Programmiersprachen für numerische Software - ein Überblick

Peter Junglas

8. Oktober 1999

Inhaltsverzeichnis

Kapitel 1

Überblick über Programmiersprachen 1: Prozedurale Sprachen

1.1 Prinzipien der strukturierten Programmierung

Idee: Verständlichere, besser wartbare Programme durch

- Datenstrukturen mit zugehörigen Grundoperationen
- Kontrollstrukturen zur Steuerung des Programmablaufs
- Techniken zur Zerlegung des Programms in unabhängige Untereinheiten

1.1.1 Datenstrukturen

Variablen werden Datentypen zugeordnet, die die Wertebereiche und die möglichen Grundoperationen festlegen.

Einfache Datentypen:

- vordefinierte Datentypen für ganze Zahlen und Fließkommazahlen,
- meistens außerdem für Zeichen und für Wahrheitswerte (wahr/falsch)

Felder:

- Zusammenfassung mehrerer Elemente des gleichen Typs
- Zugriff durch einen ganzzahligen Index

Zeichenketten:

- Folge von Zeichen zur Abspeicherung von Texten

Strukturen:

- Zusammenfassung von Elementen verschiedener Typen

Zeiger:

- Verweise auf andere Variable

Die Elemente der zusammengesetzten Datentypen können auch selbst zusammengesetzt sein. Auf diese Weise entstehen komplexe Typen, z.B.:

- Felder als Elemente von Feldern ergeben mehrdimensionale Strukturen, die sich zur Beschreibung von Matrizen oder Gittern eignen.
- Strukturen, die u.a. Zeiger enthalten, die wiederum auf eine solche Struktur verweisen, sind die Grundlage dynamischer Datentypen (Listen, Bäume).

1.1.2 Kontrollstrukturen

Die wichtigsten Kontrollstrukturen, die in ähnlicher Form in den meisten Sprachen auftauchen, sind:

Block:

- Folge von Anweisungen, die als Einheit betrachtet wird

Bedingte Anweisung:

- Anweisung, die nur ausgeführt wird, wenn eine bestimmte Bedingung erfüllt (oder verletzt) ist.
- Grundform:

```
if Bedingung then
   Anweisung1
else
   Anweisung2
end
```

Schleife:

- Wiederholung von Anweisungen bis zum Erfüllen einer Abbruchbedingung.
- Grundform:

```
do
  Anweisung1
  if (Bedingung) exit
  Anweisung2
end
```

- Häufig existieren Formen für den Fall, dass `Anweisung1` oder `Anweisung2` fehlen (`while`- bzw. `repeat`-Schleife).

Laufanweisung:

- wichtiger Spezialfall: Schleife, bei der eine Indexvariable durch einen bestimmten Wertebereich läuft.
- Grundform:

```
do i=1, 100
  Anweisung(i)
end
```

1.1.3 Unterprogramme

Unterprogramm:

- Programmteil zur Lösung einer Teilaufgabe, kann mehrmals im Programm aufgerufen werden.
- Austausch von Informationen zwischen Unterprogramm und aufrufendem Programmteil durch Liste von Variablen (Parameterliste)

Übergabe von Parametern auf zwei Weisen möglich:

call by value:

- das Unterprogramm erhält Kopien der Parameterwerte
- kein Zugriff des Unterprogramms auf die Werte des Hauptprogramms (Eingabe-Parameter)
- bei größeren Datenstrukturen u.U. aufwendig

call by reference:

- das Unterprogramm erhält Verweise auf die Parameter
- Hauptprogramm bekommt ggf. die geänderten Werte (Ausgabe-Parameter)
- kein Kopieren erforderlich

Eigenschaften von Variablen in Unterprogrammen:

Sichtbarkeit:

- Variable können nur im sie definierenden Programmteil sichtbar sein (lokal) oder im ganzen Programm (global).

- Globale Variable vereinfachen den Informationsfluß, schaffen aber starke Abhängigkeiten zwischen Unterprogrammen.

Lebensdauer:

- Variable können beim Beenden des sie definierenden Unterprogramms ihren Wert verlieren (automatisch) oder ihn zwischen Aufrufen des Unterprogramms behalten (statisch).
- Lokale Variable sind meistens automatisch. Für manche Anwendungen (z.B. Anzahl der Aufrufe) werden statische Variable benötigt.

Funktion:

- Unterprogramm mit einem besonderen Rückgabewert, der in Ausdrücken direkt verwendet werden kann.
- Anlehnung an die mathematische Schreibweise: Statt

  ```
  exp(x, result)
  y = a * result
  ```

  schreibt man einfach

  ```
  y = a * exp(x)
  ```

- Wichtige Funktionen meist in Standard-Bibliotheken verfügbar, etwa
  - mathematische Funktionen (`sqrt`, `sin`, `log`)
  - Funktionen zur Speicherverwaltung (`alloc`, `free`)
  - Ein- und Ausgabe-Funktionen (`read`, `write`, `open`)

1.2 FORTRAN77 - der Urahn der numerischen Programmierung

1.2.1 Übersicht

Geschichte:

- in den 50ern entwickelt, eine der ersten "Hochsprachen"
- Anwendungsbereich: numerisch/naturwissenschaftlich ("FORmula TRANslator")
- erster Sprachenstandard überhaupt: Fortran 66
- lange weitverbreitet: FORTRAN77

Eigenschaften:

- besondere Unterstützung von Matrizen und komplexen Zahlen
- nur wenige Elemente der strukturierten Programmierung
- (heute) veraltete Konstruktionen zur Speicherverwaltung

1.2.2 Sprachelemente

Grundelemente:

- Programmtext an Lochkartenformat ausgerichtet
  (Text in Spalten 7 - 72, Label in 1-5 etc.)
- keine Kleinbuchstaben
- Kommentare mit C oder * in Spalte 1

