Einführung in das Programmieren für Technische Mathematik

DI Thomas Führer

Marcus Page, MSc

Prof. Dr. Dirk Praetorius

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

Institut für Analysis
und Scientific Computing

Formalia

- Rechte & Pflichten
- Benotung
- Anwesenheitspflicht
- Literatur

EPROG-Homepage

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

Literatur

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

„freiwillige“ Literatur

- Brian Kernighan, Dennis Ritchie
  Programmieren in C
- Klaus Schmaranz
  Softwareentwicklung in C
- Ralf Kirsch, Uwe Schmitt
  Programmieren in C, eine mathematikorientierte Einführung
- Bjarne Stroustrup
  Die C++ Programmiersprache
- Klaus Schmaranz
  Softwareentwicklung in C++
- Dirk Louis
  Jetzt lerne ich C++
- Jesse Liberty
  C++ in 21 Tagen

Das erste C-Programm

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

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

Programm

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

Algorithmus

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

Source-Code

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

Programmiersprachen

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

Achtung

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

Wie erstellt man ein C-Programm?

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

Das erste C-Programm

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

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

Syntaxfehler

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

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

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

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

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

- Fehlt Semikolon am Zeilenende 4
  - Compilieren liefert Fehlermeldung:

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

Laufzeitfehler

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

Variablen

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

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

Variable

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

Datentypen

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

Deklaration

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

Initialisierung

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

Ein erstes Beispiel zu `int`

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

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

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

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

Dasselbe Beispiel zu `double`

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

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

Zuweisungsoperator

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

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

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


Arithmetische Operatoren

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

Type Casting

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

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

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

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


Implizites Type Casting

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

- Output:

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

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

Explizites Type Casting

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

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

  a) 0.000000

  b) 0.000000

  c) 0.666667

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

Fehlerquelle beim Type Casting

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

- Output:

  a) 0.666667

  b) 2.000000

  c) 0

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

Einfache Verzweigung

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

Logische Operatoren

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

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

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

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

Logische Verkettung

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

- Output:

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

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

if-else

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

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

Beispiel zu if

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

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

Beispiel zu if-else

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

- Output:

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

Gerade oder Ungerade?

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

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

Zwei Zahlen aufsteigend sortieren

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

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

Innen oder Außen?

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


Gleichheit vs. Zuweisung

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

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

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` definiert, 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!

Bsp. zu lokalen & globalen Var.

```
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
```

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

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 ⇒ 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!
    - ⇒ 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  }
16
```

- 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  #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 m = 0.5*(a+b);
10    if ( b - a <= 2*tol ) {
11      return m;
12    }
13    else {
14      if ( f(a)*f(m) <= 0 ) {
15        return bisection(a,m,tol);
16      }
17      else {
18        return bisection(m,b,tol);
19      }
20    }
21  }
22
23  main() {
24    double a = 0;
25    double b = 10;
26    double tol = 1e-12;
27    double x = bisection(a,b,tol);
28
29    printf("Nullstelle x=%g\n",x);
30    printf("Funktionswert f(x)=%g\n",f(x));
31  }
```

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
      - geeignet, um Konstanten zu definieren
    - `#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 -ℓm` 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 $x^y = \mathtt{pow(x,y)}$)
  - Absolutbetrag `fabs`
  - Rundung auf ganze Zahlen: `floor`, `ceil`

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` ⇒ 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
12  main() {
13    double x[3] = {0,0,0};
14
15    printf("c) x = (%f, %f, %f)\n",x[0],x[1],x[2]);
16    callByReference(x);
17    printf("d) x = (%f, %f, %f)\n",x[0],x[1],x[2]);
18  }
```

- 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`
  - Namen von Konstanten in Großbuchstaben
- 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 $\Rightarrow$ i.a. keine Fehlermeldung
- Für `WRONG` groß genug $\Rightarrow$ Laufzeitfehler

Matrizen

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

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

  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 Zeile
  - oder mehrere Zeilen in geschwungenen 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!

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
17    for (j=1; j<dim; j=j+1) {
18      if (input[j]<minval) {
19        minval = input[j];
20      }
21    }
22    return minval;
23  }
24
25  main() {
26    double x[DIM];
27    scanvector(x,DIM);
28    printf("Minimum des Vektors ist %f\n",
29           min(x,DIM));
30  }
```


Beispiel: Summensymbol $\sum$

- Berechnung der Summe $S = \sum_{j=1}^{N} a_j$:
  - Abkürzung $\sum_{j=1}^{N} a_j := a_1 + a_2 + \ldots + a_N$
- 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
    - $S = a_1$
    - $S = S + a_2$
    - $S = S + a_3$ 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!

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

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

- 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 $a \in \mathbb{R}^{MN}$
  - $a = (A_{00}, A_{10}, \ldots, A_{M-1,0}, A_{01}, A_{11}, \ldots, A_{M-1,N-1})$
  - A_{jk} entspricht also a_ℓ mit $\ell = j + k \cdot M$
- 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    return max;
12  }
```

- Aufwand:
  - 1 Zuweisung ⇝ Zeile 5
  - $n-1$ Schritte mit jeweils ⇝ Zeile 6–10
    - 1 Vergleich ⇝ Zeile 7
    - 1 Zuweisung (worst case!) ⇝ Zeile 8
- insgesamt $1 + 2(n-1) = 2n-1$ Operationen

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  }
```

- `m` Schritte ⇝ Zeile 6–11
  - 1 Zuweisung ⇝ Zeile 7
  - jeweils `n` mal ⇝ Zeile 8–10
    - 1 Multiplikation ⇝ Zeile 9
    - 1 Addition ⇝ Zeile 9
    - 1 Zuweisung ⇝ Zeile 9
- Insgesamt $3mn + m$ Operationen
- Aufwand $\mathcal{O}(mn)$
  - bzw. Aufwand $\mathcal{O}(n^2)$ für $m = n$
  - d.h. quadratischer Aufwand für $m = n$

Suchen im Vektor

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

- Aufgabe:
  - Suche Index `j` mit `vector[j]` = `value`
  - Rückgabe `-1`, falls nicht ex.
- Achtung bei Gleichheit mit `double` (später!)
- `n` Schritte
  - 1 Vergleich
- Insgesamt n Operationen
- 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      printf("%d %d j=%d\n",start,end,j);
19    }
20
21    return -1;
22  }
```

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

Minsort

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

- n-1 Schritte
  - 1 Zuweisung
  - jeweils $n-(j+1)=n-j-1$ mal
    - 1 Vergleich
    - 1 Zuweisung (worst case!)
  - jeweils 1 Vergleich
  - jeweils 3 Zuweisungen (worst case!)

- Insgesamt $5(n-1)+\sum_{j=0}^{n-2}(n-j-1)2$
  $= 5(n-1)+2\sum_{k=1}^{n-1}k$
  $= 5(n-1)+2\frac{n(n-1)}{2}$
  $= n^2+4n-5$, d.h. quadratischer Aufwand $\mathcal{O}(n^2)$

Zeitmessung

- Wozu Zeitmesssung?
  - Vergleich von Algorithmen
  - Vergleich von Implementierungen
  - Überprüfen theoretischer Voraussagen

- theoretische Voraussagen
  - linearer Aufwand
    - Problemgröße $n \Rightarrow Cn$ Operationen
    - Problemgröße $kn \Rightarrow Ckn$ Operationen
    - d.h. 3× Problemgröße ⇒ 3× Rechenzeit
  - quadratischer Aufwand
    - Problemgröße $n \Rightarrow Cn^2$ Operationen
    - Problemgröße $kn \Rightarrow Ck^2n^2$ Operationen
    - d.h. 3× Problemgröße ⇒ 9× Rechenzeit
  - etc.

- 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,t2;
13    int i = 0;
14    int v[DIM];
15
16    for(i=0; i<DIM; i=i+1) {
17      printf("v[%d]=",i);
18      scanf("%d",&v[i]);
19    }
20    t1 = clock();
21    i = search(v,VAL,DIM);
22    t2 = clock();
23    printf("search: %f\n",
24           (double)(t2-t1)/CLOCKS_PER_SEC);
25    t1 = clock();
26    minsort(v,DIM);
27    t2 = clock();
28    printf("minsort: %f\n",
29           (double)(t2-t1)/CLOCKS_PER_SEC);
30    t1 = clock();
31    i = binsearch(v,VAL,DIM);
32    t2 = clock();
33    printf("binary search: %f\n",
34           (double)(t2-t1)/CLOCKS_PER_SEC);
35  }
```


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	0.02	0.00
8.000	0.00	0.07	0.00
16.000	0.00	0.29	0.00
32.000	0.00	1.11	0.00
64.000	0.00	4.45	0.00
128.000	0.00	17.81	0.00
256.000	0.00	71.20	0.00
512.000	0.00	284.38	0.00
1.024.000	0.00	$\geq$ 18min	0.00
2.048.000	0.00	$\geq$ 72min	0.00
4.096.000	0.01	$\geq$ 4,5h	0.00
8.192.000	0.02	$\geq$ 18h	0.00
16.384.000	0.04	$\geq$ 3d	0.00
32.768.000	0.07	$\geq$ 12d	0.00
65.536.000	0.15	$\geq$ 1,5m	0.00
131.072.000	0.31	$\geq$ 6m	0.00
262.144.000	0.60	$\geq$ 2y	0.00
524.288.000	1.21	$\geq$ 8y	0.00
1.048.576.000	2.42	$\geq$ 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 `--a`
    - Erst verringern, dann auswerten
  - Postdekrement `a--`
    - 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  }
```


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 $a \neq b$

Euklids 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 != 0) {
11      if ( a < b) {
12        tmp = a;
13        a = b;
14        b = tmp;
15      }
16      a = a-b;
17    }
18
19    printf("%d\n",b);
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 $ggT(a, b) = b$ und $a - b = 0$
- Output:

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

Euklids Algorithmus (verbessert)

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int a = 200;
 5    int b = 2110;
 6    int tmp = 0;
 7
 8    printf("ggT(%d,%d)=",a,b);
 9
10    while (b != 0) {
11      tmp = b;
12      b = a%b;
13      a = tmp;
14    }
15
16    printf("%d\n",a);
17  }
```

- Euklids Algorithmus berechnet ggT von $a, b \in \mathbb{N}$
- basiert auf $ggT(a,b) = ggT(a-b,b)$ für $a > b$
- Für $a < b$ vertausche a und b
  - Divsionsrest von a/b ist $a\%b = a$
- Für $a > b$ iteriere $a := a - b$ bis $a < b$
  - d.h. bis $a = a\%b$
  - dann vertausche a und b
- Für $a = b$ gilt $ggT(a,b) = a$ und $a\%b = 0$

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

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  }
```


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  }
```

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

- Funktion `minmax` 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!

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 printf_sizeof(double vector[]) {
4    printf("sizeof(vector) = %d\n",sizeof(vector));
5  }
6
7  main() {
8    int var = 43;
9    double array[11];
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   printf_sizeof(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
- Output:

```
sizeof(var) = 4
sizeof(double) = 8
sizeof(array) = 88
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
3  double* scanfvector(int length) {
4    double vector[length];
5    int j = 0;
6    for (j=0; j<length; ++j) {
7      vector[j] = 0;
8      printf("vector[%d] = ",j);
9      scanf("%lf",&vector[j]);
10   }
11   return vector;
12 }
13
14 main() {
15   double* x;
16   int j = 0;
17   int dim = 0;
18
19   printf("dim = ");
20   scanf("%d",&dim);
21
22   x = scanfvector(dim);
23
24   for (j=0; j<dim; ++j) {
25     printf("x[%d] = %f\n",j,x[j]);
26   }
27 }
```

- Syntax des Programms ist OK
- Problem: Speicher zu `x` mit Blockende 12 aufgelöst
  - d.h. Pointer aus 11 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!

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!

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

Beispiel

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


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

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

- 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!
- 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 $0 \leq j \leq N-1$

Eine erste Bibliothek

- 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

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

Source-Code (Ausschnitt)

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


Dynamische Matrizen

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

Statische Matrizen

- Pointer sind Datentypen $\Rightarrow \exists$ Pointer auf Pointer
- `double array[M][N];` deklariert statische Matrix `array` der Dimension $M \times N$ mit `double`-Koeffizienten
  - Indizierung mittels `array[j][k]` mit $0 \leq j \leq M-1$ und $0 \leq k \leq N-1$
  - Dimensionen `M`, `N` können während Programmablauf nicht verändert werden
  - Funktionen können `M`, `N` nicht herausfinden, d.h. stets als Input-Parameter übergeben
- Formal: Zeile `array[j]` ist Vektor der Länge `N` mit Koeffizienten vom Typ `double`
  - also `array[j]` intern vom Typ `double*`
- Formal: `array` Vektor der Länge `M` mit Koeffizienten vom Typ `double*`
  - also `array` intern vom Typ `double**`

Dynamische Matrizen

- statische Matrix `double array[M][N];`
  - `array` ist `double**` [`double*`-Vektor der Länge `M`]
  - `array[j]` ist `double*` [`double`-Vektor der Länge `N`]
- Allokation der dyn. Matrix entlang dieser Vorgaben

```
 1  double** mallocMatrix(int m, int n) {
 2    int j = 0;
 3    int k = 0;
 4    double** matrix = malloc(m*sizeof(double*));
 5    for (j=0; j<m; ++j) {
 6      matrix[j] = malloc(n*sizeof(double));
 7      for (k=0; k<n; ++k) {
 8        matrix[j][k] = 0;
 9      }
10    }
11    return matrix;
12  }
```

- Beachte Typen innerhalb von `sizeof` in 4 und 6!
- Mit Hilfe der Bibliothek für dyn. Vektoren gilt

```
1  double** mallocMatrix(int m, int n) {
2    int j = 0;
3    double** matrix = malloc(m*sizeof(double*));
4    for (j=0; j<m; ++j) {
5      matrix[j] = mallocvector(n);
6    }
7    return matrix;
8  }
```

Freigeben dynamischer Matrizen

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

```
1  double** freematrix(double** matrix, int m) {
2    int j = 0;
3    for (j=0; j<m; ++j) {
4      free(matrix[j]);
5    }
6    free(matrix);
7    return NULL;
8  }
```

- An dieser Stelle kein Gewinn, in 4 Bibliothek für dynamische Vektoren zu verwenden

Re-Allokation 1/3

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

```
1   for (j=mnew; j<m; ++j) {
2     free(matrix[j]);
3   }
4   matrix = realloc(matrix,mnew*sizeof(double*));
5   for (j=0; j<mnew; ++j) {
6     matrix[j] = realloc(matrix[j],
7                         nnew*sizeof(double));
8     for (k=n; k<nnew; ++k) {
9       matrix[j][k] = 0;
10    }
11  }
```

- Realisierung mittels Bibliothek für dyn. Vektoren:

```
1   for (j=mnew; j<m; ++j) {
2     free(matrix[j]);
3   }
4   matrix = realloc(matrix,mnew*sizeof(double*));
5   for (j=0; j<mnew; ++j) {
6     matrix[j] = reallocvector(matrix[j],n,nnew);
7   }
```


Re-Allokation 2/3

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

```
1   for (j=0; j<m; ++j) {
2     matrix[j] = realloc(matrix[j],
3                         nnew*sizeof(double));
4     for (k=n; k<nnew; ++k) {
5       matrix[j][k] = 0;
6     }
7   }
8   matrix = realloc(matrix,mnew*sizeof(double*));
9   for (j=m; j<mnew; ++j) {
10    matrix[j] = malloc(nnew*sizeof(double));
11    for (k=0; k<nnew; ++k) {
12      matrix[j][k] = 0;
13    }
14  }
```

- Realisierung mittels Bibliothek für dyn. Vektoren:

```
1   for (j=0; j<m; ++j) {
2     matrix[j] = reallocvector(matrix[j],n,nnew);
3   }
4   matrix = realloc(matrix,mnew*sizeof(double*));
5   for (j=m; j<mnew; ++j) {
6     matrix[j] = mallocvector(nnew);
7   }
```


Re-Allokation 3/3

```
1   double** reallocmatrix(double** matrix,
2                          int m, int n,
3                          int mnew, int nnew) {
4
5    int j = 0;
6
7    if (mnew<m) {
8      for (j=mnew; j<m; ++j) {
9        free(matrix[j]);
10     }
11     matrix = realloc(matrix,mnew*sizeof(double*));
12     for (j=0; j<mnew; ++j) {
13       matrix[j] = reallocvector(matrix[j],n,nnew);
14     }
15   }
16   else {
17     for (j=0; j<m; ++j) {
18       matrix[j] = reallocvector(matrix[j],n,nnew);
19     }
20     matrix = realloc(matrix,mnew*sizeof(double*));
21     for (j=m; j<mnew; ++j) {
22       matrix[j] = mallocvector(nnew);
23     }
24   }
25   return matrix;
26 }
```

Bemerkungen

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

Strings

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

Strings (= Zeichenketten)

- Strings = `char`-Arrays, also 2 Definitionen möglich
  - statisch: `char array[N];`
    - `N` = statische Länge
    - Deklaration & Initialisierung möglich `char array[] = "text";`
  - dynamisch (wie oben, Typ: `char*`)
- Fixe Strings in Anführungszeichen `"..."`
- Zugriff auf einzelnes Zeichen mittels `'...'`
- Zugriff auf Teil-Strings nicht möglich!
- Achtung bei dynamischen Strings:
  - als Standard enden alle Strings mit Null-Byte `\0`
    - Länge eines Strings dadurch bestimmen!
  - Bei statischen Arrays geschieht das automatisch (also wirkliche Länge `N+1` und `array[N]='\0'`)
    - Bei dyn. Strings also 1 Byte mehr reservieren!
    - und `\0` nicht vergessen
- An Funktionen können auch fixe Strings (in Anführungszeichen) übergeben werden
  - z.B. `printf("Hello World!\n");`

