Funktion behandelt
  - Keine ständige Kontrolle ob Fehler aufgetreten

- Syntax in C++:
- `try {...}` schließt risikobehafteten Code ein
  - d.h. hier werden eventuell Exceptions geworfen
    - d.h. Fehler werden implizit / explizit erkannt
  - Sobald Exception geworfen wird, wird Code beim nächsten `catch` fortgesetzt
    - Prg terminiert, falls kein passendes `catch`
- `throw name;` wirft eine Exception
  - `name` ist Objekt vom Typ `type`
  - enthält Info über den aufgetretenen Fehler
- `catch(type name) {...}` fängt `type` Exception
  - reagiert auf Exception, sog. Exception Handler

Beispiel zu `throw`

```
1 #include <iostream>
2 using std::cout;
3
4 int main() {
5   cout << "*** throw\n";
6   throw int(1);
7   cout << "*** continue\n";
8   return 0;
9 }
```

- `throw` löst Exception aus
- da kein passendes `catch` folgt, wird Code beendet
  - Achtung: `catch` erfordert vorausgehendes `try`
- Output:

```
*** throw
terminating with uncaught exception of
type int
```

Beispiel zu try-catch 1/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 int main() {
 5   cout << "*** throw\n";
 6   try{
 7     throw int(1);
 8   }
 9   catch(double x) {
10     cout << "*** catch\n";
11   }
12   cout << "*** continue\n";
13   return 0;
14 }
```

- `throw` löst Exception aus (vom Type `int`)
- da kein passendes `catch` folgt, wird Code beendet
- kein impliziter Type Cast, sondern Type sensitiv!
  - Exception vom Type `int` (Zeile 7)
  - `catch` aber nur für Type `double` (Zeile 9)
- Output:

```
*** throw
terminating with uncaught exception of
type int
```

Beispiel zu try-catch 2/2

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 int main() {
 5   cout << "*** throw\n";
 6   try{
 7     throw int(1);
 8   }
 9   catch(int x) {
10     cout << "*** catch, x = " << x << "\n";
11   }
12   cout << "*** continue\n";
13   return 0;
14 }
```

- `throw` löst Exception aus (vom Type `int`)
- passendes `catch` fängt Exception (Zeile 9)
- Code wird danach normal fortgesetzt

- Output:

```
*** throw
*** catch, x = 1
*** continue
```

Eigene Klassen für Exceptions

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Error {
 5 private:
 6   int code;
 7 public:
 8   Error(int x) {
 9     code = x;
10   }
11   void print() {
12     cout << "error code: " << code << "\n";
13   }
14 };
15
16 int main() {
17   cout << "*** throw\n";
18   try{
19     throw Error(1);
20   }
21   catch(Error info) {
22     info.print();
23   }
24   cout << "*** continue\n";
25   return 0;
26 }
```

- Kann beliebige Objekte als Exception werfen
  - erlaubt systematische Gliederung / Behandlung

- Output:

```
*** throw
error code: 1
*** continue
```

Abgeleitete Klassen 1/4

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::string;
 4
 5 class Error {
 6 private:
 7   int code;
 8 public:
 9   Error(int x) {
10     code = x;
11   }
12   void print() {
13     cout << "error code: " << code << "\n";
14   }
15   int getCode() {
16     return code;
17   }
18 };
19
20 class NewError : public Error {
21 private:
22   string message;
23 public:
24   NewError(int x, string txt) : Error(x) {
25     message = txt;
26   }
27   void print() {
28     cout << message << " (code: " << getCode() << ")\n";
29   }
30 };
```

- Kann beliebige Objekte als Exception werfen
  - zwei Klassen / Typen für Exceptions:
  - `Error` (Zeile 5–18), `NewError` (Zeile 20–30)

Abgeleitete Klassen 2/4

```
32 int main() {
33   cout << "*** throw\n";
34   try{
35     throw NewError(1,"exception");
36   }
37   catch(Error info) {
38     info.print();
39   }
40   cout << "*** continue\n";
41   return 0;
42 }
```

- jedes Objekt der abgeleiteten Klasse ist auch vom Typ der Basisklasse (Polymorphie)
- `catch(Error)` fängt Objekte vom Typ `Error` und Typ `NewError`
  - d.h. `Error::print()` in Zeile 38
- Output:

```
*** throw
error code: 1
*** continue
```

Abgeleitete Klassen 3/4

```
32 int main() {
33   cout << "*** throw\n";
34   try{
35     throw NewError(1,"exception");
36   }
37   catch(NewError info) {
38     info.print();
39   }
40   catch(Error info) {
41     info.print();
42   }
43   cout << "*** continue\n";
44   return 0;
45 }
```

- einem `try` können beliebig viele `catch` folgen
- erstes passendes `catch` fängt Exception
- alle anderen `catch` werden übergangen
- `catch(NewError)` fängt Objekte vom Typ `NewError`
- `catch(Error)` fängt alle übrigen vom Typ `Error`
- Output:

```
*** throw
exception (code: 1)
*** continue
```

Abgeleitete Klassen 4/4

```
32 int main() {
33   cout << "*** throw\n";
34   try{
35     throw int(1);
36   }
37   catch(NewError info) {
38     info.print();
39   }
40   catch(Error info) {
41     info.print();
42   }
43   catch(...) {
44     cout << "some unknown error occured\n";
45   }
46   cout << "*** continue\n";
47   return 0;
48 }
```

- einem `try` können beliebig viele `catch` folgen
- `catch(...)` fängt alle verbliebenen Exceptions
- Output:

```
*** throw
some unknown error occured
*** continue
```

Exception bei Speicherallokation

```
 1 #include <iostream>
 2 using std::cout;
 3 using std::bad_alloc;
 4
 5 int main() {
 6   double* ptr = (double*) 0;
 7   try {
 8     cout << "*** allocate memory\n";
 9     while(1) {
10       ptr = new double[1024*1024*1024];
11     }
12   }
13   catch (bad_alloc) {
14     cout << "*** out of memory\n";
15   }
16   cout << "*** continue\n";
17   return 0;
18 }
```

- iterierte Allokation von jeweils 1 GB (Zeile 9–11)
- gescheitertes `new` wirft Exception `bad_alloc`
- Output:

```
*** allocate memory
*** out of memory
*** continue
```