Datentypen:

- Einfache Datentypen:
  - ganze Zahlen: `INTEGER`
  - Fließkommazahlen: `REAL`, `DOUBLE PRECISION`
  - Komplexe Zahlen: `COMPLEX`
  - Zeichen: `CHARACTER`
  - Logische Größen: `LOGICAL`
- Felder:
  - Vereinbarung eines Feldes von 100 `REAL`-Werten:

    ```
          REAL A(100)
    C     erstes Element ist A(1)
          REAL A(-5:5)
    C     11 Elemente
    ```

  - Zugriff auf die Elemente durch ganzzahligen Index:

    ```
          A(6) = 2.71828
    ```

  - keine Tests auf Index-Überschreitung (aber häufig als Compiler-Switch)

    ```
          A(2000) = 47.11
    ```

    unvorhersehbares Laufzeitverhalten!
  - keine dynamischen Felder
  - Mehrdimensionale Felder:

    ```
          DOUBLE PRECISION A(1000,1000)
    ```

    Speicherreihenfolge: `A(1,1), A(2,1), ..`
    (erst Spalten, dann Zeilen)
- Zeichenketten:
  - Folge von Zeichen fester Länge

```
CHARACTER*9  GRUSS
GRUSS = 'HALLOCHEN'
```

  - Teile herausgreifen:

```
GRUSS(1:5)
```

  - kein Feld, `GRUSS(2)` ist nicht erlaubt!

- Strukturen: keine
- Zeiger: keine

Kontrollstrukturen:

- Bedingte Anweisung:

```
IF (Bedingung) THEN
  Anweisungen1
ELSE
  Anweisungen2
END IF
```

- Schleifen:
  - keine allgemeinen Schleifen
  - gezielt mit `GOTO` nachbilden, z.B. abweisende Schleife:

```
100   IF (Bedingung) THEN
        Anweisungen
        GOTO 100
      END IF
```

- Laufanweisung:
  - einfache Zählschleife mit expliziter Schrittweite

```
      DO 101 I = -3, 8, 2
        Anweisungen
 101  CONTINUE
```

  - häufig `DO ... END DO`, ist *nicht* FORTRAN77-Standard

Programmstrukturen:

- Unterprogramme:
  - `SUBROUTINE` und `FUNCTION`
  - immer getrennte Übersetzungseinheit (unabhängig von Datei-Aufteilung)
  - Parameterübergabe durch call-by-reference

  - Matrizen als Parameter: Unterscheidung zwischen physikalischer und logischer Dimension nötig
  - Routinen als Parameter möglich (z.B. für allgemeine Integrations-Routine)

- COMMON-Blöcke:
  - Zusammenstellung von Variablen, die global verwendet werden können
  - Alternative zu langen Parameterlisten (mangels Daten-Strukturen)
  - häufig Ursache schwer zu findender Fehler

- mathematische Standard-Funktionen:
  - jeweils Funktionen für jeden Fließkommatyp, z.B.
    `SQRT`, `DSQRT`, `CSQRT` für `REAL`, `DOUBLE PRECISION`, `COMPLEX`
  - dazu generische Funktionen, die sich an den Datentyp anpassen
  - alle Grundfunktionen vorhanden, u.a (generisch):
    `SIN`, `COS`, `TAN`, `ASIN`, `ACOS`, `ATAN`, `ATAN2`
    `SINH`, `COSH`, `TANH`
    `SQRT`, `EXP`, `LOG`, `LOG10`

1.2.3 Beispielprogramme

Matrixmultiplikation:

- Sourcen: s. Anhang A.1.1
- Initialisierung:
$$A_{ij} = i + j, B_{ij} = i - j + \delta_{ij}$$
- Tests:
  Spur($A * B$) und ein Nicht-Diagonal-Element prüfen
- Bemerkungen:
  immer Compiler-Optimierungen einschalten (`-O3` o.ä.)

Matrixmultiplikation mit Blas-Routine:

- Sourcen: s. Anhang A.1.2
- Blas:
  - "Basic Linear Algebra Subroutines"
  - Standard für grundlegende Matrix- und Vektor-Operationen
  - hochoptimierte Bibliotheken verfügbar
  - Routinen jeweils für `REAL`, `DOUBLE PRECISION`, `COMPLEX`, `DOUBLE COMPLEX`
  - Namen beginnen jeweils mit S, D, C, Z

- Bemerkungen:
  - `DGEMM`: verallgemeinerte Matrixmultiplikation

$$C = \alpha C + \beta A \cdot B$$

  - A, B wahlweise auch transponiert oder konjugiert
  - logische Matrixgrößen und jeweils Zahl der (physikalischen) Zeilen

Vergleich der Laufzeiten:

- Verwendete Rechner:
  - pcpj: Pentium II/400MHz
  - orion: HP V2250
- Ergebnisse (in Sekunden):

Rechner	ohne Opt.	-O3	Blas
pcj	345.5	76.8	7.6
orion	279.3	70.7	2.9

1.2.4 FORTRAN77 für numerische Anwendungen

Besonderheiten:

- Potenzierungsoperator: `A**B`
- komplexe Zahlen
- überladene Standardfunktionen

Geschwindigkeit:

- Matrixoperationen sehr schnell
- hochoptimierende Compiler verfügbar

Bemerkungen:

- lange *die* Sprache für numerische Anwendungen
- große vorhandene Basis an Programmen und Bibliotheken
- programmiertechnisch veraltet
- Ablösung durch Fortran90 (s.u.) sehr zögernd

1.3 C - das universelle Arbeitspferd

1.3.1 Übersicht

Geschichte:

- entwickelt um 1970, primär zur Betriebssystem-Entwicklung
- standardisiert 1989 durch ANSI
- auf sehr vielen Plattformen verfügbar
- weit verbreitet in professioneller SW-Entwicklung

Eigenschaften:

- alle Elemente der strukturierten Programmierung enthalten
- sehr knapper (aber auch unverständlicher) Code möglich
- erlaubt sehr hardware-nahe Programmierung

1.3.2 Sprachelemente

Grundelemente:

- Programmtext in freier Form, ohne Zeilenstruktur
- Groß- und Kleinbuchstaben werden unterschieden
- Kommentar: `/* erklaer mal was */`
- Anweisungen werden mit ; abgeschlossen

Datentypen:

- Einfache Datentypen:
  - ganze Zahlen: `char`, `short`, `int`, `long`
  - Fließkommazahlen: `float`, `double`, `long double`
  - Zeichen: `char` (Abspeicherung als kleiner Integer, meist 1 Byte)
  - Logische Größen: nicht vorhanden
Ersatz: ganze Zahlen ($0 \mathrel{\hat{=}}$ false, $(\neq 0) \mathrel{\hat{=}}$ true)
- Felder:
  - Vereinbarung eines Feldes von 100 `double`-Werten:

    ```
    double a[100];
    ```

  - Zugriff auf die Elemente durch ganzzahligen Index:

```
a[6] = 2.71828;
```

erstes Element ist `a[0]`

  – keine Tests auf Index-Überschreitung
  – Mehrdimensionale Felder:

```
double a[1000][1000][1000];  /* gib mir 8 GB! */
```

Speicherreihenfolge: `a[0][0][0], a[0][0][1] ..`
(erst Zeilen, dann Spalten)

- Zeichenketten:
  – Felder von Zeichen, abgeschlossen durch ein NULL-Zeichen ('\0')
  – Konstante:

```
char littleString[] = "juhu";
```

  – Initialisierung:

```
char myString[] = "joobar";  /* 7 Zeichen lang! */
```

- Strukturen:
  – Definition des neuen Typs:

```
struct Person {
  char    name[20];
  int     alter;
  float   groesse;
}
```

  – Benutzen:

```
struct Person mrPresident = {"Bill", 53, 1.78};
mrPresident.height = mrPresident.height - 0.03;  /* tjaja, das Alter */
mrPresident.alter++;        /* Geburtstag */
```

- Zeiger:
  – Adressoperator & (Zeiger auf ein Objekt)
  – Inhaltsoperator * (Inhalt eines Zeigers)
  – Beispiel:

```
double  a = 2.7;
double  * pointer;

pointer = &a;      /* pointer zeigt auf a */
*pointer = 3.1;    /* Inhalt von pointer wird gesetzt */
                          /* a hat den Wert 3.1 ! */
```

– generischer Zeigertyp (`void *`) : darf auf alles zeigen

– dynamischer Speicher mit Standardfunktionen:

```
double * a;
a = (double *) malloc(1000 * sizeof(double));
a[100] = 28.8;
free(a);
```

Speicherverwaltung per Hand - fehlerträchtig!

Kontrollstrukturen:

- Block:

```
{
  Anweisung1
  Anweisung2
    ...
}
```

- Bedingte Anweisung:

```
if (Bedingung)
  Anweisung1
else
  Anweisung2
```

- Verzweigung:

```
switch (Auswahl-Ausdruck) {
 case Konstante1:
    Anweisung1
    break;
 case Konstante2:
    Anweisung2
    break;
   ...
 default:
    AnweisungN
}
```

- Abweisende Schleife:

```
while (Bedingung)
  Anweisung
```

- Nichtabweisende Schleife:

```
do
  Anweisung
while (Bedingung);
```

- Laufanweisung:
  - sehr allgemeine Syntax:

    ```
    for (Initialisierung; Bedingung; Inkrementierung)
      Anweisung
    ```

    Abkürzung für:

    ```
    Initialisierung;
    while (Bedingung) {
      Anweisung
      Inkrementierung;
    }
    ```

  - einfache Zählschleife als Spezialfall:

    ```
    for (i=0; i<100; i++)
      Anweisung(i)
    ```

  - spezielle Anweisungen:
    * `break`: sofortiges Verlassen der Schleife
    * `continue`: Beginn des nächsten Schleifendurchgangs (incl. Inkrementierung)

Programmstrukturen:

- Unterprogramme:
  - syntaktisch immer Funktionen
  - spezieller Rückgabewert void
  - Parameterübergabe durch call-by-value
  - call-by-reference mit Zeigern nachbilden
- Gliederungsebenen:
  - Hauptprogramm = spezielle Funktion `main`
  - Zusammenfassen mehrerer Funktionen in Dateien
- mathematische Standard-Funktionen:
  - die üblichen Grundfunktionen:
    `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
    `sinh`, `cosh`, `tanh`
    `pow`, `sqrt`, `exp`, `log`, `log10`
  - Parameter und Ergebnisse vom Typ `double`
  - vordefinierte Konstanten für Zahlbereiche und Genauigkeit

1.3.3 Beispielprogramme

Matrixmultiplikation:

- Sourcen: s. Anhang A.2.1
- Matrizen hier und im Fortran-Beispiel nicht identisch (wegen 0 als kleinstem Index)

Matrixmultiplikation mit Blas-Routine:

- Sourcen: s. Anhang A.2.2
- Blas z.Zt. nur für FORTRAN defininiert
- Kopplung C - Fortran:
  - Datentypen abbilden (`int` $\leftrightarrow$ `INTEGER` etc.)
  - Fortran-Parameter immer call-by-reference $\rightarrow$ Zeiger übergeben
  - Matrix-Argumente in C transponiert abgespeichert
  - Trick: nicht explizit transponieren, sondern Reihenfolge der Parameter a und b ändern mit
  $$C^T = (A \cdot B)^T = B^T \cdot A^T$$