Funktionen zur String-Manipulation

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

Beispiel

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

- Output:

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

- Fixe Strings in Anführungszeichen `"..."` (Z. 13)
  - erzeugt statisches Array mit zusätzlichem Null-Byte am Ende
- Zugriff auf einzelne Zeichen eines Strings mit einfachen Hochkommata `'...'` (Zeile 15)
- Platzhalter für Strings in `printf` ist `%s` (Zeile 16)

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 in C ist Modulo-Arithmetik
  - d.h. Zahlenbereich ist geschlossen
    - `intmax` + 1 liefert `intmin`
    - `intmin` − 1 liefert `intmax`

```
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 $\mathbb{F}(2, M, e_{\min}, e_{\max})$
- schlecht gestellte Probleme
- Rechenfehler und Gleichheit

- `float`, `double`

Definition

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

  sodass gilt $x = \sigma\Big(\sum_{k=1}^{\infty} a_k 2^{-k}\Big)2^e$
- 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 $x = \sigma\Big(\sum_{k=1}^{M} a_k 2^{-k}\Big)2^e$ 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 $e_{\min} \le e \le e_{\max}$
- 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. $x \ge 0$ — Multipliziere ggf. mit $\sigma = -1$.
- Sei $e \in \mathbb{N}_0$ mit $0 \le x < 2^e$
- 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. $x_1 = 1/2$
    - $x_1 = a_1/2 \le x$
    - $x < (a_1 + 1)/2 = x_1 + 2^{-1}$
- 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 $a_{n+1} = 1$
    - $x_{n+1} = x_n + a_{n+1}2^{-(n+1)} \le x$
    - $x < x_n + (a_{n+1}+1)2^{-(n+1)} = x_{n+1} + 2^{-(n+1)}$
- Es folgt $|x_n - x| \le 2^{-n}$, also $x = \sum_{k=1}^{\infty} a_k 2^{-k}$

Anmerkungen zum Satz

- Satz gilt für jede Basis $b \in \mathbb{N}_{\ge 2}$
  - Ziffern dann $a_j \in \{0, 1, \ldots, b-1\}$
- Dezimalsystem $b = 10$ ist übliches System
  - $47.11 = (4{\cdot}10^{-1}{+}7{\cdot}10^{-2}{+}1{\cdot}10^{-3}{+}1{\cdot}10^{-4}){*}10^2$
    - $a_1 = 4$, $a_2 = 7$, $a_3 = 1$, $a_4 = 1$, $e = 2$
- Mit $b = 2$ sind Brüche genau dann als endliche Summe darstellbar, wenn Nenner Zweierpotenz

Arithmetik für Gleitkommazahlen

- Ergebnis `Inf` bei Überlauf
- Ergebnis `-Inf` bei Unterlauf
- 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. $|x - y| \le \varepsilon \cdot \max\{|x|, |y|\}$ 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`

- Gleitkommazahlen sind endliche Teilmenge von $\mathbb{Q}$
- `float` ist idR. einfache Genauigkeit nach IEEE-754-Standard
  - $\mathbb{F}(2, 24, -125, 128) \rightarrow$ 4 Byte
  - sog. *single precision*
  - ca. 7 signifikante Dezimalstellen
- `double` ist idR. doppelte Genauigkeit nach IEEE-754-Standard
  - $\mathbb{F}(2, 53, -1021, 1024) \rightarrow$ 8 Byte
  - sog. *double precision*
  - ca. 16 signifikante Dezimalstellen
- Platzhalter in `printf` und `scanf`

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

Strukturen

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

Deklaration von Strukturen

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

```
 1  // Declaration of structure
 2  struct _Student_ {
 3    char* firstname; // Vorname
 4    char* lastname;  // Nachname
 5    int studentID;   // Matrikelnummer
 6    int studiesID;   // Studienkennzahl
 7    int test1;       // Noten der Tests
 8    int test2;
 9    int uebung;      // Note der Uebung
10  };
11
12  // Declaration of corresponding data type
13  typedef struct _Student_ Student;
```

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

Strukturen & Members

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

```
 1  // Declaration of structure
 2  struct _Student_ {
 3    char* firstname; // Vorname
 4    char* lastname;  // Nachname
 5    int studentID;   // Matrikelnummer
 6    int studiesID;   // Studienkennzahl
 7    int test1;       // Noten der Tests
 8    int test2;
 9    int uebung;      // Note 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.test1 = 3;
22    var.test2 = 4;
23    var.uebung = 5;
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 unterste 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
4     (*pointer).firstname = NULL;
5     (*pointer).lastname = NULL;
6     (*pointer).studentID = 0;
7     (*pointer).studiesID = 0;
8     (*pointer).test1 = 0;
9     (*pointer).test2 = 0;
10    (*pointer).uebung = 0;
11
12    return pointer;
13  }
```


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

```
1   Student* newStudent() {
2     Student* student = malloc(sizeof(Student));
3
4     student->firstname = NULL;
5     student->lastname = NULL;
6     student->studentID = 0;
7     student->studiesID = 0;
8     student->test1 = 0;
9     student->test2 = 0;
10    student->uebung = 0;
11
12    return student;
13  }
```


Strukturen: Speicher freigeben

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

```
1   Student* delStudent(Student* student) {
2     if (student != NULL) {
3       if (student->firstname != NULL) {
4         free(student->firstname);
5       }
6
7       if (student->lastname != NULL) {
8         free(student->lastname);
9       }
10
11      free(student);
12    }
13    return NULL;
14  }
```

Shallow Copy

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

```
1   Student* copyStudent(Student* student) {
2     Student* copy = newStudent();
3
4     // ACHTUNG: Pointer!
5     copy->firstname = student->firstname;
6     copy->lastname = student->lastname;
7
8     // Kopieren der harmlosen Daten
9     copy->studentID = student->studentID;
10    copy->studiesID = student->studiesID;
11    copy->test1 = student->test1;
12    copy->test2 = student->test2;
13    copy->uebung = student->uebung
14
15    return copy;
16  }
```


Deep Copy

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

```
1   Student* cloneStudent(Student* student) {
2     Student* copy = newStudent();
3     int length = 0;
4
5     if (student->firstname != NULL) {
6       length = strlen(student->firstname)+1;
7       copy->firstname = malloc(length*sizeof(char));
8       strcpy(copy->firstname, student->firstname);
9     }
10
11
12    if (student->lastname != NULL) {
13      length = strlen(student->lastname)+1;
14      copy->lastname = malloc(length*sizeof(char));
15      strcpy(copy->lastname, student->lastname);
16    }
17
18    copy->studentID = student->studentID;
19    copy->studiesID = student->studiesID;
20    copy->test1 = student->test1;
21    copy->test2 = student->test2;
22    copy->uebung = student->uebung;
23
24    return copy;
25  }
```


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();
```

- 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
10  struct _Employee_ {
11    char* firstname;
12    char* lastname;
13    char* title;
14    Address* home;
15    Address* office;
16  };
17  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:
  - Punkte im $\mathbb{R}^3$
  - allgemeine Vektoren
  - Matrizen

Strukturen für Punkte im $\mathbb{R}^3$

- Struktur zur Speicherung von $v = (x, y, z) \in \mathbb{R}^3$

```
1  // Declaration of structure
2  struct _Vector3_ {
3    double x;
4    double y;
5    double z;
6  };
7
8  // Declaration of corresponding data type
9  typedef struct _Vector3_ Vector3;
```

- kann Struktur-Deklaration und Datentyp-Definition verbinden

```
1  typedef struct _Vector3_ {
2    double x;
3    double y;
4    double z;
5  } Vector3;
```


Abstand zweier Punkte im $\mathbb{R}^3$

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <math.h>
4
5  typedef struct _Vector3_ {
6    double x;
7    double y;
8    double z;
9  } Vector3;
10
11 Vector3* newVector3(double x, double y, double z) {
12   Vector3* v = malloc(sizeof(Vector3));
13   v->x = x;
14   v->y = y;
15   v->z = z;
16   return v;
17 }
18
19 Vector3* delVector3(Vector3* v) {
20   free(v);
21   return NULL;
22 }
23
24 double dist(Vector3* v, Vector3* w) {
25   return sqrt( (v->x - w->x)*(v->x - w->x)
26                + (v->y - w->y)*(v->y - w->y)
27                + (v->z - w->z)*(v->z - w->z) );
28 }
29
30 main() {
31   Vector3* v = newVector3(1,1,1);
32   Vector3* w = newVector3(1,2,3);
33   printf("dist(x,y) = %f\n", dist(v,w));
34   v = delVector3(v);
35   w = delVector3(w);
36 }
```


Strukturen und Vektoren

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

```
1  // Declaration of structure
2  struct _Vector_ {
3    int n;          // Dimension
4    double* entry;  // Vector coefficients
5  };
6
7  // Declaration of corresponding data type
8  typedef struct _Vector_ Vector;
```

- kann Struktur-Deklaration und Datentyp-Definition verbinden

```
1  typedef struct _Vector_ {
2    int n;          // Dimension
3    double* entry;  // Vector coefficients
4  } Vector;
```

Allokieren eines Vektors

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

```
1   Vector* newVector(int n) {
2     Vector* X = malloc(sizeof(Vector));
3     int i = 0;
4
5     X->n = n;
6     X->entry = malloc(n*sizeof(double));
7
8     for (i=0; i<n; ++i) {
9       X->entry[i] = 0;
10    }
11
12    return X;
13  }
```

Freigeben eines Vektors

- Datenfeld freigeben
- Struktur freigeben
- `NULL` zurückgeben

```
1   Vector* delVector(Vector* X) {
2     free(X->entry);
3     free(X);
4
5     return NULL;
6   }
```


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

```
1   int getVectorN(Vector* X) {
2     return X->n;
3   }
4
5
6   double getVectorEntry(Vector* X, int i) {
7     return X->entry[i];
8   }
9
10
11  void setVectorEntry(Vector* X, int i, double Xi){
12    X->entry[i] = Xi;
13  }
```

- Wenn kein `set`, dann Schreiben nicht erlaubt!
- Wenn kein `get`, dann Lesen nicht erlaubt!
- Dieses Vorgehen erlaubt leichte Umstellung der Datenstruktur bei späteren Modifikationen

Beispiel: Vektor einlesen

```
1   Vector* inputVector() {
2
3     Vector* X = NULL;
4     int i = 0;
5     int n = 0;
6     double input = 0;
7
8     printf("Dimension des Vektors n=");
9     scanf("%d",&n);
10
11    X = newVector(n);
12
13    for (i=0; i<n; ++i) {
14      input = 0;
15      printf("x[%d]=",i);
16      scanf("%lf",&input);
17      setVectorEntry(X,i,input);
18    }
19
20    return X;
21  }
```

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

Beispiel: Euklidische Norm

```
1   double normVector(Vector* X) {
2
3     int n = getVectorN(X);
4     int i = 0;
5     double Xi = 0;
6     double norm = 0;
7
8     for (i=0; i<n; ++i) {
9       Xi = getVectorEntry(X,i);
10      norm = norm + Xi*Xi;
11    }
12    norm = sqrt(norm);
13
14    return norm;
15  }
```

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

Beispiel: Skalarprodukt

```
1   double productVector(Vector* X, Vector* Y) {
2
3     int n = getVectorN(X);
4     double Xi = 0;
5     double Yi = 0;
6     double product = 0;
7     int i = 0;
8
9     for (i=0; i<n; ++i) {
10      Xi = getVectorEntry(X,i);
11      Yi = getVectorEntry(Y,i);
12      product = product + Xi*Yi;
13    }
14
15    return product;
16  }
```

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

Strukturen und Matrizen

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

```
1   // Declaration of structure
2   struct _Matrix_ {
3     int m;          // Dimension
4     int n;
5     double** entry; // Matrix entries
6   };
7
8   // Declaration of corresponding data type
9   typedef struct _Matrix_ Matrix;
```

- kann Struktur-Deklaration und Datentyp-Definition verbinden

```
1   typedef struct _Matrix_ {
2     int m;          // Dimension
3     int n;
4     double** entry; // Matrix entries
5   } Matrix;
```


Allokieren einer Matrix

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

```
1   Matrix* newMatrix(int m, int n) {
2     int i = 0;
3     int j = 0;
4
5     Matrix* A = malloc(sizeof(Matrix));
6
7     A->m = m;
8     A->n = n;
9     A->entry = malloc(m*sizeof(double*));
10
11    for (i=0; i<m; ++i) {
12      A->entry[i] = malloc(n*sizeof(double));
13      for (j=0; j<n; ++j) {
14        A->entry[i][j] = 0;
15      }
16    }
17
18    return A;
19  }
```


Freigeben einer Matrix

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

```
1   Matrix* delMatrix(Matrix* A) {
2     int i = 0;
3
4     for (i=0; i<A->m; ++i) {
5       free(A->entry[i]);
6     }
7
8     free(A->entry);
9
10    free(A);
11
12    return NULL;
13  }
```

Zugriffsfunktionen

```
1   int getMatrixM(Matrix* A) {
2     return A->m;
3   }
4
5
6   int getMatrixN(Matrix* A) {
7     return A->n;
8   }
9
10
11  double getMatrixEntry(Matrix* A, int i, int j) {
12    return A->entry[i][j];
13  }
14
15
16  void setMatrixEntry(Matrix* A, int i, int j,
17                      double Aij) {
18    A->entry[i][j] = Aij;
19  }
```


Beispiel: Matrix-Vektor-Produkt

```
1   Vector* matrixvector(Matrix* A, Vector* X) {
2
3     int m = getMatrixM(A);
4     int n = getMatrixN(A);
5     Vector* B = newVector(m);
6     double Aij = 0;
7     double Xj = 0;
8     double Bi = 0;
9     int i = 0;
10    int j = 0;
11
12    for (i=0; i<m; ++i) {
13      Bi = 0;
14      for (j=0; j<n; ++j) {
15        Aij = getMatrixEntry(A,i,j);
16        Xj = getVectorEntry(X,j);
17        Bi = Bi + Aij*Xj;
18      }
19      setVectorEntry(B,i,Bi);
20    }
21
22    return B;
23  }
```

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

Beispiel: Zeilensummennorm

```
1   double normMatrix(Matrix* A) {
2
3     int m = getMatrixM(A);
4     int n = getMatrixN(A);
5     double Aij = 0;
6     double max = 0;
7     double sum = 0;
8     int i = 0;
9     int j = 0;
10
11    for (i=0; i<m; ++i) {
12      sum = 0;
13      for (j=0; j<n; ++j) {
14        Aij = getMatrixAij(A,i,j);
15        sum = sum + fabs(Aij);
16      }
17      if (sum > max) {
18        max = sum;
19      }
20    }
21
22    return max;
23  }
```

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

Strukturen und Matrizen, v2

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

```
1   typedef struct _Matrix_ {
2     int m;
3     int n;
4     double* entry;
5   } Matrix;
```

- Allokieren der neuen Matrix-Struktur

```
6   Matrix* newMatrix(int m, int n) {
7     int i = 0;
8
9     Matrix* A = malloc(sizeof(Matrix));
10
11    A->m = m;
12    A->n = n;
13    A->entry = malloc(m*n*sizeof(double));
14
15    for (i=0; i<m*n; ++i) {
16      A->entry[i] = 0;
17    }
18
19    return A;
20  }
```

Noch einmal free, set, get

- Freigeben der neuen Matrix-Struktur

```
1   Matrix* delMatrix(Matrix* A) {
2     free(A->entry);
3     free(A);
4     return NULL;
5   }
```

- `set` und `get` für Matrix-Einträge

```
7   void setMatrixEntry(Matrix* A, int i, int j,
8                       double Aij) {
9     A->entry[i+j*A->m] = Aij;
10  }