1.3.4 C für numerische Anwendungen

Besonderheiten:

- dynamische Matrizen langsam (indirekte Adressierung) oder unhandlich

Geschwindigkeit:

- Compiler-Optimierung auf höherer Ebene oft nicht effektiv (wegen unbeschränkter Pointer)
- schneller Code möglich durch Handoptimierung ("hardware-nah")

Bemerkungen:

- weit verbreitet bei System-Programmierern
- keine besonderen Abstraktionen für Numerik

1.4 Fortran90 - die moderne Alternative

1.4.1 Übersicht

Geschichte:

- Modernisierung und Weiterentwicklung von Fortran
- ISO-Standard: 1991
- kontinuierliche Weiterentwicklung geplant
- aktuell: Fortran95

Eigenschaften:

- alle Elemente der strukturierten Programmierung
- Unterstützung für abstrakte Datentypen
- hohe Effizienz auf modernen Großrechnern möglich

1.4.2 Sprachelemente

alle Elemente von FORTRAN77 sind Teil von Fortran90 einige Konstruktionen als veraltet ("obsolescent") bezeichnet

Grundelemente:

- freier Programmtext
- Groß-/Kleinschreibung nicht signifikant
- Kommentare mit !, bis zum Zeilenende

Datentypen:

- Einfache Datentypen:
  - spezielle Parameter für Typen unterschiedlicher Länge (`KIND`)
  - Unterstützung für portable Datentypen
- Felder:
  - punktweise Operationen auf Feldern

    ```
    A = B + C                  ! A, B, C gleiche Dimensionen
    A = sin(A)
    ```

  - Abschnitte von Feldern

    ```
    A(2:11:3, 3:11:2) = 0.0    ! setzt 4x5-Teilmatrix von A
    ```

  - dynamische Felder ("allocatable arrays")

```
real, allocatable, dimension(:,:) :: A
allocate(a(n,m))
```

- spezielle Matrix-Funktionen
  - Abfrage der Dimensionen mit `size`, `shape`
  - `matmul`, `sum`, `maxval`

- Strukturen:
  - ähnlich wie in C oder Pascal, leicht andere Syntax

- Zeiger:
  - Zeiger sind typsicher (zeigen auf Objekte eines festen Typs)
  - Ziele eines Zeigers werden mit dem `target`-Attribut gekennzeichnet
  - keine Zeiger-Arithmetik

Kontrollstrukturen:

- alle nötigen Konstrukte vorhanden:
  - bedingte Anweisung
  - Verzweigung
  - allgemeine Schleifen (incl. Abbruch mit `exit`)
  - Laufanweisung (mit `end do`)

Programmstrukturen:

- Funktions-Deklarationen mit Interfaces
  eigene generische Funktionen
- Operatoren überschreiben oder selbst definieren
- Module zur Kapselung von Typen, Daten, Funktionen und Operatoren
  ermöglicht abstrakte Datentypen

1.4.3 Beispielprogramme

Matrixmultiplikation:

- Sourcen: s. Anhang A.3.1
- Standard-Funktion `matmul`
- `real(kind=8)` entspr. `double precision` bei HP-UX-f90
  `kind`-Nummer nicht portabel
- `forall`-Konstrukt, drückt Parallelität aus:

```
forall (i = 1:n, j = 1:n)
  a(i,j) = i + j
  b(i,j) = i - j
end forall
```

Matrixmultiplikation mit Blas-Routine:

- Sourcen: s. Anhang A.3.2
- Modul kinds für portable Datentypen
- Anpassung der BLAS-Routine an Fortran90 durch eigene Interface-Routine
- vorgeschlagenes Fortran95-Interface für BLAS
  - generische Funktionen
  - Matrixgrößen: logisch = physikalisch
  - keine Parameter für Matrixgrößen
  - besondere Parameter (Transponierung, Addition) optional

Rationale Zahlen als abstrakter Datentyp:

- Sourcen: s. Anhang A.3.3
- Rationale Zahl = Struktur aus Zähler und Nenner
- Funktionen für die Bruch-Arithmetik
- Brüche wie Standard-Datentypen durch Operatoren
- Normaldarstellung durch interne Funktionen zum Kürzen

1.4.4 Fortran90 für numerische Anwendungen

Besonderheiten:

- Matrixfunktionen
- abstrakte Datentypen incl. Operatoren
- portable Standardtypen incl. komplexer Zahlen

Geschwindigkeit:

- so schnell wie FORTRAN77 (im Prinzip)

Bemerkungen:

- immer noch *die* Sprache für numerische Anwendungen
- (fast) kompatibel zu FORTRAN77 (alte Programme, Bibliotheken)
- kommt leider etwas aus der Mode

1.5 Übungen

1.5.1 Übung 1

Übersetzen Sie alle Beispielprogramme. Probieren Sie dabei verschiedene Optimierungs-Optionen der Compiler aus und vergleichen Sie die Laufzeiten.

Anmerkung: Die Kommandos und Optionen der diversen Compiler sind systemabhängig. Suchen Sie im Online-Manual die entsprechenden Befehle, z.B. mit

```
man -k compiler
```

Fortgeschrittene sollten zum Übersetzen das make-Programm einsetzen.

1.5.2 Übung 2

Schreiben Sie ein Programm in Ihrer Lieblingssprache, das mit Hilfe von Routinen aus der LAPACK-Bibliothek ein lineares Gleichungssystem mit 1000 Unbekannten löst. Denken Sie sich zum Testen einfache Koeffizienten-Matrizen und rechte Seiten aus, deren Lösung Sie kennen, ähnlich wie in den Beispielprogrammen.