11  double getMatrixEntry(Matrix* A, int i, int j) {
12    return A->entry[i+j*A->m];
13  }
```

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

C++

- Was ist C++
- Datenkapselung
- `Hello World!` mit C++
- `string`

Was ist C++

- Eine Weiterentwicklung von C
- Objektorientierte Programmiersprache (OOP)
- Objekte stehen im Vordergrund bei OOP
- Reale Objekte als Software Objekte abgebildet
  - Auto
  - Matrix
- Zusammenfassung von Daten und Funktionen zu Objekten
  - Funktionalität hängt von Daten ab
    - vgl. Multiplikation für Skalar, Vektor, Matrix
  - Realisierung mittels Klassen in C++
- Datenkapselung!
  - Kein direkter Zugriff auf Daten
  - Erfolgt über Schnittstellen

Good to know

- C++ ist eine *höhere Programmiersprache*
- Entwicklung ab 1979 bei AT&T
  - Entwickler: Bjarne Stroustrup
- Erweiterung von C
  - Einführung eines Klassenkonzeptes (C with classes)
  - Abwärtskompatibel (zu C) $\Rightarrow$ keine Syntaxkorrektur
- Heute:
  - voll objektorientiert
  - ISO genormt
- Inspiration für andere Sprachen
  - C#, Java
- Compiler:
  - `g++`, `gpp` frei verfügbar
  - Microsoft Visual C++ Compiler
  - Borland C++ Compiler

Was kann mir helfen?

- Endlose Anzahl an Büchern
  - *Die C++ Programmiersprache*
  - *Jetzt lerne ich C++*
  - *C++ in 21 Tagen*
  - *C++ für C Programmierer*
- Online Tutorien
  - einfach googeln
- C++ API
  - C++ Befehlsreferenz
    `http://en.cppreference.com/w/cpp`
    `http://www.cplusplus.com`
- Entwicklungsumgebungen
  - *engl. Integrated Development Environment* (IDE)

Hello World 1/2

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

- Speichern unter `helloworld.cpp`
- Compilieren mittels `g++ helloworld.cpp`
- Ausgabe: `Hello World!`
- Standardbibliothek für Ein- und Ausgabe in C++ heißt iostream
  - Stream für Lesen und Schreiben von Dateien in Bibliothek `fstream` definiert
- `cout` ist der Standard-Ausgabestream
  - kann unterschiedliche Datentypen übernehmen
  - `cin` ist Standard-Ausgabestream
- Scope-Operator `::` deklariert den Namensbereich
- Operator `<<` übergibt sein zweites Argument (rechtes Argument) an den Ausgabestream
- `main` hat Rückgabewert `int`

Hello World 2/2

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

- Zeile 2: `using namespace std;`
  - Bereich `std` wird durchgehend benutzt
  - Benutzung von Funktionen wie `cout` ohne vorangestelltes `std::` möglich
- Ausgabe mit `cout`:
  - Verschiedene Datentypen
  - Mehrere Ausgaben hintereinander werden durch Operator `<<` getrennt
- Beispiel:
```
int x = 5;
double y = 0.25
cout << x << " * "  << y << " = " << x*y;
```
- Ausgabe: `x * y = 1.25`

Datentyp `string` 1/2

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

- Ausgabe:
```
Hallo Welt!
Hallo Peter!
```
- Strings sind mächtiger als `char*`
- liefert eine Reihe nützlicher Operationen
  - '+' zur Konkatenation
  - `replace` zum ersetzen von Teilstrings
  - `length` zum auslesen der Länge u.v.m.
- Datentypen können Methoden haben
  - Prinzip der Datenkapselung
  - Sie enthalten mehr als nur die Zeichen (Unterschied zu C)
- Mittels Punktoperator (`.`) Zugriff auf Member eines Objekts
  - Insbesondere auch auf Methoden (=Funktionen) eines Objekts

Datentyp `string` 2/2

```
1   #include <iostream>
2   #include <string>
3   #include <stdio.h>
4   using namespace std;
5
6   int main() {
7     string str1 = "Hallo";
8     string str2 = "Welt";
9     string str3 = str1 + " " + str2;
10
11    printf("%s\n", str3.c_str());
12    cout << str3 << endl;
13
14    return 0;
15  }
```

- Ausgabe: `Hallo Welt`
  `Hallo Welt`
- 'Inhalt' mittels `c_str()` erreichbar
  - Das sind die bekannten `char`-Arrays aus C
  - Zugriff über `name.c_str()`
  - können mittels `printf` ausgegeben werden
- Wichtig: `string` $\neq$ `char`-Array
- Zeile 3: Kann auch `#include <cstdio>` verwenden um C-Standardbibiliotheken einzubinden
- Zeile 12: Zeilenumbruch mit `endl`

Klassen

- Was sind Klassen
- Objekte
- Klassen und Strukturen

Was sind Klassen

- Klassen sind benutzerdefinierte Datentypen
- Verwendung völlig analog zu `int`, `string`
- Klassen erweitern `struct` aus C
  - erlauben Methoden (Funktionen)
- Klassen erlauben das Prinzip der Datenkapselung
  - Code wird besser wiederverwendbar
  - *black-box-Implementierung* möglich
- Klassen sind *Schablonen* für Objekte
- Klassen sind der Grundbaustein zur OO
- Beschreiben Attribute und Methoden, also Verhaltensweisen, der Objekte

Beispiel

```
1   #include <string>
2   using namespace std;
3
4   class Student {
5     string name;
6     string studiengang;
7     int matrikelnummer;
8
9     void lernen();
10    void testSchreiben();
11  };
```

- Deklaration mit Schlüsselwort `class`
- enthält Daten (*members*)
  - analog zu `struct` aus C
- enthält Funktionen (*Methoden*)
  - erlaubt es komplette Objekte abzubilden
- Strichpunkt nach Deklaration

- zu Namenskonvention:
  - C++ ist *case sensitive*
  - Groß- und Kleinschreibung beachten
- Klassen (Coding Standard)
  - `AlleWorteCapitalizedOhneUnderlines`
- Methoden, Funktionen und Variablen
  - `erstesWortKleinRestCapitalizedOhneUnderlines`

Objekte

- Objekte sind Instanzen einer Klasse
  - Sie sind die tatsächlichen Variablen
  - werden angelegt wie bei primitiven Typen
  - entsprechen Variablen vom `struct` Datentyp

```
Student Marcus;
int x;
```

- Zugriff auf Klassenelemente mit Punktoperator (.)

```
Marcus.name = "Marcus Page";
Marcus.lernen();
```

- Achtung: Zugriff muss erlaubt sein
  - sonst Fehlermeldung von Compiler
- Achtung: An Objekte zuweisen, nicht an Klassen

```
int = 5;                        // Fehler
Student = "Marcus Page";        // Fehler
```


Zugriffskontrolle

- Klassen und Objekte dienen der Abstraktion
  - genaue Implementierung nicht wichtig
  - zum Beispiel durch Bibliotheken
- Benutzer soll so wenig wissen wie möglich
  - *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 (später)

Beispiel 1/2

```
1  class Dreieck {
2  private:
3    double x[2];
4    double y[2];
5    double z[2];
6
7  public:
8    double flaeche();
9  };
```

- Dreieck in $\mathbb{R}^2$ mit Eckpunkten `x,y,z`
- Benutzer kann Funktion `flaeche` aufrufen
- Benutzer kann Daten `x,y,z` nicht direkt verändern
  - `get/set` Funktionen in `public`-Bereich einbauen
- Benutzer muss nicht wissen wie Daten intern verwaltet werden
  - Kann interne Datenstruktur später verändern
  - Dreieck kann auch durch einen Punkt und zwei Vektoren abgespeichert werden

Beispiel 2/2

```
1  class Dreieck {
2  private: // kann auch weggelassen werden
3    double x[2];
4    double y[2];
5    double z[2];
6
7  public: // ab hier alle members/methoden public
8    double getFlaeche();
9  };
10
11 int main() {
12   Dreieck tri; // Objekt tri vom Typ Dreieck
13
14   tri.x[0] = 1.0; // Zugriff auf private member!
15
16   return 0;
17 }
```

- Zeile 2: `private:` kann weggelassen werden
  - In Klasse sind alle Members/Methoden standarmäßig `private`
  - Nach `public:` sind alle Members/Meth öffentlich
- Beim Kompilieren tritt Fehler auf

  `double Dreieck::x [2] is private`
  - da Member `x` als `private` deklariert
  - Daher: `get/set`-Funktionen vewenden
    - Nach `public:`

Methoden implementieren 1/2

```
1   #include <cmath>
2   class Dreieck {
3   private:
4     double x[2];
5     double y[2];
6     double z[2];
7   public:
8     void setX(double x0, double x1);
9     double getFlaeche();
10  };
11
12  // Implementierung
13  void Dreieck::setX(double x0, double x1) {
14    x[0] = x0; x[1] = x1;
15  }
16  double Dreieck::getFlaeche() {
17    return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
18                - (z[0]-x[0])*(y[1]-x[1]) );
19  }
20
21  int main() {
22    Dreieck tri; // Objekt tri vom Typ Dreieck
23    tri.setX(0.0,0.0); // Zugriff mit set-Funktion
24
25    return 0;
26  }
```

- Implementierung wie bei anderen Funktionen
  - Unterschied besteht nur in Funktionssignatur
- Syntax: `rType ClassName::funName(...)`
  - `rType` ist Rückgabewert (`void,double,etc.`)
  - `...` sind Übergabeparameter
  - Wichtig: `ClassName::` vor `funName`
    - Methode `funName` gehört zu Klasse `ClassName`

Methoden implementieren 2/2

```
1   #include <iostream>
2   #include <cmath>
3   using namespace std;
4
5   class Dreieck {
6   private:
7     double x[2]; double y[2]; double z[2];
8   public:
9     void setX(double x0, double x1);
10    void setY(double y0, double y1);
11    void setZ(double z0, double z1);
12    double getFlaeche();
13  };
14
15  void Dreieck::setX(double x0, double x1) {
16    x[0] = x0; x[1] = x1;
17  }
18  void Dreieck::setY(double y0, double y1) {
19    y[0] = y0; y[1] = y1;
20  }
21  void Dreieck::setZ(double z0, double z1) {
22    z[0] = z0; z[1] = z1;
23  }
24  double Dreieck::getFlaeche() {
25    return 0.5*fabs( (y[0]-x[0])*(z[1]-x[1])
26                - (z[0]-x[0])*(y[1]-x[1]) );
27  }
28
29  int main() {
30    Dreieck tri;
31    tri.setX(0.0,0.0);
32    tri.setY(1.0,0.0);
33    tri.setZ(0.0,-1.0);
34    cout << "Flaeche= " << tri.getFlaeche() << endl;
35    return 0;
36  }
```


Klassen und Strukturen

- Struktur:
  - Zusammenfassung elementarer Datentypen (wie `int,double`) zu einem Gesamtdatentyp
  - Können von Funktionen manipuliert werden
- Klasse:
  - Enthält neben den Daten auch noch Methoden, welche diese manipulieren
- Realisierung von Strukturen in C mit `struct`
  - Keine Funktionen in Struktur erlaubt
- Realisierung von Klassen in C++ mit `class`
- **ACHTUNG:** `struct` in C $\neq$ `struct` in C++
  - `struct` in C++ wie `class`, nur sind alle Members standardmäßig `public`!

Header-Files

- Funktionsaufrufe
- Warum Header-Files?
- `inline`
- `#ifndef ... #endif`

Funktionsaufrufe

- Funktionsaufrufe sind strukturell langsam
  - Interner Overhead aufgrund des Stacks
- Zugriff auf Members einer Struktur mittels get-Funktion kann daher ineffizient sein
  - trotzdem besserer Stil
  - denn: Trennung von Funktionalität und Daten
- Abhilfe: Verwendung von `inline`:
  - `inline double getMatrixEntry(Matrix* A, int i, int j)`
  - d.h. einfach `inline` vor Funktionssignatur
- Compiler versucht, Funktion durch expliziten Code aufzulösen (als ob keine Fkt verwendet wurde)
  - stammt aus C++
  - wird auch von fast allen C-Compilern akzeptiert

Datei-Konventionen

- Grundregel: Jeder C-Code besteht aus
  - Header-File `file.h`
  - Programmcode `file.c`
- Analog: Jeder C++ Code besteht aus
  - Header-File `file.hpp` oder `file.h`
  - Programmcode `file.cpp`
- Header-File besteht aus
  - `#include` der benötigten Bibliotheken
  - Definition der benötigten Strukturen (`struct`)
  - Definition der neuen Variablentypen (`typedef`)
  - Signaturen zu allen Funktionen aus `file.c`
  - `inline`-Funktionen inkl. Code!
  - Kommentaren zu allen Funktionen
    - Was tut eine Funktion?
    - Was ist Input? Was ist Output?
- Warum Code auf mehrere Files aufteilen?
  - Übersichtlichkeit & Verständlichkeit des Codes
  - Anlegen von Bibliotheken
- Header-File beginnt mit

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

- Header-File ended mit

  ```
  #endif
  ```

- erlaubt mehrfaches Einbinden


```
 1  #ifndef MATRIX
 2  #define MATRIX
 3  #include <stdlib.h>
 4
 5  // data structure to store m x n matrices
 6  typedef struct _Matrix_ {
 7    int m;          // the matrix dimensions
 8    int n;          // of m x n double matrix
 9    double* entry;  // dynamic vector of length m*n
10  } Matrix;
11
12  // allocate and initialize a new matrix
13  Matrix* newMatrix(int m, int n);
14
15  // free matrix and return NULL pointer
16  Matrix* delMatrix(Matrix* A);
17
18  // get functions for matrix structure
19  inline int getMatrixM(Matrix* A);
20  inline int getMatrixN(Matrix* A);
21  inline double getMatrixEntry(Matrix* A, int i, int j);
22
23  // set functions for matrix structure
24  inline void setMatrixEntry(Matrix* A, int i, int j,
25                             double Aij);
26
27  // the code of the inline functions
28  inline int getMatrixM(Matrix* A) {
29    return A->m;
30  }
31
32  inline int getMatrixN(Matrix* A) {
33    return A->n;
34  }
35
36  inline double getMatrixEntry(Matrix* A, int i, int j) {
37    return A->entry[i+j*A->m];
38  }
39
40  inline void setMatrixEntry(Matrix* A, int i, int j,
41                             double Aij) {
42    A->entry[i+j*A->m] = Aij;
43  }
44  #endif
```


Bibliotheken

- Aufteilen von Source-Code auf mehrere Files
- Object-File
- Shared Object-File

- `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* Exe aus mehreren Source-Codes
  - analog zu `gcc all.c`
    - wenn `all.c` ganzen Source-Code enthält
  - insb. Funktionsnamen müssen eindeutig sein
  - `main()` darf nur 1x vorkommen
  - Reihenfolge der Codes nicht wichtig

Precompiler, Compiler & Linker

- Beim Kompilieren von *Source-Code* (z.B. mit `gcc`) werden mehrere Stufen durchlaufen:

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

  (2) *Compiler* erstellt *Object-Code*

  (3) *Object-Code* aus Bibliotheken hinzufügen

  (4) *Linker* ersetzt symbolische Namen im Object-Code und erstellt *Executable*

- 2 Arten von *Object-Code* für Bibliotheken
  - Bibliotheken, die erst bei Ausführung des Executable eingebunden werden: `libname.so` (*shared object*)
  - Bibliotheken in Form von Object-Code, die in Schritt (3) direkt eingebunden werden: `name.o`

Statische Bibliotheken

- Ziel: Bibliothek mit Fktn aus `bib.c`
- Ziel: Executable `exe`, das Bibliothek einbindet
- Vorhandener Source-Code
  - `prg.c`, `bib.c`, `bib.h`
  - `prg.c` und `bib.c` enthalten `#include "bib.h"`
  - Bisher: `prg.c` enthielt `#include "bib.c"`

(1) Object-Code aus Source-Code erzeugen
  - `gcc -c prg.c` erzeugt `prg.o`
  - `gcc -c bib.c` erzeugt `bib.o`

(2) Object-Code verbinden & Linken
  - `gcc -o exe prg.o bib.o`
  - erzeugt Executable `exe` aus Object-Codes
  - Reihenfolge & Anzahl der Object-Codes egal

- Vorteil: Zeitersparnis beim Compilieren
  - Nach Änderungen lediglich Object-Code `prg.o` neu erstellen und Object-Codes linken, denn Object-Code der Bibliothek ändert sich ja nicht!
- formal `gcc` mit beliebig viel Source-Code und Object-Code aufrufbar
  - `gcc -o exe name1.c name2.c bib1.o bib2.o ...`
  - Reihenfolge nicht wichtig!

Statische Bibliotheken & `make`

- UNIX-Befehl `make` erkennt, wenn Source-Code geändert wurde und erzeugt dann und nur dann den zugehörigen Objekt-Code ⇒ Automatisierung
- Aufruf:
  - `make` wertet Datei `Makefile` aus
  - `make -f filename` wertet Datei `filename` aus
- Befehlsliste für `make` enthält, z.B.