Anmerkung: Die benötigte LAPACK-Routine heißt `DGESV`.

Kapitel 2

Überblick über Programmiersprachen 2: Objektorientierte Sprachen

2.1 Prinzipien der objektorientierten Programmierung

Idee: Programmentwurf basiert auf den Datenstrukturen (Objekten) statt auf den Operationen (Routinen)

2.1.1 Datenkapselung

Klasse:

- Zusammenfassung von Datenstrukturen und den darauf ausführbaren Operationen (Methoden) zu einem neuen Datentyp
- Methoden und Daten können gegen Zugriff von außen (= aus Methoden anderer Klassen) geschützt sein (privat).
- Klassen-Schnittstelle = Menge aller von außen sichtbaren Methoden und Daten.

Objekt:

- Variable eines Klassentyps
- Zustand des Objekts = Werte der internen Datenstrukturen

Beispiel:

- Gerüst einer Klasse für Matrizen:

```
Klasse Matrix
  öffentlich:
    neu(nSpalten, nZeilen)
```

```
    get(i, j)
    set(i, j, value)
    mult(matrix)
  privat:
    integer nRows, nCols
    double[] val
```

- Schnittstelle:
  - neue Matrix
  - Zugriff auf Elemente
  - Rechenmethoden
- immer konsistent:
  - `neu` prüft `nSpalten`, `nZeilen`
  - `set/get` prüft Indexbereich
  - kein direkter Zugriff auf Speicherbereiche
- Implementierung kann sich ändern:
  - keine Index-Prüfung
  - Multiplikation mit Strassen-Algorithmus

Bemerkungen:

- Änderung der Implementierung ändert nichts an Programmen, die die Klasse verwenden.
- Objekte werden häufig dynamisch erzeugt.
  - Problem: Speicher nicht mehr gebrauchter Objekte muss freigegeben werden
  - vom Programmierer: delete-Methode (schnell, fehlerträchtig), z.B. in C++
    vom Laufzeit-System: Garbage-Collection (langsam, einfach), z.B. in Java

2.1.2 Vererbung

Vererbung:

- Erzeugung neuer Klassen durch Erweiterung vorhandener
- neue Klasse bekommt alle Methoden und Daten der Basisklasse
- definiert zusätzlich weitere Methoden oder Daten
- kann Methoden der Basisklasse durch eigene ersetzen
- kann genau wie Basisklasse eingesetzt werden ("ist ein"-Beziehung)

Beispiel:

- ```
  Klasse BandMatrix erweitert Matrix
    öffentlich:
      mult(matrix)
    privat:
      integer bandbreite
  ```

Kapselung:

- kein Zugriff auf die privaten Elemente der Basisklasse
- Klasse kann Zugriff genau für die abgeleiteten Klassen erlauben

Mehrfach-Vererbung:

- Klasse erbt von mehreren Basisklassen
- problematisch u.a. durch Namenskollisionen
- in manchen Sprachen nicht möglich (z.B. Java)

2.1.3 Polymorphie

Grundprinzip objektorientierter Programmierung:

- Wenn
  - Methode in Basisklasse und abgeleiteter Klasse definiert
  - Variable der Basisklasse enthält Objekt der abgeleiteten Klasse
- dann
  - wird die Methode der abgeleiteten Klasse verwendet.

Beispiel:

- ```
  Matrix        m, m1
  Bandmatrix    bm
     ...
  m = bm                      ! geht, denn BandMatrix ``ist eine'' Matrix
  m.mult(m1)                  ! multipliziere von BandMatrix!
  ```

Late Binding:

- Festlegung der wirklich aufgerufenen Methode erst zur Laufzeit

Abstrakte Basisklasse:

- Klasse, die Methoden definiert, aber nicht implementiert
- legt einheitliches Verhalten für alle abgeleiteten Klassen fest
- nur für die Klassenstruktur wichtig, keine Objekte

2.2 Java - die Sprache des WWW

2.2.1 Übersicht

Geschichte:

- entwickelt von der Firma SUN
- Designziele: Sicherheit und Portabilität
- weite Verbreitung durch Einbettbarkeit in WWW-Browser ("Applets")

Eigenschaften:

- sauberes Sprachdesign (keine "Altlasten")
- viele Standard-Klassenbibliothek, u.a.:
  - Ein- und Ausgabe
  - graphische Benutzeroberflächen
  - Netzwerk-Programmierung
- Ablauf in einer "virtuellen Maschine" = maschinenunabhängiger Zwischencode
- Standard-Compiler (`javac`) erkennt Klassenabhängigkeiten und Interfaces
  - keine extra Deklarationen erforderlich
  - automatische Neukompilierung benötigter Klassen (wie `make`)

2.2.2 Beispielprogramm Matrixmultiplikation

Design:

- Sourcen: s. Anhang A.4.1
- Matrizen als "Speicherblöcke" statt Ansammlung von Zeilen
  Organisation wie in Fortran (Indizes ab 1, spaltenweise gespeichert)
- ausgiebige Fehlerprüfung
  bei Fehler Exception (s.u.) auslösen
- Ergebnismatrizen werden im aufrufenden Programm initialisiert
  alternativ: neue Matrix erzeugen und als Funktionswert zurückgeben
- interne Größen für abgeleitete Klassen zugänglich (`protected`)

Schnittstellen:

- Matrix:

```
Matrix(int rows, int cols)
int[] shape()
void set(int i, int j, double v)
double get(int i, int j)
void mult(Matrix b, Matrix result)
```

- SquareMatrix:

```
SquareMatrix(int n)
void mult(SquareMatrix b, SquareMatrix result)
double trace()
```

- TestMatrix:

```
TestMatrix(int n)
void run()
void check()
static public void main(String[] args)
```