```
1  exe : prg.o bib.o
2          gcc -o exe prg.o bib.o
3
4  prg.o : prg.c bib.h
5          gcc -c prg.c
6
7  bib.o : bib.c bib.h
8          gcc -c bib.c
```

- `Makefile` besteht aus Abhängigkeiten und Befehlen
- BSP für Abhängigkeit (*nicht* eingerückt): Zeile 1
- Zieldatei `exe` hängt ab von Dateien
  - `prg.o`
  - `bib.o`
- Falls Zieldatei älter als Dateien, von denen sie abhängt, werden Befehle ausgeführt
- BSP für Befehl (*stets* eingerückt): Zeile 2
  - Befehlszeile beginnt mit Tabulator-Einrückung!
- (Viel!) mehr zu `make` bei Schmaranz Kap. 15

Dynamische Bibliotheken

- Dynamische Bibliothek besteht aus zwei Dateien
  - Header-File `name.h` mit Funktionsdeklarationen
  - *Shared Object-Code* `libname.so` der Funktionen
- Name `name` von beiden Files kann verschieden sein
- BSP: `math`-Bibliothek
  - Im *Source-Code*: `#include <math.h>`
  - Beim Compilieren: `gcc file.c -lm`
    - bindet zugehörigen *Object-Code* `libm.so` aus Standardverzeichnis ein
- Erstellen von dynamischen Bibliotheken
  - `gcc -shared -o libbib.so bib.c`
- Einbinden von dynamischen Bibliotheken
  - In `prg.c`: `#include "bib.h"`
  - `gcc -o exe2 -L. -lbib`
    - Option `-Lverzeichnis` gibt Verzeichnis, in dem die dynamische Bibliothek liegt.
    - Option `-L.` bezeichnet *aktuelles* Verzeichnis
    - Bei manchen UNIX-Systemen *Path*-Variable setzen, um Programm auszuführen
      ↦ `export LD_LIBRARY_PATH=verzeichnis`

Dynamische Bibliotheken & `make`

- Ziel: dyn. Bibliothek mit Fktn aus `bib.c`
- Ziel: Executable `exe2`, das Bibliothek einbindet
- Vorhandener Source-Code
  - `prg.c`, `bib.c`, `bib.h`
  - `prg.c` hat `#include "bib.h"`
  - `make` funktioniert wie im ersten Fall auch

```
1  exe2 : prg.o libbib.so
2         gcc -o exe prg.o -L. -lbib
3
4  prg.o : prg.c bib.h
5         gcc -c prg.c
6
7  libbib.so :  bib.c  bib.h
8         gcc -shared -o libbib.so bib.c
9
```

- Aufruf von `exe2` liefert häufig Fehlermeldung:
  exe2: error while loading shared libraries: libbib.so: cannot open shared object file: No such file or directory
  ⇒ Path-Variable setzen: `export LD_LIBRARY_PATH=.`
- Vergleich der beiden Executable `exe`, `exe2` zeigt:
  - `exe` (Statische Bibliothek) ↦ 13447 Bytes
  - `exe2` (Dynamische Bibliothek) ↦ 12548 Bytes
  - Beide Programme liefern dasselbe Ergebnis
  - `exe` ist (generisch) ein wenig schneller als `exe2`

Klassen II

- Header-Files
- Konstruktor
- Destruktor

Deklaration in Header-Datei

- Trennen von Deklaration und Implementierung
- Deklaration der Klasse `ClassName` in Header-File
  - `ClassName.h` (oder `ClassName.hpp`)
- Implementierung der zugehörigen Methoden in
  - `ClassName.cpp`
  - Einbinden des Header-File `ClassName.h`
- Grundsätzlich: Jede Klasse aufteilen
- Steht restlicher Programmcode in `prog.cpp`
  - Kompilieren z.B. mit
    `g++ prog.cpp ClassName.cpp ClassName2.cpp`
  - Bzw. siehe Kapitel Bibliotheken
- Einsatz von `Makefile` und `make` sinnvoll!

Beispiel

- student.h

```
 1  #ifndef _STUDENT_H
 2  #define _STUDENT_H
 3
 4  #include <string>
 5  using namespace std;
 6
 7  class Student {
 8  private:
 9    string name;
10    int matrikelnummer;
11  public:
12    void lernen();
13    void testSchreiben();
14  };
15
16  #endif // _STUDENT_H
```

- student.cpp

```
 1  #include <iostream>
 2  #include "student.h"
 3  using namespace std;
 4
 5  void Student::lernen() {
 6    cout << "oh mann ..." << endl;
 7  }
 8
 9  void Student::testSchreiben() {
10    cout << "geschafft!" << endl;
11  }
```


Methoden direkt implementieren

- student.h

```
 1  #ifndef _STUDENT_H
 2  #define _STUDENT_H
 3
 4  #include <string>
 5  #include <iostream>
 6  using namespace std;
 7
 8  class Student {
 9  private:
10    string name;
11    int matrikelnummer;
12  public:
13    void lernen() {cout << "oh mann ..." << endl;};
14    void testSchreiben();
15  };
16
17  #endif // _STUDENT_H
```

- student.cpp

```
 1  #include "student.h"
 2
 3  void Student::testSchreiben() {
 4    cout << "geschafft!" << endl;
 5  }
```

- Implementierung direkt bei Klassendef. möglich
- In geschweiften Klammern
  - für kurze Methoden (`get, set`)
  - unübersichtlich für längere Methoden

Wozu Zugriffskontrolle? 1/2

```
 1  class Bruch {
 2  public:
 3    int zaehler;
 4    unsigned int nenner;
 5  };
 6
 7  int main() {
 8    Bruch meinBruch;
 9    meinBruch.zaehler = -1000;
10    meinBruch.nenner = 0;
11
12    return 0;
13  }
```

- Wie sinnvolle Werte sicherstellen? (Z. 10)
  - mögliche Fehlerquellen direkt ausschließen
  - Programmierer muss sich um möglichst wenig kümmern
  - ⇒ Lösung: `get` und `set` Funktionen für Memberdaten `zaehler, nenner` einbauen

Wozu Zugriffskontrolle? 2/2

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  class Bruch {
 5  private:
 6    int zaehler;
 7    unsigned int nenner;
 8  public:
 9    int getZ() { return zahler;};
10    unsigned int getN() { return nenner;};
11    void setZ(int z) { zaehler = z;};
12    void setN(unsigned int n);
13  };
14
15  void Bruch::setN(unsigned int n) {
16    if(n==0) {
17      cout << "Nenner darf nicht 0 sein!" << endl;
18      nenner = 1;
19    }
20    else
21      nenner = n;
22  }
23
24  int main() {
25    Bruch meinBruch;
26    meinBruch.setN(0);
27
28    return 0;
29  }
```

- Ausgabe: `Nenner darf nicht 0 sein!`
- Direkter Zugriff auf `zaehler,nenner` nicht möglich

Warum so viel Kontrolle?

- Fakt ist: alle Programmierer machen Fehler
  - Code läuft beim ersten mal nie richtig
- Großteil der Entwicklungszeit geht in Fehlersuche
- Wie unterscheiden sich Profis von 'Anfängern' ?
  - durch effizientere Fehlersuche
- Compiler-Fehler sind leicht einzugrenzen
  - es steht Zeilennummer dabei
- Laufzeitfehler sind viel schwieriger zu finden
  - Programm 'läuft', tut aber nicht das richtige
  - manchmal fällt der Fehler ewig nicht auf
    $\Rightarrow$ sehr schlecht
- $\Rightarrow$ möglichst viele Fehler durch Compiler abfangen
- Methoden werden mit Verstand geschrieben
- das sollte sich im Code wiederspiegeln
  - gehören Daten strikt zu einer Klasse $\Rightarrow$ `private`
  - Zugriff kontrollieren mittels `get` und `set`
    (reine Daten sollten immer `private` sein)

Konstruktoren 1/2

```
1   #include <iostream>
2   using namespace std;
3
4   class Auto {
5   private:
6     double preis;
7     int alter;
8   public:
9     Auto();
10    double getP() {return preis;};
11  };
12
13  Auto::Auto() {
14    preis = 0.0;
15    alter = 0;
16  }
17
18  int main() {
19    Auto test;
20    cout << "Preis = " << test.getP() << endl;
21  }
```

- Konstruktoren dienen zum Initialisieren
- Automatisch nach Erzeugung aufgerufen (Z.19)
- Standardkonstruktor `Auto()` (Z.9)
  - Name: `ClassName()`
  - keine Inputparameter
- Konstruktoren besitzen keinen Rückgabewert

Konstruktoren 2/2

```
1   class Auto {
2   private:
3     double preis;
4     int alter;
5   public:
6     Auto() : preis(0.0),alter(0) {};
7     Auto(double preis, int alter);
8   };
9
10  Auto::Auto(double preis, int alter) {
11    this->preis = preis;
12    this->alter = alter;
13  }
14
15  int main() {
16    Auto test;              // test.preis == 0.0
17    Auto test2(999.90,10); // test2.preis == 999.9
18
19    return 0;
20  }
```

- Alle Konstruktoren heißen wie die Klasse
  - unterscheiden sich durch ihre Signatur
  - also Aufruf je nach Signatur!
- Kurzschreibweise mittels Doppelpunkt (:) (Z. 6)
  - direkte Zuweisung an Membervariablen
  - Aufruf von Konstr falls Member Obj einer Klasse
- Pointer `this` zeigt auf das Objekt selbst
  - kann lokale Namensgleichheiten auflösen
  - nützlich wenn man das Objekt übergeben muss

Destruktoren

```
1   class Auto {
2   private:
3     double preis;
4     int alter;
5   public:
6     Auto() : preis(0.0),alter(0) {};
7     Auto(double p, int a) : preis(p),alter(a) {};
8     ~Auto();
9   };
```

- Beim Auflösen des Objektes aufgerufen
  - Verlassen eines Blocks (vgl. Scope & Lifetime)
  - Funktions- bzw. Programmende
  - Mit `delete` (später)
- Name des Destruktors = `~ClassName()`
- Es gibt nur einen Destruktor (Standarddestruktor)
  - besitzt keine Ein- u. Rückgabewerte
  - wird ggfs. automatisch generiert
  - kann bei einfachen Objekten (nur elementare Datentypen) weggelassen werden
- Komplizierte Objekte müssen aufgeräumt werden
  - Freigeben eines dynamisch angelegten Arrays
  - Schließen von offenen Dateien

Bsp zu Konstruktor & Destruktor

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

- Ausgabe:

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


Schachtelung von Klassen

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

- Klassen können geschachtelt werden
  - Standardkonstr/-destr automatisch aufgerufen
  - Konstruktoren der Member zuerst
  - Allg. Konstr mittels Doppelpkt (:)
  - Destruktoren der Member zuletzt
- Ausgabe:

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


Eigene Vektorklasse

```
 1  #include <cstdlib>   // malloc, realloc, free
 2
 3  class MyVector {
 4    double * entries;
 5    int size;
 6  public:
 7    MyVector() { size = 0; entries = NULL;};
 8    MyVector(int size, double init = 0);
 9    ~MyVector();
10  };
11
12  MyVector::MyVector(int size, double init) {
13    this->size = size;
14    entries = (double*)malloc(size*sizeof(double));
15    for(int j=0; j<size; ++j)
16      entries[j] = init;
17  }
18
19  MyVector::~MyVector() {
20    if(size>0)
21      free(entries);
22  }
```

- Allg. Konstruktor (Z.8)
  - Array mit Länge `size`, Elte auf Wert `init`
  - Erzeugen eines Obj: `MyVector vec(5,1.0);`
  - Alternativ: `MyVector vec(5);`
    - Vektor mit Länge 5 und Einträge 0
    - Defaultwert bei Dekl des Konstr angeben
- Destruktor gibt Speicher bei Auflösen des Obj frei
  - Ohne Destr: Nur Speicher von Pointer frei
- Def. von Zählvariable direkt in `for` Schleife möglich

Standardbibliotheken

- Ein- und Ausgabe
- Vektoren
- Container

- `bool`
- `cout,cin`
- `vector`

Datentyp `bool`

- In C gibt es keinen logischen Datentyp
- Abhilfe schafft Interpretation
  - `0 == false`
  - `1 == true`
- Das könnte so aussehen:

```
#define false 0
#define true 1
typedef int bool;
```

- Könnte auch mit Abfragen gelöst werden
- C++ enthält logischen Datentyp `bool`
  - Werte `true` und `false`
- Implizite Konversion arithmetischer Typen
  - `0` entspricht `false`
  - alles andere entspricht `true`

Eingaben 1/2

- Standardbibliothek zur Ein-/Ausgabe `iostream`
- Ausgabestream `cout` mit Shiftoperator <<
- Eingabestream `cin` mit Shiftoperator >>
  - Elementare Datentypen `int`, `double`, ...
  - Objekte der Klasse `string`
  - Unterschied zu `scanf`: Keine Platzhalter nötig

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main() {
 5    int iwert;
 6    double dwert;
 7    cout << "Geben Sie einen Integerwert ein: ";
 8    cin >> iwert;
 9    cout << "Wert = " << iwert << endl;
10    cout << "Geben Sie einen Doublewert ein: ";
11    cin >> dwert;
12    cout << "Wert = " << dwert << endl;
13
14    return 0;
15  }
```

- Ausgabe:

```
Geben Sie einen Integerwert ein: 10
Wert = 10
Geben Sie einen Doublewert ein: 1.1
Wert = 1.1
```


Eingaben 2/2

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  int main() {
 6    string str;
 7    cout << "Geben Sie Ihren Namen ein\n";
 8    cin >> str;
 9    cout << "Hallo " << str << "!\n";
10
11    return 0;
12  }
```

- Eingabe: 'Praetorius'
  - Ausgabe: `Hallo Praetorius!`
- Eingabe: 'Dirk Praetorius'
  - Ausgabe: `Hallo Dirk!`
- Verwende `getline` für ganze Zeile

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  int main() {
 6    string str;
 7    cout << "Geben Sie Ihren Namen ein\n";
 8    getline(cin, str);
 9    cout << "Hallo, " << str << "!\n";
10
11    return 0;
12  }
```

- Eingabe: 'Dirk Praetorius'
  - Ausgabe: `Hallo, Dirk Praetorius!`

Vektoren

- Verwendung als dynamische Arrays
- Ohne `malloc, realloc, free`
- Sind 'Container' für beliebige Datentypen
  - (mehr dazu später)
- Speicher wird durch Destruktor freigegeben
  - Benutzer muss sich nicht um Freigabe kümmern

Vektoren 1/3

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   struct Eintrag {
6     string name;
7   };
8
9   int main() {
10    Eintrag telbuch[3];
11    telbuch[0].name = "Peter Pan";
12    telbuch[1].name = "Wolverine";
13    telbuch[2].name = "Angela Merkel";
14    cout << telbuch[2].name + "\n";
15  }
```

- Ausgabe: `Angela Merkel`
- Verwendung sinnvoll falls Größe bekannt
- Problem: Speicher nicht dynamisch erweiterbar
  - Lösung in C: Pointer, `malloc, realloc, free`
- Lösung in C++: Vektoren
  - Automatische Speicherverwaltung
- Hinweis: `return 0;` am Ende von `main` weggelassen
  - Manche C++ Compiler setzen das automatisch

Vektoren 2/3

```
1   #include <string>
2   #include <vector>
3   using namespace std;
4
5   struct Eintrag {
6     string name;
7   };
8
9   int main() {
10    vector<Eintrag> telbuch(2);
11    telbuch[0].name = "Peter Pan";
12    telbuch[1].name = "Wolverine";
13  }
```

- In Z.11: Vektor der Länge 2 wird angelegt. Einträge im Vektor sind vom Datentyp `Eintrag`
- `Vektoren` sind C++ Standardcontainer
  - man kann beliebige Datentypen verwenden
  - dienen zum Verwalten von Datenmengen
- Verwendung: `vector<type> name(size);`
  - Achtung, nicht verwechseln:

    `vector<Eintrag> buch(1000);` *1000 Einträge*

    `vector<Eintrag> buecher[1000];` *1000 Vektoren*
- Zugriff auf j-tes Element wie bei Arrays
  - `name[j]` (Z. 12–13)

Vektoren 3/3

```
1   #include <iostream>
2   #include <string>
3   #include <vector>
4   using namespace std;
5
6   struct Eintrag {
7     string name;
8   };
9
10  int main() {
11    vector<Eintrag> telbuch(1);
12    telbuch[0].name = "Peter Pan";
13    cout << "size: " << telbuch.size() << endl;
14    telbuch.resize(telbuch.size()+4);
15    cout << "size: " << telbuch.size() << endl;
16  }
```

- Ausgabe:
```
size: 1
size: 5
```
- Speicher dyn veränderbar mittels `resize(newsize)`
- Weitere hilfreiche Funktionen:
  - `size()` Gibt Länge des Vek zurück
  - `push_back(val)` Fügt `val` am Ende ein (vgl Liste!)
  - `pop_back()` Gibt letztes Ele des Vek zurück und löscht es aus Vek
  - uvm.

Weitere Standardcontainer

- `list` (verkettete Listen)
- `queue`
- `stack`
- `deque`
- `set`
- `multiset`
- `map`
- `multimap`

- Weitere C++ Bibliotheken
  - Boost Library: Große Sammlung an Bib.
  - Teile davon bald in C++ Standardbib.
  - `http://www.boost.org`
  - `http://http://www.highscore.de/cpp/boost/`

Ratschläge und Unterschiede zu C

- Ratschläge zu C++
  - Bibliotheken verwenden
  - Standardbibliothek eher als andere
- Unterschiede zu C
  - `string` anstelle von `char*`
  - `vector<T>` anstelle von `T[]`
  - `cout` anstelle von `printf`
  - Std. Bib. verwendet den Namensbereich `std`
  - `struct` in C $\neq$ `struct` in C++
    - `struct` in C darf keine Fkt enthalten
    - `struct` in C++ wie `class`, nur alle Member automatisch `public` statt `private`
    - Insbes. existieren Konstr./Destr. für `struct` in C++
    - D.h. Initialisierung von Strukturobj möglich
    - Keine strikte Trennung zw Konzept **Struktur** und **Klasse** in C++!