Bemerkungen:

- Zugriff auf Objekte:
  - Variable für Klassentypen und Arrays sind Referenzen ("Zeiger")
  - auf null initialisiert
  - bekommen dynamisch Speicher mit new und Konstruktor

    ```
    Matrix  m = new Matrix(42, 28)
    ```

  - keine free-Routine, sondern automatisches Aufräumen ("garbage collector")
- Fehlerbehandlung durch Ausnahmen:
  - vordefinierte Klasse Exception und abgeleitete Klassen
  - Auslösen einer Ausnahme mit

    ```
    throw Exception(message) // Programm springt zum Errorhandler
    ```

  - Behandeln von Ausnahmen:

    ```
    try {
      // etwas, was Fehler auslösen könnte, etwa
      m.get(100, 100);
    } catch {
      // tue etwas, was den Fehler repariert oder beende vernünftig
      // Programm macht ggf. weiter hinter den catch-Blöcken
    }
    ```

  - Standard-Verhalten (kein `try-catch`-Block):
    Ausgabe von `message` und Stacktrace, dann Abbruch

    ```
    java.lang.IndexOutOfBoundsException: Invalid index: (4,2)
            at Matrix.get(Matrix.java:57)
            at TestMatrix.main(TestMatrix.java:44)
    ```

- Implementation von `set/get`:
  - sorgt für Fortran-artiges Verhalten
  - leicht an andere Modelle anpassbar
- Implementation von `mult`:
  - benutzt interne Struktur (`val[]`) statt `set/get`
  - deutlich schneller (Faktor 4), da keine Indextests nötig
- SquareMatrix:
  - `mult(Matrix, Matrix)` geht (natürlich) auch für `SquareMatrix`
  - aber: Ergebnis ist `Matrix`, nicht `SquareMatrix`!
- Hauptprogramm `main`:
  - `static` Routinen gehören zur Klasse
    können ohne Objekt aufgerufen werden
  - viele Klassen können eine Routine `main` haben
  - Aufruf:

    ```
    java TestMatrix
    ```

    startet `main` in Klasse `TestMatrix`

2.2.3 Java für numerische Anwendungen

Besonderheiten:

- maschinenunabhängige Numerik

Geschwindigkeit:

- sehr langsam wegen Interpretation von Zwischencode (echte Compiler allmählich verfügbar)
- keine Ausnutzung effizienter Hardware

Bemerkungen:

- große Bedeutung im Internet und in der Ausbildung
- z.Zt. Bemühungen, Numerik-Eigenschaften zu verbessern ("JavaGrande")

2.3 C++ - die eierlegende Wollmilchsau

2.3.1 Übersicht

Geschichte:

- objektorientierte Erweiterung von C
- Designziele: Performance und Vielseitigkeit
- ISO-Standard 1998
- weit verbreitete objektorientierte Sprache

Eigenschaften:

- enthält C als Teilmenge (bis auf einige kleine Unterschiede)
- sehr umfangreiche Funktionalität, z.B.
  - Templates
  - Mehrfach-Vererbung
- Klassenbibliotheken u.a. :
  - Containerklassen (Menge, Stack etc.)
  - I/O (Streams)
  - Strings

2.3.2 Beispielprogramm Matrixmultiplikation

- Sourcen: s. Anhang A.5.1
- Programm weitestgehend identisch zur Java-Version
- Trennung in Header-Datei (Schnittstelle) und Implementierung
- Zugriff auf Objekte:
  - Variable können Werte, Zeiger oder Referenzen (implizite Zeiger) sein
  - dynamischer Speicher über new oder Konstruktor
  - Initialisierungslisten für interne Daten eines Objekts
  - explizite Routine zum Aufräumen ("Destruktor") nötig
- Ausgabe mit Streams:
  - mit Standard-OutputStream `cout` und Operator << :

    ```
    cout << "Hello Universe!"
    ```

  - << überschrieben für die numerischen Datentypen:

```
int answer = 42;
cout << "Die Antwort lautet " << answer << "!"
```

  – spezielle Funktionen für Formatanweisungen einstreuen, z.B.:

```
double pi = 3.1415926;
cout << "pi = " << setprecision(6) << pi << endl;
```

  – Ausgabe kann auch in einen String (genauer: einen `ostrstream`) erfolgen

- Ausnahmen:
  – Standard-Verhalten: Programmabbruch, keine Meldung
  – expliziter `try-catch`-Block nötig, um Fehlermeldung zu bekommen

2.3.3 C++ für numerische Anwendungen

Besonderheiten:

- Parametrisierung von Klassen mittels Templates
- Überladen von Operatoren
- mit dem neuen Standard: komplexe Zahlen, schnelle Vektoren

Geschwindigkeit:

- fast wie C (mit guten Compilern und optimierter Programmierung)

Bemerkungen:

- beste Wahl für objektorientierte Numerik
- deutlich schwerer zu lernen als Java
- wesentlich kleinere Klassenbibliothek (z.B. keine Graphik)

2.4 Übungen

2.4.1 Übung 1

Übersetzen Sie die Beispielprogramme und vergleichen Sie die Laufzeiten.
Anmerkung: Insbesondere für die Java-Version empfiehlt sich, mit kleineren Matrizen (300x300) zu starten.

2.4.2 Übung 2

Erweitern Sie die Klasse Matrix (wahlweise in Java oder C++) um Funktionen zur Manipulation von Matrizen, z.B. einfache Matrixalgebra und Herausgreifen von Teilmatrizen. Legen Sie dabei besonderen Wert auf das Design der Schnittstelle und überlegen Sie in jedem Fall, welche Vor- und Nachteile Ihre Design-Entscheidungen haben, etwa in Bezug auf Einfachheit, Geschwindigkeit oder Erweiterbarkeit.

Anhang A

Sourcen der Beispielprogramme

A.1 FORTRAN77

A.1.1 Matrixmultiplikation

```
      PROGRAM MATMUL
C
C  MULTIPLIKATION VON MATRIZEN
C

      IMPLICIT LOGICAL(A-Z)

      INTEGER N
      PARAMETER (N = 1000)

      DOUBLE PRECISION   A(N, N), B(N,N), C(N,N)
      DOUBLE PRECISION   SPUR, SPUR1, CTH
      INTEGER I, J, K

C     ANFANGSWERTE SETZEN
      DO 100 I=1, N
         DO 110 J=1, N
            A(I,J) = I + J
            B(I,J) = I - J
 110     CONTINUE
         B(I,I) = 1.0
 100  CONTINUE

C     MATRIXMULTIPLIKATION
      DO 200 I = 1, N
         DO 210 J = 1, N
            C(I, J) = 0.0
            DO 220 K = 1, N
               C(I, J) = C(I, J) + A(I, K) * B(K, J)
 220        CONTINUE
```

```
 210      CONTINUE
 200   CONTINUE

C      TESTDATEN AUSGEBEN
       SPUR1 = N * (N+1)
       PRINT *, "SPUR THEORETISCH: ", SPUR1
       SPUR = 0.0
       DO 300 I = 1, N
          SPUR = SPUR + C(I,I)
 300   CONTINUE
       PRINT *, "SPUR            : ", SPUR

       CTH = 333331003.0D0
       PRINT *, "C(1,2) THEOR.   : ", CTH
       PRINT *, "C(1,2)          : ", C(1,2)

       END
```

A.1.2 Matrixmultiplikation mit Blas-Routine

```
       PROGRAM MATMUL
C
C  MULTIPLIKATION VON MATRIZEN
C

       IMPLICIT LOGICAL(A-Z)

       INTEGER N
       PARAMETER (N = 1000)

       DOUBLE PRECISION   A(N, N), B(N,N), C(N,N)

       DOUBLE PRECISION ALPHA, BETA
       INTEGER NN
       CHARACTER TA, TB

C      ANFANGSWERTE SETZEN
       CALL INIT(A, B, N)

C      MATRIXMULTIPLIKATION
       CALL DGEMM('N', 'N', N, N, N, 1.0D0, A, N, B, N, 0.0D0, C, N)

C      TESTDATEN AUSGEBEN
       CALL TEST(C, N)

       END
```

```
      SUBROUTINE INIT(A, B, N)

      INTEGER            N
      DOUBLE PRECISION   A(N,N), B(N,N)
      INTEGER            I, J

      DO I=1, N
         DO J=1, N
            A(I,J) = I + J
            B(I,J) = I - J
         ENDDO
         B(I,I) = 1.0
      ENDDO

      END


      SUBROUTINE TEST(C, N)

      DOUBLE PRECISION   C(N,N)
      INTEGER            N

      DOUBLE PRECISION   SPUR, SPUR1, CTH
      INTEGER            I

      SPUR1 = N * (N+1)
      PRINT *, "SPUR THEORETISCH: ", SPUR1
      SPUR = 0.0
      DO I = 1, N
         SPUR = SPUR + C(I,I)
      ENDDO
      PRINT *, "SPUR             : ", SPUR

      CTH = 333331003.0D0
      PRINT *, "C(1,2) THEOR.    : ", CTH
      PRINT *, "C(1,2)           : ", C(1,2)

      END
```

A.2 C

A.2.1 Matrixmultiplikation

```
/*
 *  Multiplikation von Matrizen
```

```
 */

#include <stdio.h>

#define N   300

int main(int argc, char* argv[])
{
  double   a[N][N], b[N][N], c[N][N];
  double   spur, spur1, cth;
  int      i,j,k;

  /* Anfangswerte setzen */
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      a[i][j] = i+j;
      b[i][j] = i-j;
    }
    b[i][i] = 1.0;
  }

  /* Matrixmultiplikation */
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      c[i][j] = 0.0;
      for (k=0; k<N; k++) {
c[i][j] += a[i][k] * b[k][j];
      }
    }
  }

  /* Testdaten ausgeben */
  spur1 = N * (N-1);
  printf("Spur theoretisch: %lf\n", spur1);
  spur = 0.0;
  for (i=0; i<N; i++) {
    spur += c[i][i];
  }
  printf("Spur            : %lf\n", spur);

  cth = 332334001.0;
  printf("c[0][1] theor.  : %lf\n", cth);
  printf("c[0][1]         : %lf\n", c[0][1]);
}
```

A.2.2 Matrixmultiplikation mit Blas-Routine

```
/*
 *  Multiplikation von Matrizen
 */

#include <stdio.h>

#define N   1000


void init(double a[N][N], double b[N][N]) {
  int   i,j;

  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      a[i][j] = i+j;
      b[i][j] = i-j;
    }
    b[i][i] = 1.0;
  }
}


void test(double c[N][N]) {
  int      i;
  double   spur, spur1, cth;

  spur1 = N * (N-1);
  printf("Spur theoretisch: %lf\n", spur1);

  spur = 0.0;
  for (i=0; i<N; i++) {
    spur += c[i][i];
  }
  printf("Spur             : %lf\n", spur);

  cth = 332334001.0;
  printf("c[0][1] theor.   : %lf\n", cth);
  printf("c[0][1]          : %lf\n", c[0][1]);
}


int main(int argc, char* argv[])
{
  double   a[N][N], b[N][N], c[N][N];
  int      i,j,k;
```

```
  /* Anfangswerte setzen */
  init(a, b);

  /* Matrixmultiplikation */
  {
    int     n     = N;
    char    trans = 'N';
    double  alpha = 1.0;
    double  beta  = 0.0;

    dgemm_(&trans, &trans, &n, &n, &n, &alpha, b, &n, a, &n, &beta, c, &n);
  }

  /* Testdaten ausgeben */
  test(c);
}
```