C++ und Pointer

- Dynamische Speicherverwaltung
- Pointer auf Objekte
- `new, new[]`
- `delete, delete[]`

Dyn. Speicherverwaltung 1/3

- Speicherallokierung eines Obj bisher mit `malloc`
  - `type* name=(type*)malloc(length*sizeof(type))`
    - `int* i1 = malloc(sizeof(int));`
      `*i1 = 9;`
    - `int* i2 = (int*) malloc(128*sizeof(int));`
- `malloc` kann auch für komplexere Datentypen (Klassen) verwendet werden
  - **Problem:** Konstr werden nicht aufgerufen
  - Abhilfe schafft `new` bzw. `new[]`
- `type* name = new type`
  - Allokiert Speicher für ein Obj vom Typ `type`
  - `int* i1 = new int;`
    `*i1 = 9;`
  - Defaultkonstr. wird automatisch aufgerufen
- `type* name = new type(varlist)`
  - Allokiert Speicher für ein Obj vom Typ `type`
  - Ruft Konstr mit Signatur passend zu `varlist` auf
  - `varlist` ist dabei eine Liste von Variablen
  - `int* i1 = new int(9);`
- `type* name = new type[length]`
  - Allokiert Array der Länge `length` und Typ `type`
  - Standardkonstr werden für alle Ele aufgerufen
  - `int* i2 = new int[128];`

Dyn. Speicherverwaltung 2/3

- Bisher: Freigeben von Speicher mit `free`
  - Speicher wurde zuvor mit `malloc` allokiert
  - **Problem:** Destr werden nicht aufgerufen
  - Abhilfe schafft `delete` bzw. `delete[]`
- `delete name`
  - Ruft Destruktor auf und gibt Speicher frei
  - Zu `name` gehörendes Obj wurde erzeugt mit
    `type* name = new type` bzw.
    `type* name = new type(varlist)`
  - Setze Pointer `name` auf `NULL` oder `0` nach Freigabe
- `delete[] name`
  - dient zum Freigeben von dynamischen Arrays
  - Array wurde erzeugt mit
    `type* name = new type[length]`
  - Ruft Destr aller Ele auf und gibt allokierten Speicher frei
  - Setze Pointer `name` auf `NULL` oder `0` nach Freigabe
- **ACHTUNG:** 4 unterschiedliche Operatoren
  - `new, new[]`
  - `delete, delete[]`
  - Nicht mit `malloc, free` verwechseln/-mischen
  - Kein entsprechender Operator für `realloc`

Dyn. Speicherverwaltung 3/3

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  class Test {
 5  public:
 6    Test() {cout << "Std.konstr" << endl;}
 7    Test(int nr) {cout << "Wert= " << nr << endl;}
 8    Test(int i1, int i2) { cout << i1+i2 << endl;}
 9    ~Test() { cout << "Destr." << endl;}
10  };
11
12  int main() {
13    Test* p1 = new Test;
14    delete p1;
15    p1 = 0;  // Oder p1 = NULL;
16
17    p1 = new Test(1);
18    Test* p2 = new Test(3,4);
19    delete p2;
20    delete p1;
21    p2 = 0;
22    p1 = NULL;
23
24    Test* pArr = new Test[3];
25    delete[] pArr;
26    pArr = 0;
27
28    cout << "Und jetzt mit malloc/free" << endl;
29    // Folgendes liefert keine Ausgabe
30    Test* m1 = (Test*)malloc(sizeof(Test));
31    free(m1);
32
33    return 0;
34  }
```


Pointer auf Objekte 1/2

- Zur Erinnerung:
  - *-Operator zum Deklarieren und Dereferenzieren eines Pointers
  - Adressoperator &
  - `type* pName = &varName`
    - `pName` ist Name des Pointers
    - `&varName` ist Adresse der Variable `varName`
    - Mit `*pName` auf Inhalt zugreifen
    - `type` kann beliebiger Datentyp sein (Klasse)
- Implementierung von Call by Reference
  - Sinnvoll bei großen Objekten
    - z.B. Matrizen (mit großen Dimensionen)
  - Mehrere Rückgabeparameter
  - Kann auch mit Referenzen realisiert werden (vgl. nächstes Kapitel)
- Array von Objekten

Pointer auf Objekte 2/2

- Ziel: Eine Funktion welche eine Matrix plottet
  - Übergabe mittels Call by Reference

```
 1  #include "matrix.h"
 2  #include <iostream>
 3  using namespace std;
 4
 5  void plotMatrix(Matrix* mat) {
 6    for(int i=0; i<mat->getM(); ++i) {
 7      for(int j=0; j<mat->getN(); ++j) {
 8        cout << mat->getEntry(i,j) << " ";
 9      }
10      cout << endl;
11    }
12  }
13
14  int main() {
15    Matrix mat(3,4,1);
16    plotMatrix(&mat);
17
18    Matrix* pMat = &mat;
19    plotMatrix(pMat);
20
21    pMat = new Matrix(); // 1x1 Matrix 0
22    plotMatrix(pMat);
23
24    delete pMat;
25  }
```

- Zugriff auf Members in `plotMatrix` mit
  - Pfeiloperator ->

Referenzen

- Definition
- Unterschied zw Referenz und Pointer

Was ist eine Referenz

- Referenzen sind Aliasnamen
- Erzeugung mittels (&)
  - nicht verwechseln mit Adressoperator
- `type& refName = varName`
  - Erzeugt eine Referenz `refName`
  - `refName` ist vom Typ "Referenz auf `type`"
  - `type` ist beliebiger Datentyp
  - `varName` ist Variablenname (Typ `type`)
  - Referenz muss bei Definition initialisiert werden!
- Referenz verhält sich wie Zielobjekt

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

- Ausgabe:
```
var = 5
ref = 5
var = 7
ref = 7
```


Adressoperator bei Referenzen

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

- Ausgabe:
```
var = 5
ref = 5
Adresse von var = 0x7fff5fbffa2c
Adresse von ref = 0x7fff5fbffa2c
```

- Adressen der beiden Variablen sind identisch
- Referenzen werden bei Erzeugung initialisiert
  - dienen nur als Synonyme für ihre Ziele

Funktionsargumente als Zeiger

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

- Ausgabe:
```
x = 5, y = 10
x = 10, y = 5
```

- Dereferenzieren des Pointers um auf Inhalt der Variable zugreifen zu können (Zeile 6–8)
- Bei Aufruf der Funktion `swap` muss Adresse übergeben werden (Zeile 14)

Funktionsargumente als Referenz

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

- Ausgabe:
```
x = 5, y = 10
x = 10, y = 5
```

- Syntax: `rtype funName(type& refName)`
  - `funName` ist Fkt mit `rtype` als Rückgabetyp
  - Aufruf der Funktion mit Variablen vom Typ `type` (vgl. Zeile 14)
  - Innerhalb Funktion `funName` ist `refName` Referenz
    - Änderungen innerhalb der Funktion an `refName` wirken sich auch außerhalb aus
      ⇒ Call by Reference
- Referenzen können auch Rückgabetyp sein
  - Achtung: Keine Ref auf lokale Var zurückgeben

Referenzen vs. Pointer

- Refenzen sind Synonyme für Variablen
- Müssen direkt bei Definition initialisiert werden
  - mit &
  - nicht mit Adressoperator verwechseln
- Man kann Referenzen nicht nachträglich zuordnen!
- Wie bei Zeigern auf Lebensdauer achten:

```
1  int& f() {
2     int x = 4711;
3
4     // Achtung: Referenz auf lokale Variable
5     return x;
6  }
```

- Syntax kann Programmablauf verschleiern
  - Bei Aufruf nicht klar ob Call by Reference oder Call by Value
  - Passiert bei Pointer nicht
  - Innerhalb Funktion analog
- Keine vollständige Alternative zu Pointern
  - Keine Mehrfachzuweisung
  - Referenzen dürfen nicht 0 (`NULL`) sein
  - Kein dynamischer Speicher möglich
  - Keine Felder von Referenzen möglich

Read-Only Referenzen

- `const type& refName = varName`
  - Schlüßelwort `const` definiert konstante Referenz
  - `varName` ist vom Typ `type`
  - `refName` hat Typ "konstante Referenz auf `type`"
  - Zugriff auf Referenz nur **lesend** möglich
  - ```
    int x = 5;
    const int& ref = x;
    x = 5; // ok!
    ref = 6; // Geht nicht! Nur Lesend
    ```
- Read-Only Referenzen auch als Funktionsparameter möglich
  - Innerhalb der Funktion darf nur lesend darauf zugegriffen werden
- Auch als Ergebnistyp der Rückgabe möglich (vgl. Überladen)
- Da nur lesender Zugriff möglich
  ⇒ Zugriffskontrolle
  - Falls versucht wird eine konstante Ref zu ändern
    ⇒ Fehlermeldung von Compiler

Überladen

- Überladen von Funktionen
- Überladen von Operatoren
- Dreierregel
- operator

Überladen von Funktionen 1/2

- Mehrere Funktionen gleichen Namens möglich
  - Wie bei Konstruktoren
  - Unterscheiden sich durch ihre Signatur
- Diesen Vorgang nennt man überladen
- Durch Aufruf wird die richtige ausgewählt
  - Compiler erkennt dies über Signatur
- Rückgabewerte können unterschiedlich sein
  - Dann müssen sich aber auch Eingabeparameter unterscheiden
  - Also: unterschiedliche Rückgabeparameter und gleiche Eingabeparameter geht nicht
    ```
    int pow(int x);
    double pow(double x);
    ```
    Obiger Code ist in Ordnung während folgender zu einem Compilerfehler führt
    ```
    int pow(int x);
    double pow(int x);
    ```
    ```

Überladen von Funktionen 2/2

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

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

Überladen von Operatoren 1/5

- Ziel: Klasse zum Verwalten komplexer Zahlen
  - Addition, Multiplikation, ... bereitstellen
  - Operatoren `+,-,*,/`

```
1   #include <iostream>
2   using namespace std;
3
4   class Complex {
5     double re;
6     double im;
7   public:
8     Complex(double r, double i) {re = r; im = i;}
9     double getRe() const {return re;}
10    double getIm() const {return im;}
11  };
12
13  const Complex operator+(const Complex& lhs,
14                          const Complex& rhs) {
15    Complex tmp(lhs.getRe()+rhs.getRe(),
16                lhs.getIm()+rhs.getIm());
17    return tmp;
18  }
19
20  int main() {
21    Complex a(1,0), b(0,1);
22    Complex c = a+b;
23    cout << c.getRe() << " + i*" << c.getIm()
24         << endl;
25  }
```

- Ausgabe: `1 + i*1`
- Schlüsselwort `operator`

Überladen von Operatoren 2/5

- `+,-,*,/` sind binäre Operatoren
  - Haben also 2 Operanden (links u. rechts davon)
- Definition von Op außerhalb Klassendef
- `const Class operator+(const Class& lhs, const Class& rhs)`
  - `Class` ist Name der Klasse
  - Übergabeparameter sind Read-Only Referenzen
    - Call by Reference!
  - Rückgabewert ist vom Typ `const Class`
    - `const` wird verwendet um Sinnloses(!) wie `(a+b)=c` zu vermeiden.
  - Ergebnis von `lhs + rhs` wird zurückgegeben
  - Analog für `-,*,/`
- Nachgestelltes `const` bei Klassenmethode gibt an, dass Methode Memberdaten nicht ändert
- `rtype funName(...) const`
  - `rtype` Rückgabetyp von Methode `funName`
  - `...` steht für Liste von Eingabeparametern
  - `const` bewirkt dass `funName` nur lesend auf Members zugreifen darf
    - Ansonsten: Fehlermeldung von Compiler

Überladen von Operatoren 3/5

- Ziel: Unäre Operatoren für Vorzeichenwechsel (-) und komplexe Konjugation (~)

```
1   #include <iostream>
2   using namespace std;
3
4   class Complex {
5     double re,im;
6   public:
7     Complex(double r, double i) {re = r; im = i;}
8     double getRe() const {return re;}
9     double getIm() const {return im;}
10    const Complex operator-() const;
11    const Complex operator~() const;
12  };
13
14  const Complex Complex::operator-() const {
15    return Complex(-this->getRe(),-this->getIm());
16  }
17
18  const Complex Complex::operator~() const {
19    return Complex(this->getRe(),-this->getIm());
20  }
21
22  int main() {
23    Complex a(1,-2);
24    Complex b = -a;
25    cout << b.getRe() << " + i*" << b.getIm()
26         << endl;
27    a = ~b;
28    cout << a.getRe() << " + i*" << a.getIm();
29         << endl;
30  }
```

- Ausgabe: `-1 + i*2`
  `-1 + i*-2`

Überladen von Operatoren 4/5

- Unäre Operatoren `-,~` können überladen werden
  - Haben 1 Operanden: `-a`, `~a` (rechts davon)
- `const Class operator-() const`
  - Wird als Membermethode def (also in Klasse!)
  - Erzeugt Kopie von Objekt und wechselt Vz
  - Zurückgegeben wird neu erstelltes Objekt
  - Nachgestelltes `const` damit nur lesend auf Obj zugegriffen wird
  - Rückgabewert wie vorher vom Typ `const Class`
  - `this`-Pointer vorhanden
- Man kann auch eigene Typecasts def
- `operator type() const`
  - Wird als Membermethode einer Klasse `Class` def
  - Konvertiert Variable vom Typ `Class` zu Typ `type`
  - Kein Rückgabetyp, da nur Typ `type` möglich
  - "Nur" Richtung `Class` $\longrightarrow$ `type` möglich
- Will man Konvertierung vom Typ `type` nach `Class`, dann kann man das mit Konstr realisieren

Überladen von Operatoren 5/5

- Ziel: Typkonvertierung von `double` auf `Complex`
  Typkonvertierung von `Complex` auf `double`

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  class Complex {
 5    double re,im;
 6  public:
 7    Complex(double r, double i) {re=r; im=i;}
 8    double getRe() const { return re;}
 9    double getIm() const { return im;}
10    // Folgender Konstr fuer double --> Complex
11    Complex(double r) {re = r; im = 0;};
12    // Folgender Op fuer Complex --> double
13    operator double() const;
14  };
15
16  Complex::operator double() const {
17    return re;
18  }
19
20  int main() {
21    Complex a(1,0);
22    double x = 5;
23    cout << "x = " << x << endl;
24    x = a;
25    cout << "x = " << x << endl;
26    a = 10.0;
27    cout << "a = " << a.getRe() << " + i*"
28                    << a.getIm() << endl;
29  }
```

- Ausgabe:
```
x = 5
x = 1
a = 10 + i*0
```


Was kann überladen werden

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

- Unäre und binäre Operatoren
- Unterscheide zwischen Postfix und Präfix
  - Leicht unterschiedliche Syntax
- Ein Op kann öfters überladen werden
  - Müssen sich in Signatur unterscheiden
  - `const Complex operator+(const Complex& lhs, double x)`
    Addition komplexe Zahl `lhs` + reelle Zahl `x`
- Typkonvertierungen (Casts)
- Was **nicht** überladen werden kann
  - `.,::,sizeof,.*`
- Man kann keine neuen Operatoren definieren
- Mehr Informationen findet man bei
  - `http://www.c-plusplus.de/forum/232010-full`
  - Dirk Louis: *Schnellübersicht C / C++*

Dreierregel

- Die Dreierregel ("goldene Regel") besagt:
  Existiert eine der folgenden Membermethoden, dann sollen (müssen) auch die anderen 2 existieren
  - Destruktor
  - Kopierkonstruktor
  - Zuweisungsoperator
- Kopierkonstruktor: `Class(const Class& name)`
  - Spezieller Konstruktor
  - Neues Obj ist Kopie von `name`
  - Wird aufgerufen durch
    - `Class obj(name);` oder
    - `Class obj = name;`
- Zuweisungsoperator hat Syntax
  `Class& operator=(const Class& rhs)`
  - Ist Membermethode (gehört also zur Klasse)
  - **Kein** neues Obj erstellt, daher Rückgabe als Ref
  - Ist Kopie von `rhs`
    - Soll **Deep Copy** realisieren
  - Wird aufgerufen bei Zuweisungen, z.B. `a = rhs;`
- Kein Zuweisungoperator def $\Rightarrow$ **Shallow Copy**
  - d.h. Nur oberste Ebene wird kopiert,
  - Ok, falls "einfache" Member (wie `int,double`)
  - Ok, falls Member Zuweisungsop. besitzen
  - Nicht Ok wenn Speicher allokiert, z.B. Array
- Gleiches gilt für Kopierkonstr

MyVector Klasse 1/2

- Ziel: Klasse `MyVector` erweitern
  - Destruktor wurde definiert
  - Nach Dreierregel sollte daher auch der
    - Kopierkonstruktor und
    - Zuweisungsoperator definiert werden
  - `[]`-Operator, um auf Einträge zuzugreifen

```
1  #ifndef _MYVECTOR_HPP
2  #define _MYVECTOR_HPP
3
4  #include <cstdlib>   // malloc, realloc, free
5  #include <iostream>
6  using namespace std;
7
8  class MyVector {
9    double * entries;
10   int size;
11 public:
12   MyVector() { size = 0; entries = NULL;};
13   MyVector(int size, double init = 0);
14   MyVector(const MyVector& rhs); // Kopierkonstr
15   ~MyVector();
16   MyVector& operator=(const MyVector& rhs); //Zuw
17   double& operator[](int j);
18   const double& operator[](int j) const;
19   int getSize() const { return size;}
20   void print() const {for(int j=0; j<size; ++j)
21                       cout << entries[j] << endl;}
22 };
23 #endif
```


MyVector Klasse 2/2

```
1  #include "MyVector.hpp"
2  MyVector::MyVector(int size, double init) {
3    this->size = size;
4    entries = (double*)malloc(size*sizeof(double));
5    for(int j=0; j<size; ++j)
6      entries[j] = init;
7  }
8
9  MyVector::MyVector(const MyVector& rhs) {
10   size = rhs.getSize();
11   entries = (double*)malloc(size*sizeof(double));
12   for(int j=0; j<size; ++j)
13     entries[j] = rhs[j];
14 }
15
16 MyVector::~MyVector() {
17   if(size>0)
18     free(entries);
19 }
20
21 MyVector&
22   MyVector::operator=(const MyVector& rhs) {
23   if(size==rhs.getSize()) {
24     for(int j=0; j<size; ++j)
25       entries[j] = rhs[j];
26   }
27   return *this;
28 }
29
30 double& MyVector::operator[](int j) {
31   return entries[j];
32 }
33
34 const double& MyVector::operator[](int j) const {
35   return entries[j];
36 }
```


Abschließende Bemerkungen

- Manche Op werden außerhalb von Klassen def
- Manche Op außer- od innerhalb von Klassen mögl
  - Unterschiedliche Bedeutungen
- Manche Op kann man nur in Klassen def
- `this`-Zeiger vorhanden falls Op in Klasse def
  - Zeigt bei binärem Op auf linken Operanden
- Dreierregel ist essentiell
  - Läßt man Kopierkonstr und Zuweisungsoperator in `MyVector`-Klasse weg:
    ⇒ Segmentation fault!
- 2 Überladungen des `[]`-Operators in `MyVector`
  - Version mit `const`: Für lesenden Zugriff auf Ele
    - Bei Kopierkonstr für Read-Only Ref `rhs` nötig
    - Analog bei Zuweisungsoperator
    - z.B.: `cout << a[j];` (lesend)
  - Version ohne `const`: Für Elementzuweisung
    - z.B.: `a[j] = 5;`

Matrixklasse mit `vector` 1/2

- Header Datei:

```
1  #ifndef _MATRIX_H
2  #define _MATRIX_H
3
4  #include <vector>
5  #include <iostream>
6  using namespace std;
7
8  class Matrix {
9  private:
10   int m,n; //dimensions
11   vector<vector<double> > entries;
12 public:
13   int getM(){return m;};
14   int getN(){return n;};
15   double getEntry(int i, int j);
16   void setEntry(int i, int j, double entry);
17   Matrix();
18   Matrix(int m, int n, double init);
19 };
20 #endif // _MATRIX_H
```

- Geschachtelte Verwendung von **vector**
- `vector<vector<type> > name`
  - def Matrix `name` mit Einträgen vom Typ `type`
  - **ACHTUNG** auf Leerzeichen bei > >
    - Ansonsten Verwechslung mit Shiftoperator >>
- `name = vector<vector<type> >(m,vector<type>(n))`
  - weist Variable `name` eine $m \times n$-Matrix zu
    - d.h. Speicher wird automatisch allokiert
    - Einträge können auch initialisiert werden (nächste Folie)

Matrixklasse mit `vector` 2/2

- Implementierung der Klassenmethoden:

```
 1  #include "matrix.h"
 2
 3  double Matrix::getEntry(int i, int j){
 4    if(i < getM() && j < getN()){
 5      return entries[i][j];
 6    }
 7    else {
 8      cout << "Index exceeds matrix dimensions\n";
 9    }
10  }
11
12  void Matrix::setEntry(int i,int j,double entry){
13    if (i >= m || j >= n){
14      cout << "Index exceeds matrix dimensions\n";
15    }
16    else {
17      entries[i][j] = entry;
18    }
19  }
20
21  Matrix::Matrix() : m(1), n(1),
22    entries(vector<vector<double> >(m,
23          vector<double>(n, 0))){}
24
25  Matrix::Matrix(int m,int n,double init): m(m),
26    n(n), entries(vector<vector<double> >(m,
27          vector<double>(n, init))){}
```

- Vorteile der Klasse:
  - Kein (Kopier-)/Konstruktor, Destruktor, oder Zuweisungsoperator nötig
  - Automatische Speicherverwaltung bei Verwendung von `vector`

Testen

- Motivation
- Qualitätssicherung
- Arten von Tests
- Exception Handling

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

Motivation

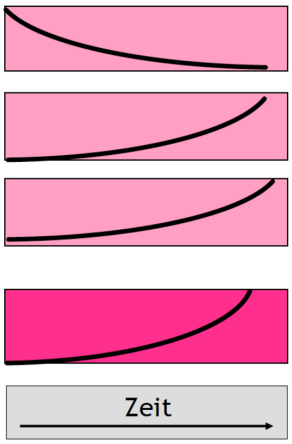
- Ariane 5 Explosion ('96)
  - Konversion `double` → `int`
  - Schaden ca 850 Mio. EUR
- Patriot Missile Fehler, Golfkrieg ('91)
  - Zeitmessung falsch berechnet & Rundungsfehler

Qualitätssicherung

- Warum wir analysieren müssen
  - Software entsteht durch menschliche Hand
  - Software wird Fehler enthalten
  - Fehler zu machen ist menschlich
  - Fehler finden bevor sie Schaden anrichten (Leben kosten)
- Wie intensiv müssen wir analysieren?
  - Wie kann man gründlich suchen?
  - Wie lange muss man suchen?
- Was ist die Aufgabe der Analyse?
  - Nur das Aufspüren von Fehlern
  - Debugging ist eine andere Disziplin
- Mögliches Spektrum von Fehlern
  - nicht tolerierbar (können Schaden anrichten)
  - ungewollt (Kunden könnten verloren gehen)
  - nicht dramatisch

Wert & Kosten des Testens

- Entdeckungsrate
- Aufwand der Tests
- Fehlerqualität
- Kosten pro Fehler

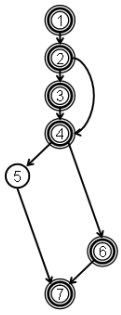


- Kosten steigen überproportional
  - Aufhören wenn Testen zu schwierig/teuer wird?
  - unsystematische Methoden sind inakzeptabel (kein ad-hoc testing - 'hier und da')
- Notwendigkeit für Systematik
  - Strukturelle Methoden
  - Funktionale Methoden

Strukturelle Tests

- Basieren auf der Programmstruktur
  - White-Box Tests
- Programm wird gegen sich selbst getestet
  - Testfälle werden anhand der Struktur generiert
- Spezifikation wird bei Beurteilung nicht beachtet

```
public int bonus(int x, int y){
    int ergebnis=0;
    if (x>0 && y>0)
        ergebnis=30;
    if (x>10 && y<10)
        ergebnis+=20;
    else
        ergebnis+=10;
    return ergebnis;
}
```



- Kontrollflussgraph
- stellt Programmfluss grafisch dar
- Knoten = Anweisung
- Kante beschreibt möglichen Kontrollfluss zwischen Anweisungen

Überdeckungstest

- Nutzt Strukturen zur Definition von Testzielen
  - Anweisungen, Zweige, Bedingungen
- Anweisungsüberdeckung (C0)
  - Alle Anweisungen werden ausgeführt
  - Hilft, toten Code zu finden
  - notwendig aber nicht ausreichend (wenig relevant)
  - wird C0-Test genannt (C=Coverage)
- Zweigüberdeckung (C1)
  - Alle Zweige des Graphen werden durchlaufen
  - Alle Entscheidungen werden *wahr* und *falsch*
  - Anweisungsüberdeckung vollständig enthalten
  - berücksichtigt keine Abhängigkeiten
  - unzureichend für komplexe Bedingungen
  - minimales Testkriterium
- Pfadüberdeckung (C7)
  - Alle Kombinationen von Zweigen abdecken
  - Zweigüberdeckung enthalten
  - unfassbar viele Testfälle
  - Höchste Erfolgsaussichten, völlig unpraktikabel

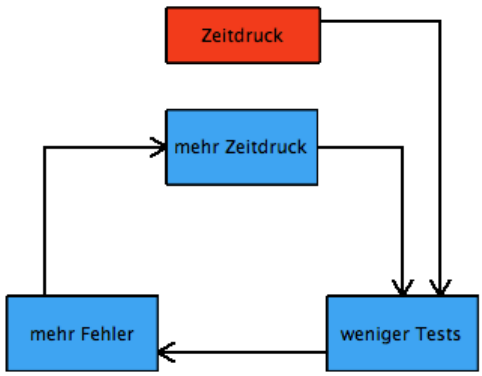
Funktionales Testen

- Programm wird gegen Spezifikation geprüft
  - Black-box Test
  - Tut das Programm das Richtige? (nur gegen sich selbst testen ist unzureichend)
- Ziel: effiziente Prüfung der Funktionalität
- Problem: Ableitung geeigneter Testfälle
  - Testfälle repräsentativ?
- Testfallbestimmung:
  - Funktionale Äquivalenzklassen
  - Grenzwertanalyse
- Motivation: Testfälle sinnvoll einschränken
- Reaktion für jeden Repräsentanten gleich
- Fehler typischerweise bei Grenzfällen
  - extreme Repräsentanten besonders geeignet

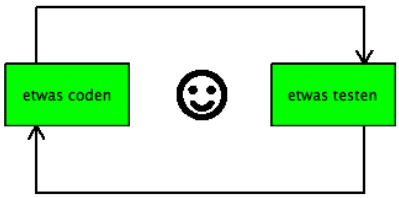
Testklassifizierung

- Strukturelles Testen
  - Abbruchfehler
  - unerreichbare Zweige
  - Endlosschleifen
  - inkonsistente Bedingungen
- Funktionales Testen
  - Falsche Antworten
- kombinierte Strategie ist nötig
- Teststadien:
- Unit Test
  - Test von Komponenten (Methoden, Klassen) → strukturelles Testen
- Integrationstest
  - (Zusammenwirken) → strukturelles Testen
- Systemtest
- (ganzes System gegen Anforderung testen) → funktionales testen
- Abnahmetest
  - Test mit echten Kundendaten

Ein Teufelskreis

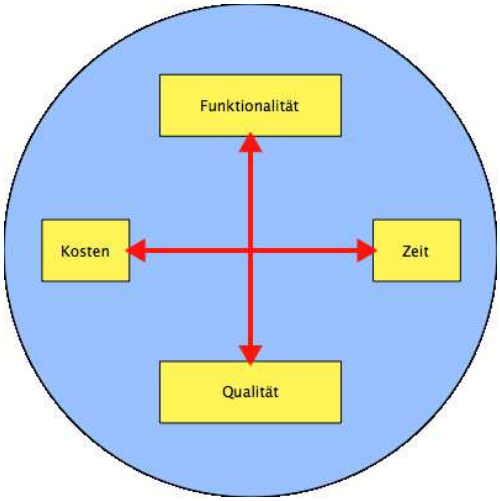


- testen ist unverzichtbar
- wird immer teurer und stressiger
- → iterativer Prozess



Qualitätssicherung

- Softwareentwicklung hat vier Fakroten
- Kunde darf drei priorisieren
- vierte Faktor ergibt sich daraus
- Qualität sollte immer gewahrt bleiben!



Fehler erkennen und behandeln

- Viele Programme sind fehlerhaft
  - u.a. auch kommerzielle Software
- Großteil der Entwicklung geht in Fehlersuche
  - ⇒ Kosten
- Fehler lassen sich in Kategorien einteilen
  - syntaktische Fehler ⇒ 'leicht'
  - logische Fehler ⇒ 'schwerer'
  - Ausnahmen (Exceptions)
- Die ersten beiden kann man ausmerzen
- Die letzte muss man handhaben

```
 1  int main() {
 2    double a, b;
 3    cout << "Zahl 1: ";
 4    cin >> a;
 5    cout << "Zahl 2: ";
 6    cin >> b;
 7    cout << "Produkt = " << a*b << endl;
 8
 9    return 0;
10  }
```

- Was passiert wenn keine Zahl eingegeben?
- Können Sie für den Benutzer garantieren?

Exception handling 1/3

- Was tun bei falscher Benutzereingabe
  - Programm abstürzen lassen?
  - Benutzer informieren und Programm beenden?
  - Eingreifen ohne Benutzer zu informieren?
  - Dem Benutzer helfen den Fehler zu korrigieren?
- Exceptions sind Ausnahmezustände
- Gründe für die Entstehung
  - Benutzereingabe
  - Kein Speicher mehr verfügbar
  - Dateizugriffsfehler
  - Division durch 0
- Wir können diese 'Fehler' nicht verhindern
  - Wir können sie aber antizipieren
  - An zentraler Stelle behandeln
- Programmabsturz ist keine Option!

Exception handling 2/3

- Konzept des Exception Handling
  - Trennung von normalen Programmfluss und Behandlung von Fehlern
  - Fehler die in einem Teil auftauchen, werden zentral von Aufrufumgebung behandelt
  - Keine ständige Kontrolle ob Fehler aufgetreten
  - Fehler gehen automatisch zur Aufrufumgebung
- Syntax in C++:
- `try{...}`
  - Leitet risikobehafteten Code ein (umschließt ihn)
  - Hier können exceptions geworfen werden
- `throw name`
  - Hier wird eine exception ausgelöst (geworfen)
  - `name` ist Variable vom Typ `type`
    - Ist Obj (beliebiger Datentyp)
- `catch(type name)`
  - Hier wird auf exceptions reagiert (exception handler)
  - `type` ist beliebiger Datentyp
    - `name` kann manchmal weggelassen werden

Beispiel

- Ziel: 'Standardprobleme' abfangen

```
1   #include <iostream>
2   using namespace std;
3
4   double divide(double valA, double valB){
5     if(valB == 0) {
6       throw int(0);
7     }
8     return valA / valB;
9   }
10
11  int main() {
12    double result;
13    try{
14      result = divide(5, 0);
15    }
16    catch(const int& error){
17      cout << "Division durch 0" << endl;
18    }
19  }
```

- Z.6: `int(0)` erzeugt Integer-obj mit Wert 0
  - Wird zurückgeworfen
- `catch`-Umgebung fängt Exception von Typ `int` auf
- Alternative mittels `if - else`
  - Mühsamer
  - Nicht immer sinnvoll möglich

Wissenswertes

- Exceptions sind Objekte
  - Haben einen Datentyp (kann manchmal weggelassen werden)
  - Instanzen jeder Klasse können geworfen werden
- Typ gibt Aufschluss über Fehlerart
  - oftmals: Ausnahmehierarchien
- Schachtelung von `try-catch`-Blöcken möglich
- Fehlerklassen
  - Für verschiedene Fehler werden eigene Fehlerklassen def
  - Müssen weder Daten noch Methoden haben
  - Können zusätzliche Informationen zu Fehler haben
    - z.B. genauere Fehlerursache
    - daher besser als "nur" `int`-Obj zurückgeben
- Exceptions müssen nicht direkt behandelt werden
  - gehen zurück an aufrufende Funktion (stack unwinding)

Beispiel 1/3

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   class Error {  // Eigene Fehlerklasse
6     string errMsg; // Enthaelt Fehlerinformation
7   public:
8     Error(string msg) : errMsg(msg) {};
9     const string getMsg() const {return errMsg;}
10  };
11
12  double divide(double valA, double valB){
13    if(valB == 0) {
14      throw Error("Division durch 0");
15    }
16    return valA / valB;
17  }
18
19  int main() {
20    double result;
21    try{
22      result = divide(5, 0);
23    }
24    catch(const Error& error){
25      cout << error.getMsg() << endl;
26    }
27  }
```

- Eigene Fehlerklasse `Error`
  - Besitzt `string` für Fehlerinformation
  - Wird in `catch`-Umgebung ausgegeben
  - Kann für unterschiedliche Fehler benutzt werden

Beispiel 2/3

- Ziel: Fehlerbehandlung für `MyVector`-Klasse
  - Kontrollierter Indexzugriff
- Erweitere Header-Datei der `MyVector`-Klasse um eigene Fehlerklasse `IdxError`:

```
1   class IdxError { // Fehlerklasse
2     ; // Ohne Methoden oder Daten!
3   };
```

- Ändere Implementierung des []-Operator in `MyVector.cpp`

```
1   double& MyVector::operator[](int j) {
2     if(j<0 || j>=getSize())
3       throw IdxError();
4     return entries[j];
5   }
6
7   const double& MyVector::operator[](int j) const {
8     if(j<0 || j>=getSize())
9       throw IdxError();
10    return entries[j];
11  }
```


Beispiel 3/3

```
1   #include "MyVector.hpp"
2
3   int main() {
4     int j;
5     MyVector a(10,1);
6
7     cout << "Index j = ";
8     cin >> j;
9
10    try {
11      cout << "a[" << j << "] = "
12           << a[j] << endl;
13    }
14    catch(const IdxError&) {
15      cout << "Indexfehler" << endl;
16    }
17  }
```

- Bei Eingabe von 2
  - Ausgabe: `a[2] = 1`
- Bei Eingabe von -1
  - Ausgabe: `Indexfehler`
- In Z.14: Nur Typ der Exception angegeben
  - Name des Parameters kann weggelassen werden, falls nicht benötigt
- Fehlerabfrage in `MyVector`-Klasse sinnvoll?
  - z.B. für große Vektoren (viele Zugriffe)?

Was ist zu beachten?

- Exceptions ohne `catch`
  - Führen in der Regel zu Programmabbruch
  - Werden immer weiter zurückgegeben
  - Am Ende wird `terminate()` aufgerufen (kann auch bearbeitet werden)
- Mehrere `catch`-Blöcke hintereinander
  - Abfangen von verschiedenen Fehlertypen
  - Verwende `catch(...)` falls Fehlertyp unbekannt
    - z.B. falls externer Code benutzt
- Wieviel `try`, `throw` und `catch`?
  - Nicht pauschal zu beantworten
  - Kein 'globaler' `try` Block um ganzen Code
  - Code muss lesbar bleiben
  - Gefährliche Stellen und Standardfehler
  - So selten wie möglich, so oft wie nötig
- Es gibt auch Standardexceptions
  - `bad_alloc`, `bad_cast`, `bad_typeid` usw.
  - Verhindern etwa Programmabsturz wenn Arbeitsspeicher oder Festplatte voll

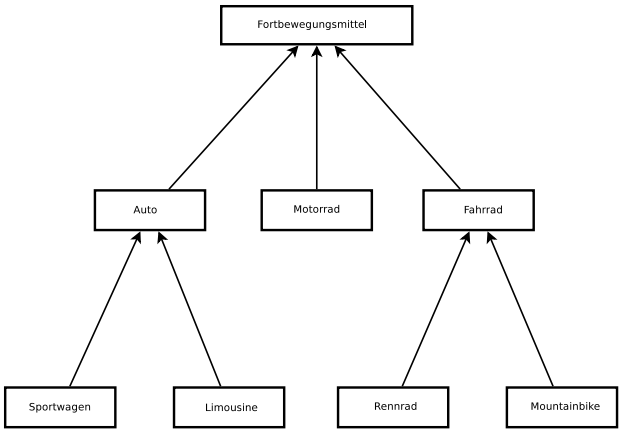
Vererbung

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

Was ist Vererbung 1/2

- Im Alltag klassifizieren wir Objekte
- Wir teilen unsere Umgebung in Kategorien ein
  - Ein Auto ist ein Fortbewegungsmittel
  - Ein Hund ist ein Säugetier
  - Eine quadratische Matrix ist eine Matrix
- Alle Säugetiere können sich bewegen und atmen
  - Hunde haben zusätzlich spezielle Eigenschaften ... bellen, mit dem Schwanz wedeln ...
- Die Kategorien lassen sich weiter unterteilen
  - Sportwagen und Limousine beschreiben spezielle Autos
- Ganze Hierarchien entstehen
  - Unterkategorien sind jeweils Erweiterungen
  - Eigenschaften der Basiskategorie haben alle
    $\Rightarrow$ ist-ein-Beziehung
- C++ stellt Kategorien durch Klassen dar
- Spezielle werden von allgemeineren abgeleitet
- Diesen Vorgang nennt man Vererbung

Was ist Vererbung 2/2



Die Syntax der Vererbung

- Vererbung in C++ bildet Wirklichkeit ab
  - Spezialisierung durch zusätzliche Methoden
  - Allgemeine Methoden sind direkt verfügbar
  - Fähigkeit bewegen muss bei Auto nicht separat implementiert werden
    $\Rightarrow$ das wurde von *Fortbewegungsmittel* geerbt
- Beispiel:

```
class Auto : public FortbewegungsMittel
```

- Ableitung erfolgt durch (:)
- `Auto` erbt alle Methoden und Felder von `FortbewegungsMittel`
  - `Auto` ist eine abgeleitete Klasse
  - `FortbewegungsMittel` ist die Basisklasse
- `class Abgeleitet : public BasisKlasse`
  - `Abgeleitet` ist Name der abgeleiteten Klasse
  - `BasisKlasse` ist Name der Basisklasse
  - `public` gibt Art der Vererbung an
    - vgl. später

Ein Beispiel 1/2

- FortbewegungsMittel.hpp

```
1  #ifndef _FORTBEWEGUNGSMITTEL_HPP
2  #define _FORTBEWEGUNGSMITTEL_HPP
3
4  #include <string>
5  using namespace std;
6
7  class FortbewegungsMittel {
8    double speed;
9  public:
10   FortbewegungsMittel() {};
11   FortbewegungsMittel(double s) { speed = s; }
12   double getSpeed() const { return speed; }
13   double setSpeed(double s) { speed = s; }
14   void bewegen();
15 };
16
17 class Auto : public FortbewegungsMittel {
18   string farbe; // Zusaetzliche Eigenschaft
19 public:
20   Auto() {};
21   Auto(double s, string f) { setSpeed(s);
22                              farbe = f; }
23   string getFarbe() const { return farbe; }
24   void setFarbe(string f) { farbe = f; }
25   void schalten(); // Zusaetzliche Faehigkeit
26 };
27
28 #endif // _FORTBEWEGUNGSMITTEL_HPP
```


Ein Beispiel 2/2

- FortbewegungsMittel.cpp

```
1  #include "FortbewegungsMittel.hpp"
2  #include <iostream>
3  using namespace std;
4
5  void FortbewegungsMittel::bewegen() {
6    cout << "Ich habe mich mit " << speed
7         << " km/h bewegt" << endl;
8  }
9
10 void Auto::schalten() {
11   cout << "Geschaltet" << endl;
12 }
```

- main.cpp

```
1  #include "FortbewegungsMittel.hpp"
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6    FortbewegungsMittel fahrrad(10);
7    Auto cabrio(100,"rot");
8
9    fahrrad.bewegen(); // public Meth. auch in
10   cabrio.bewegen();  // abeleit. Klasse
11
12   cabrio.schalten();
13 }
```

- Ausgabe:
```
Ich habe mich mit 10 km/h bewegt
Ich habe mich mit 100 km/h bewegt
Geschaltet
```


Die public-Vererbung

- Syntax bereits kennegelernt:
  `class Abgeleitet : public BasisKlasse`
- Öffentliche (public) Memberdaten und -methoden der Basisklasse sind in abgeleiteter Klasse
  - sichtbar
  - wieder öffentliche Klassendaten bzw. -methoden
    - also public
- Private Members (private) der Basisklasse sind in abgeleiteter Klasse
  - **nicht** sichtbar
  - Direkter Zugriff ist daher **nicht** möglich
    - Zugriff ist nur mit öffentlichen (public) Methoden möglich
    - z.B. get und set Funktionen

```
1  class Motorrad : public FortbewegungsMittel {
2  public:
3    // Motorrad() { speed = 300; }; // Nicht OK
4    Motorrad() { setSpeed(300); }; // OK
5  };
```

- Zeile 3 (ohne Kommentar) nicht OK, da versucht wird direkt auf private Daten der Basisklasse zuzugreifen (sind nicht sichtbar)
- Zeile 4 OK, da setSpeed öffentliche Methode von Basisklasse ist und daher sichtbar

Das Schlüsselwort protected 1/2

- Neben Zugriffspezifizierer public, private gibt es auch noch protected
- Problem: private Members der Basisklassen sind bei abgeleiteten Klassen nicht verfügbar
  - Deklaration als public in Basisklasse nicht sinnvoll
    - (Ungeschützter) Zugriff von "außen" wäre möglich
- Lösung: Members als protected def.
  - Oder get, set Funktionen verwenden
- protected Member der Basisklasse verhält sich wie private Members
  - Von "außen" also kein direkter Zugriff mögl
- protected Member der Basisklasse bei public-Vererbung in abgeleiteter Klasse
  - sichtbar
  - wie private Members
    - innerhalb von abgeleiteter Kl besteht Zugriff
    - außerhalb nicht
- Unterschied private vs. protected Members:
  - Bei public-Vererbung sind private Members der Basiskl in abgeleiteter Klasse nicht sichtbar

Das Schlüsselwort `protected` 2/2

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


Weitere Typen von Vererbung

- Es muss nicht immer `public` vererbt werden

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

Konstruktoren & Vererbung 1/2

- Vererbung bedeutet eine ist-ein-Beziehung
  - Jedes Auto **ist ein** Fortbewegungsmittel
- Das merkt man am Aufruf
  - Zuerst Konstruktor der Basisklasse aufgerufen
  - Danach Konstruktor der abgeleiteten Klasse
  - Bei Destruktoren umgekehrt
- Entsprechend stellen Konstruktoren oft Erweiterungen dar
  - z.B. zusätzliche Felder werden initialisiert
  - daher: Basisklassenkonstruktor aufrufen

```
1   #include <iostream>
2   using namespace std;
3
4   class Basis {
5   public:
6     Basis() { cout << "Basiskonstr.\n";}
7   };
8   class Abgelitten : public Basis {
9   public:
10    Abgelitten() { cout << "Abg.konstr.\n";}
11  };
12
13  int main() {
14    Abgelitten abg;
15  }
```

- Ausgabe: `Basiskonstr.`
  `Abg.konstr.`

Konstruktoren & Vererbung 2/2

```
1   #ifndef _FORTBEWEGUNGSMITTEL_HPP
2   #define _FORTBEWEGUNGSMITTEL_HPP
3
4   #include <string>
5   using namespace std;
6
7   class FortbewegungsMittel {
8     double speed;
9   public:
10    FortbewegungsMittel() {};
11    FortbewegungsMittel(double s) { speed = s; }
12    double getSpeed() const { return speed; }
13    double setSpeed(double s) { speed = s; }
14    void bewegen();
15  };
16
17  class Auto : public FortbewegungsMittel {
18    string farbe; // Zusaetzliche Eigenschaft
19  public:
20    Auto() {};
21    Auto(double s, string f)
22       : FortbewegungsMittel(s) { farbe = f; }
23    string getFarbe() const { return farbe; }
24    void setFarbe(string f) { farbe = f; }
25    void schalten(); // Zusaetzliche Faehigkeit
26  };
27
28  #endif // _FORTBEWEGUNGSMITTEL_HPP
```

- Zeile 21–22: Expliziter Aufruf eines Basisklassenkosntruktors

Funktionen redefinieren 1/2

- Funktionen können in der abgeleiteten Klasse komplett neu gestaltet (redefiniert) werden

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  class Basis {
 5  public:
 6    void print() { cout << "Basisklasse\n";}
 7  };
 8
 9  class Abgelitten : public Basis {
10  public:
11    // Funktion print() aus Basiskl redefiniert
12    void print() { cout << "Abgeleitete Kl\n";}
13  };
14
15  int main() {
16    Basis b;
17    Abgelitten a;
18    b.print();
19    a.print();
20  }
```

- Ausgabe: `Basisklasse`
  `Abgeleitete Kl`
- **Nicht** verwechseln mit Überladen
  - Überladen bedeutet mehrere Methoden gleichen Namens mit unterschiedlicher Signatur
- Methode `print` aus Basisklasse wird in abg. Klasse überdeckt

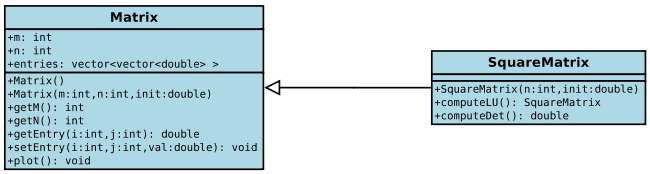
Funktionen redefinieren 2/2

- Problem: Basismethoden werden verdeckt
  - eine redefinierte Methode verdeckt alle Basismethoden

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  class Basis {
 5  public:
 6    void print() { cout << "Basisklasse\n";}
 7    void print(int s) { cout << s << endl;}
 8    void print(int s, double d)
 9      { cout << s << " " << d << endl;}
10  };
11
12  class Abgelitten : public Basis {
13  public:
14    // Funktion print() aus Basiskl redefiniert
15    void print() { cout << "Abgeleitete Kl\n";}
16  };
17
18  int main() {
19    Basis b;
20    Abgelitten a;
21    b.print(10);  // OK
22    a.print();    // OK, redef. Fkt aus abg.Kl
23    a.print(10);  // Nicht OK
24  }
```

- Kein Zugriff mehr auf überladene Basisfunktionen
- Lösung: Basisfunktion manuell aufrufen
  - vollständigen Namen verwenden
    `a.Basis::print(10);`

Vererbung mit Matrizen 1/3




```
 1  #ifndef _MATRIX_HPP
 2  #define _MATRIX_HPP
 3
 4  #include <vector>
 5  using namespace std;
 6
 7  class Matrix{
 8    int m,n; // Dimensionen
 9    vector<vector<double> > entries;
10  public:
11    int getM() const {return m;};
12    int getN() const {return n;};
13    double getEntry(int i, int j) const;
14    void setEntry(int i, int j, double entry);
15    void plot();
16    Matrix();
17    Matrix(int m, int n, double init);
18  };
19
20  class SquareMatrix : public Matrix{
21  public:
22    SquareMatrix computeLU();
23    double computeDet();
24    SquareMatrix(int n, double init)
25      : Matrix(n,n,init) {};
26  };
27  #endif // _MATRIX_HPP
```


Vererbung mit Matrizen 2/3

```
 1  #include "Matrix.hpp"
 2  #include <iostream>
 3  using namespace std;
 4
 5  double Matrix::getEntry(int i, int j) const {
 6    return entries[i][j];
 7  }
 8
 9  void Matrix::setEntry(int i, int j, double val) {
10    entries[i][j] = val;
11  }
12
13  void Matrix::plot(){
14    for(int i = 0; i < getM(); ++i)
15      for(int j = 0; j < getN(); ++j)
16        cout << "Eintrag " << i << ", " << j
17          << ": " << getEntry(i,j) << endl;
18  }
19
20  Matrix::Matrix(): m(1), n(1),
21    entries(vector<vector<double> >(m,
22          vector<double>(n, 0))){}
23
24  Matrix::Matrix(int m, int n, double init): m(m),
25    n(n), entries(vector<vector<double> >(m,
26        vector<double>(n, init))){}
27
28  SquareMatrix SquareMatrix::computeLU(){
29      // Implementierung LU-Zerlegung
30  }
31
32  double SquareMatrix::computeDet(){
33      // Determinante berechnen
34  }
```

Vererbung mit Matrizen 3/3

```
1  #include "Matrix.hpp"
2
3  int main() {
4    Matrix mat1(3,4,2);
5    mat1.plot();
6
7    SquareMatrix mat2(2,1);
8    mat2.plot();
9  }
```

- Ausgabe:
```
Eintrag 0, 0: 2
Eintrag 0, 1: 2
Eintrag 0, 2: 2
Eintrag 0, 3: 2
Eintrag 1, 0: 2
Eintrag 1, 1: 2
Eintrag 1, 2: 2
Eintrag 1, 3: 2
Eintrag 2, 0: 2
Eintrag 2, 1: 2
Eintrag 2, 2: 2
Eintrag 2, 3: 2
Eintrag 0, 0: 1
Eintrag 0, 1: 1
Eintrag 1, 0: 1
Eintrag 1, 1: 1
```


Übersicht – Vererbung

- Vererbung macht den Code leichter zu warten
- Verebung erhöht die Wiederverwendbarkeit massiv
- Abgeleitete Klasse erbt alle Methoden und Felder
  - sofern der Zugriff dies erlaubt
- Neue Implementierung für geerbte Methoden möglich
  - Methoden redefinieren
  - Achtung:
    - verdeckt alle gleichnamigen Basismethoden (dies kann natürlich gewollt sein)
    - Zugriff über vollen Funktionsnamen
- Fazit:
  - verwende Vererbung immer wenn (sinnvoll) möglich
  - Wiederverwendung von Code wenn möglich
  - Senkt die Fehlerquote
  - Erleichtert spätere Änderungen

Vererbung 2

- Polymorphie
- Virtuelle Methoden
- Abstrakte Klassen
- Mehrfachvererbung

- 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
  - Obj abgeleiteter Klassen haben mindestens zwei (Typ der abg Kl und Typ der Basiskl)
- In C++ kann der jeweils passende Typ verwendet werden
  - Diese Eigenschaft nennt man Polymorphie (*griech.* Vielgestaltigkeit)

virtual 1/2

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

- Ausgabe: `Abgeleitete Kl`
  `Basisklasse`
- Zeile 17: Objekt `a` vom Typ `Abgelitten` **ist** auch vom Typ `Basis`
  - Pointer auf Typ `Basis` mit Adresse von `a` möglich
  - **Problem:** Redefinierte Methode `print` wird in Zeile 20 **nicht** aufgerufen
    - Es wird Implementierung aus `Basis` verwendet
    - Das ist nicht gewollt

virtual 2/2

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

- Ausgabe: `Abgeleitete Kl`
  `Abgeleitete Kl`
- Schlüßelwort `virtual`
  - vor Signatur der Methode `print` (in Basisklasse!)
  - deklariert virtuelle Methode
  - in Zeile 20 wird redefinierte Methode `print` aus der abgeleiteten Klasse `Abgelitten` aufgerufen
  - dies passiert während Laufzeit
    - durch sog. **V-tables**

V-table

- Tabelle virtueller Methoden
  - ist *look-up* Tabelle
- Für jede Klasse wird eine Tabelle ihrer virtuellen Methoden angelegt
- Ermöglicht **dynamische Bindung** (late binding)
  - bei **Laufzeit** wird für jede virtuelle Methode in V-table nachgeschlagen
  - V-table enthält Pointer zu virtueller Methode
- Statische Bindung (early binding)
  - Gegenteil zu dynamischer Bindung
  - Bei Kompilierung wird Aufruf der Meth. fixiert
- `virtual rType method(varlist)`
  - erzeugt **virtuelle** Methode `method`
    - mit Rückgabewert `rType`
    - `varlist` ist Liste von Eingabeparametern
  - wird in Basisklasse deklariert

Beispiel 1/2

- Ziel: Klasse für geoemtrische Figuren in 2D
  - Davon Klasse für Kreis, Dreieck ableiten
  - Methode um Figur zu verschieben

```
1   class Figur {
2     double schwerPunkt[2];
3   public:
4     virtual void verschiebe(double x, double y) {
5       schwerPunkt[0] += x; schwerPunkt[1] +=y;
6     }
7   };
8
9   class Kreis : public Figur {
10    double radius;
11  };
12
13  class Dreieck : public Figur {
14    double a[2],b[2],c[2];
15  public:
16    void verschiebe(double x, double y) {
17      Figur::verschiebe(x,y);
18      a[0] += x; b[0] += x; c[0] += x;
19      a[1] += y; b[1] += y; c[1] += y;
20    }
21  };
```

- Klasse `Kreis` enthält zusätzlichen Member `radius`
  - Methode `verschiebe` muss **nicht** redef. werden
- Klasse `Dreieck` mit Eckpunkten `a`, `b`, `c`
  - Methode `verschiebe` wird redefiniert

Beispiel 2/2

- Ziel: Liste von verschiedenen geometrischen Obj
  - gleichzeitiges Verschieben
    - wird durch virtuelle Methode ermöglicht

```
1   #include "Figur.hpp"
2
3   int main() {
4     Figur *fig[2];
5
6     fig[0] = new Kreis(0.0,0.0,1.0);
7     fig[1] = new Dreieck(0.0,0.0,1.0,0.0,0.0,1.0);
8
9     for(int j=0; j<2; ++j)
10      fig[j]->verschiebe(1.0,0.0);
11
12    for(int j=0; j<2; ++j)
13      delete fig[j];
14  }
```

- In Zeile 4: Array von Pointer(!) auf Typ `Figur`
- Zeile 6–7: Objekte vom Typ `Kreis` bzw. `Dreieck` angelegt
  - Konstr werden aufgerufen (Initialisierung)
    - Implementierung nicht gezeigt (⇒ Übung)
- Zeile 9–10: Obj werden verschoben
- Zeile 12–13: Freigabe der Objekte

Virtueller Destruktor

```
1   #include <iostream>
2   using namespace std;
3
4   class A {
5   public:
6     virtual ~A() { cout << "Destr. A" << endl;}
7   };
8
9   class B : public A {
10  public:
11    ~B() { cout << "Destr. B" << endl;}
12  };
13
14  int main() {
15    A* var = new B;
16    delete var;
17  }
```

- Ausgabe: `Destr. B`
  `Destr. A`

- **Ohne** `virtual`
  Ausgabe: `Destr. A`
  - d.h. Destruktor von Klasse `B` wird nicht aufgerufen
- Destruktoren werden üblicherweise als virtual deklariert
  - Aufruf von Destruktor der abgeleiteten Klasse
  - Daher auch Aufräumarbeiten der abg. Klasse
    - z.B. Speicherfreigabe von in abg. Klasse allokierten Speichers

Andere Aspekte der Polymorphie

- Polymorphes Verhalten ermöglicht Behandlung des abgeleiteten Objektes wie ein Basisobjekt
  - Compiler betrachtet das Objekt als Basisobjekt (*Slicing*)
  - Zugriff auf zusätzliche Methoden nicht möglich
  - Zugriff meist nicht sinnvoll
- Problem: was wenn man doch Zugriff braucht?
- Lösung: `dynamic_cast`

```
FortbewegungsMittel* A = new Auto();
Auto* myCar = dynamic_cast <Auto*> (A);
myCar->schalten(); // Methode schalten nur in
                   // Klasse Auto def
```

- `dynamic_cast` ist sog Down-Cast, also Cast von Basisklasse auf abgeleitete Klasse
  - Gibt Nullpointer zurück wenn cast fehlschlägt
  - Nur bei polymorph verwendeten Obj möglich
  - Nur für Referenzen und Pointer
- Up-Cast passieren implizit (ist-ein Beziehung)
- Braucht man Cast von Obj der Basiskl auf abg Kl
  - ⇒ Konstr in abg Klasse erstellen

Matrixaddition

- Erweitere Kl `SquareMatrix` aus Folie 311 um Konstr
  - `SquareMatrix(const Matrix& mat)`
  - Für Cast `Matrix --> SquareMatrix`
- Implementierung der Addition nur für Basisklasse

```
1   #include "Matrix.hpp"
2
3   SquareMatrix::SquareMatrix(const Matrix& mat) {
4     if(mat.getN()<mat.getM())
5       *this = SquareMatrix(mat.getN(),0);
6     else
7       *this = SquareMatrix(mat.getM(),0);
8     for(int j=0; j<getN(); ++j)
9       for(int k=0; k<getN(); ++k)
10        setEntry(j,k,mat.getEntry(j,k));
11  }
12
13  const Matrix operator+(const Matrix& lhs,
14                         const Matrix& rhs) {
15    Matrix tmp(lhs.getN(),lhs.getM(),0);
16    for(int j=0; j<tmp.getN(); ++j)
17      for(int k=0; k<tmp.getM(); ++k)
18        tmp.setEntry(j,k,lhs.getEntry(j,k)
19                         +rhs.getEntry(j,k));
20    return tmp;
21  }
22
23  int main() {
24      SquareMatrix mat1(2,2);
25      SquareMatrix mat2(2,-1);
26
27      SquareMatrix mat3 = mat1 + mat2;
28  }
```

Virtuelle Methoden 1/4

- Ziel: Klasse für untere Dreiecksmatrizen
  - Effiziente Speicherung ⇒ Datenstruktur ändern
    - Nulleinträge nicht speichern
  - Redefinition der `get,set` Funktion

```
1  #ifndef _MATRIX_HPP
2  #define _MATRIX_HPP
3
4  #include <iostream>
5  #include <vector>
6  using namespace std;
7
8  class Matrix{
9  protected:
10   int m,n; //dimensions
11   vector<double> entries;
12 public:
13   int getM() const {return m;}
14   int getN() const {return n;}
15   virtual double getEntry(int i, int j) const;
16   virtual void setEntry(int i, int j, double e);
17   void plot();
18   Matrix();
19   Matrix(int m, int n, int init);
20 };
21
22 class TriangularMatrix : public Matrix {
23 public:
24   double getEntry(int i, int j) const;
25   void setEntry(int i, int j, double entry);
26   TriangularMatrix(int n, int init);
27 };
28
29 #endif // _MATRIX_HPP
```


Virtuelle Methoden 2/4

```
1  #include "Matrix.hpp"
2
3  Matrix::Matrix(): m(1), n(1),
4    entries(vector<double>(m*n, 0)){}
5
6  Matrix::Matrix(int m, int n, int init): m(m),
7    n(n), entries(vector<double>(m*n, init)){}
8
9  double Matrix::getEntry(int i, int j) const {
10   return entries[i+j*getM()];
11 }
12
13 void Matrix::setEntry(int i, int j, double e) {
14   entries[i+j*getM()] = e;
15 }
16
17 void Matrix::plot() {
18   for(int i = 0; i < getM(); i++) {
19     for(int j = 0; j < getN(); j++)
20       cout << " " << getEntry(i,j);
21     cout << endl;
22   }
23 }
```

- Implementierung der Methoden der Klasse `Matrix`
- Einträge spaltenweise in einem Vektor gespeichert

Virtuelle Methoden 3/4

```
1  #include "Matrix.hpp"
2
3  double
4  TriangularMatrix::getEntry(int i, int j) const {
5    if(j > i)
6      return 0;
7    else
8      return entries[i-j+j*getN()-0.5*j*(j-1)];
9  }
10
11 void TriangularMatrix::setEntry(int i, int j,
12                                  double val) {
13   entries[i-j+j*getN()-0.5*j*(j-1)] = val;
14 }
15
16 TriangularMatrix::TriangularMatrix(int dim,
17                                    int init) {
18   n = dim;
19   m = dim;
20   entries = vector<double>(0.5*n*(n+1),init);
21 }
```

- Effiziente Speicherung:
  - Es werden nur Elemente $\neq 0$ gespeichert
  - Vektor hat Länge $n(n+1)/2$
  - Spaltenweise Speicherung
    - Zugriff auf Element `i,j` vgl. Zeile 8, 13

Virtuelle Methoden 4/4

```
1  #include "Matrix.hpp"
2
3  int main() {
4    Matrix mat1(3,4,2);
5    mat1.plot();
6
7    TriangularMatrix mat2(3,4);
8
9    mat2.setEntry(2,1,9);
10
11   mat2.plot();
12 }
```

- Ausgabe:
```
2 2 2 2
2 2 2 2
2 2 2 2
4 0 0
4 4 0
4 9 4
```

- Durch `virtual` wird in Methode `plot` die redefinierte `getEntry` Methode aufgerufen
- Ohne `virtual`: In `plot()` würde Methode `getEntry` aus Klasse `Matrix` verwendet werden

Was es zu wissen gibt

- Virtuelle Methoden erlauben Polymorphie
  - Abgeleitetes Objekt verhält sich wie Basisobjekt
- Dies geschieht durch dynamische Bindung
  - Implementierung wird zur Laufzeit ausgewählt
- Methoden als virtuell deklarieren
  - wenn sie polymorph verwendet werden sollen
  - wenn Implementierung überschrieben wird
- In diesem Fall auch virtueller Destruktor
  - Objekt wird sonst evtl. nicht richtig bereinigt
- Konstruktoren können nicht virtuell sein
- In Java sind Methoden standardmäßig virtuell
- In C++ explizit durch Benutzer
- generell: Erzeugung von V-tables kostet
  - Code wird dadurch langsamer
  - nicht alles als `virtual` deklarieren

Abstrakte Datentypen

- Objekte vom Typ `Figur` machen wenig Sinn
  - da sehr abstrakt
- Klasse `Figur` ist nützlich, da man gemeinsame Eigenschaften zusammenfassen kann
  - z.B. Schwerpunkt, Umfang, etc.
- Konzept der Abstrakten Klassen
  - dienen als Schablone für abgeleitete Klassen
  - können gemeinsame Funktionen implementieren
  - müssen nicht alle Details implementieren
- Keine Objekte von abstrakten Klassen möglich
  - erleichtert Fehlersuche

Beispiel 1/2

- Abstrakte Klasse `Figur`

```
1  class Figur {
2    double schwerPunkt[2];
3  public:
4    virtual double getFlaeche() = 0;
5  };
```

- Abstrakte Methode in Zeile 4
- Abstraktion durch =0 nach virtueller Methode
- Syntax: `virtual rtype funName(varlist) = 0`
  - `rtype` ist Typ d Rückgabewerts von Fkt `funName`
  - `varlist` bezeichnet beliebige Inputparameter
  - Virtuelle Methode wird durch nachgestelltes `= 0` als abstrakt deklariert
- AM müssen nicht implementiert werden
- AM nennt man auch rein virtuell
- Durch eine AM wird die ganze Klasse abstrakt
  - abstrakte Kl. können nicht instanziiert werden

Beispiel 2/2

- Abgeleitete Klassen `Kreis, Dreieck`:

```
 1  #include<cmath>
 2
 3  class Figur {
 4    double schwerPunkt[2];
 5  public:
 6    virtual double getFlaeche() = 0;
 7  };
 8
 9  class Kreis : public Figur {
10    double radius;
11  public:
12    double getFlaeche() {
13      return radius*radius*3.14159;
14    }
15  };
16
17  class Dreieck : public Figur {
18    double a[2],b[2],c[2];
19  public:
20    double getFlaeche() {
21      return fabs(0.5*( (b[0]-a[0])*(c[1]-a[1])
22                       -(c[0]-a[0])*(b[1]-a[1])));
23    }
24  };
```

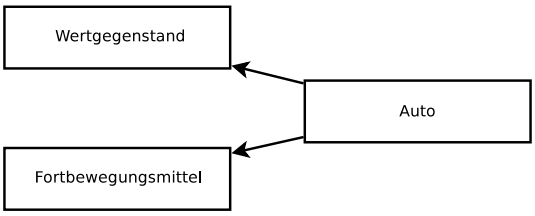
- `Kreis` und `Dreieck` redefinieren die AM `getFlaeche`
  - alle AM müssen redefiniert werden

Übersicht - abstrakte Klassen

- Dienen als Schablone / Schnittstelle für Vererbung
- Können nicht instanziiert werden
  - Fehlerfindung durch Compiler!
- Was Sie tun sollten:
  - Verwenden Sie AK für gemeinsame Funktionen (delegieren nach oben)
  - jedoch nur wenn sie für alle Kinder relevant sind
  - redefinieren Sie alle abstrakten Methoden
  - definieren Sie alle Methoden abstrakt die zu redefinieren sind

Mehrfachvererbung

- C++ erlaubt Vererbung mit multiplen Basisklassen



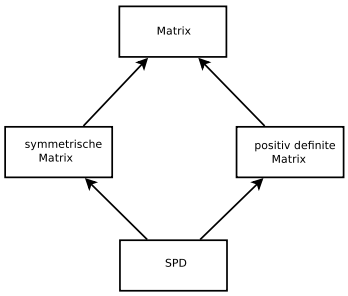
- Syntax:

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

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

Diamantvererbung 1/2

- Es könnte eine gemeinsame Oberklasse geben



- Führt zu Mehrdeutigkeit
  - Felder und Methoden sind mehrfach vorhanden
  - Unklar worauf zugegriffen werden soll
  - Speicherverschwendung
  - Schlimmstenfalls: Objekte inkonsistent
- SPD = symmetrisch positiv definite Matrix

Diamantvererbung 2/2

- Klasse Matrix hat ein Feld `int n`
- beide abgeleiteten Klassen erben `int n`
- SPD erbt von zwei Klassen
  - SPD hat `int n` doppelt
- Lösung 1: Zugriff mittels vollem Namen

  `symmMat::n` bzw. `posDefMat::n`

- Unschön, da Speicher dennoch doppelt
  - unübersichtlich
  - schlimmstenfalls sogar verschiedene Werte
- Lösung 2: virtuelle Basisklassen

```
class symmMat : virtual public matrix
class posDefMat : virtual public matrix
```

- Virtuelle Basisklasse wird nur einmal eingebunden
- Literatur: Uneinigkeit ob Mehrfachvererbung sinnvoll sein kann
- Konzept nicht möglich in Java, C#

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

- Ziel: Maximum berechnen / 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  }
```

- Gleicher Code für viele Probleme
  - Vererbung bietet sich hier nicht an
- Lösung: Templates

Funktionstemplate 1/2

- Funktionalität unabhängig vom Datentyp:

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  template <typename T>
 5  T square(const T& t) {
 6    return t*t;
 7  }
 8
 9  int main() {
10    cout << square<double>(4.7) << endl;
11    cout << square(4.7) << endl;
12  }
```

- Ausgabe: 22.09
  22.09
- `t` ist Platzhalter für Daten vom Typ `T`
  - Für alle DT die Multiplikation $*$ bereitstellen z.B. Quadrate von Matrizen
- Bei Aufruf DT in spitze Klammern (Z. 10)
- kann auch weggelassen werden (Z. 11)
- `template <typename T> rtype funName(varlist)`
  - Einleitung durch Schlüsselwort `template`
  - Templateparameter in spitzen Klammern
    - allgemeiner Datentyp: `typename`
    - alternative Syntax: `class`
    - `T` ist Name des Parameters (beliebiger Name)
  - `rtype` ist Typ des Rückgabewerts von `funName`
  - Rückgabe u Eingabe können vom Typ `T` sein
    - auch Referenz oder Pointer auf `T` möglich

Funktionstemplate 2/2

- Was passiert eigentlich bei folgendem Code?

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

- 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. Funktion `square` wird automatisch anhand des Templates (= Vorlage) generiert
  - also nur für die Typen die wirklich benötigt

Klassentemplate 1/3

- Auch allgemeine Klassen möglich
  - haben wir bereits verwendet (`vector`)
- Automatische Speicherverwaltung bei Pointern:

```
 1  template <typename T>
 2  class simple_ptr {
 3  public:
 4    simple_ptr(T* ptr);
 5    ~simple_ptr();
 6    T& operator*();
 7    T* operator->();
 8  private:
 9    //Die Klasse soll nicht kopierbar sein.
10    simple_ptr(const simple_ptr&);
11    simple_ptr& operator=(const simple_ptr&);
12
13    T* m_ptr;
14  };
```

- Idee: Speicher automatisch freigeben
  - verhindert Speicherlecks
- Für Pointer von beliebigem Typ
  - Das nennt man Smartpointer
- Um zu verhindern, dass Obj der Kl kopiert werden kann, schreibt man Kopierkonstruktor und Zuweisungsoperator in den `private` Bereich

Klassentemplate 2/3

- Implementierung von Klassenmethoden
  - Beispiel: Konstruktor

```
1  template <typename T>
2  simple_ptr<T>::simple_ptr(T* ptr) : m_ptr(ptr)
3  {}
```

- Syntax:
  - voranstellen von `template <typename T>`
  - Wichtig: `<T>` nach Klassenname
    (Methode für Klasse `simple_ptr<T>`)
- tatsächliche Implementierung wie gehabt

Klassentemplate 3/3

- Restliche Klassenmethoden

```
 1  template <typename T>
 2  simple_ptr<T>::~simple_ptr() {
 3    if(m_ptr)
 4      delete m_ptr;
 5    m_ptr = NULL;
 6  }
 7
 8  template <typename T>
 9  T& simple_ptr<T>::operator*() {
10    return *m_ptr;
11  }
12
13  template <typename T>
14  T* simple_ptr<T>::operator->() {
15    return m_ptr;
16  }
17
18  int main() {
19    simple_ptr<string>
20        some_stringptr(new string("Hallo"));
21    cout << *some_stringptr << endl;
22  }
```

- Ausgabe: `Hallo`
- Destruktor gibt Speicher wieder frei (Garbage Collection)
- Operatoren `*` und `->` überladen
  - für Funktionalität von Pointer

Private Methoden

- Probleme bei Kopien von Smartpointern:

```
 1  int main() {
 2    simple_ptr<string> p(new string("blub"));
 3    p->size();
 4    {
 5      simple_ptr<string> q = p;
 6      q->size();
 7      // Destruktor von q wird aufgerufen
 8    }
 9    p->size();
10  }
```

- Pointer wird kopiert (Z. 5)
  - Hier: Zuweisung in (Z. 5) nicht möglich (Methode ist `private`)
- Speicher wird freigegeben (Z. 8)
- Problem: Speicher von p auch freigegeben
- Sonst Lösung: Kopien zählen
  - Speicher nur freigeben wenn kein Zugriff mehr (wird hier nicht vertieft)

Danke für Ihre Aufmerksamkeit