A.3 Fortran90

A.3.1 Matrixmultiplikation

Hauptprogramm

```
program matmulf90

!  Multiplikation von Matrizen

  use kinds
  implicit none

  integer, parameter ::  n = 1000

  real(kind=REAL8), dimension(n, n) :: a, b, c
  real(kind=REAL8)                  :: spur, spur1, cth
  integer                       :: i, j, k


  ! Anfangswerte setzen
  do i=1, n
     do j=1, n
        a(i,j) = i + j
        b(i,j) = i - j
     enddo
     b(i,i) = 1.0
  enddo
```

```
  ! Matrixmultiplikation
  c = matmul(a, b)

  ! Testdaten ausgeben
  spur1 = n * (n+1)
  print *, "Spur theoretisch: ", spur1
  spur = 0.0
  do i = 1, n
     spur = spur + c(i,i)
  enddo
  print *, "Spur            : ", spur

  cth = 333331003.0D0
  print *, "c(1,2) theor.   : ", cth
  print *, "c(1,2)          : ", c(1,2)

end program matmulf90
```

Modul KIND

```
!
! portable data types
!

module kinds
  implicit none

  integer, parameter :: INT1  = selected_int_kind(2)   ! Single byte integer
  integer, parameter :: INT2  = selected_int_kind(4)   ! Two byte integer
  integer, parameter :: INT4  = selected_int_kind(9)   ! Four byte integer

  integer, parameter :: REAL4 = selected_real_kind(5)  ! Single precision
  integer, parameter :: REAL8 = selected_real_kind(12) ! Double precision

  integer, parameter :: COMPLEX8  = REAL4  ! complex kind from components
  integer, parameter :: COMPLEX16 = REAL8

end module kinds
```

A.3.2 Matrixmultiplikation mit Blas-Routine

Hauptprogramm

```
module mysubs

contains
```

```
  subroutine init(a, b)
    use kinds
    implicit none

    real(kind=REAL8), dimension(:,:), intent(out) :: a, b
    integer :: n, i, j

    n = size(a,1)
    do i=1, n
       do j=1, n
          a(i,j) = i + j
          b(i,j) = i - j
       enddo
       b(i,i) = 1.0
    enddo

  end subroutine init


  subroutine test(c)
    use kinds
    implicit none

    real(kind=REAL8), dimension(:,:), intent(in) :: c
    integer          :: n, i
    real(kind=REAL8) :: spur, spur1, cth

    n = size(c,1)
    spur1 = n * (n+1)
    print *, "Spur theoretisch: ", spur1
    spur = 0.0
    do i = 1, n
       spur = spur + c(i,i)
    enddo
    print *, "Spur            : ", spur

    cth = 333331003.0D0
    print *, "c(1,2) theor.   : ", cth
    print *, "c(1,2)          : ", c(1,2)

  end subroutine test

end module mysubs


program matmulblas
```

```
  ! Multiplikation von Matrizen

  use kinds
  use blas
  use mysubs
  implicit none

  integer, parameter                :: n = 1000
  real(kind=REAL8), dimension(n, n) :: a, b, c

  ! Anfangswerte setzen
  call init(a, b)

  ! Matrixmultiplikation
  call gemm(a, b, c)

  ! Testdaten ausgeben
  call test(c)

end program matmulblas
```

Modul BLAS

```
!
! simple version of dgemm
!
module blas
  use kinds
  implicit none

contains

  subroutine gemm(a, b, c)
    real(kind=REAL8), dimension(:,:), intent(in)    :: a, b
    real(kind=REAL8), dimension(:,:), intent(inout) :: c

    integer          :: n, m, k

    n = size(a,1)
    k = size(a,2)
    m = size(b,2)

    call dgemm('n', 'n', n, k, m, 1.0D0, a, n, b, k, 0.0D0, c, n)

  end subroutine gemm

end module blas
```

A.3.3 Rationale Zahlen als abstrakter Datentyp

```
module rational_arithmetic

  type Ratio
     integer  :: num, den
  end type Ratio

  private cancel

  interface operator(*)
     module procedure ratmul
  end interface

contains

  type(Ratio) function ratmul(r1, r2)
    implicit none
    type(Ratio), intent(in) ::  r1, r2

    ratmul%num = r1%num * r2%num
    ratmul%den = r1%den * r2%den
    call cancel(ratmul)

  end function ratmul


  subroutine cancel(rat)
    implicit none
    type(Ratio), intent(inout) ::  rat

    integer u, v, r

    u = abs(rat%num)
    v = abs(rat%den)

    do
       if (v == 0) exit
       r = mod(u, v)
       u = v
       v = r
    end do

    rat%num = rat%num / u
    rat%den = rat%den / u

  end subroutine cancel
```

```
end module rational_arithmetic


program ratiofarm
  use rational_arithmetic

  type(Ratio)  :: a, b, c

  a = Ratio(2,5)
  b = Ratio(1,6)
  c = a * b
  print*, c

end program ratiofarm
```

A.4 Java

A.4.1 Matrixmultiplikation

Klasse Matrix

```
/** Matrix:
 *  matrix of doubles
 *  secure version with checks
 *  organized like a Fortran array:
 *    indices running from 1 to n
 *    stored in column order
 */

public class Matrix {

  protected int   nRows, nCols;    // number of colums and rows
  protected double[]  val;         // vector of values

  /** creates a new matrix with given number of rows and columns */
  public Matrix(int rows, int cols) {
    if (rows < 0 || cols < 0) {
      // throw an exception
      throw new NegativeArraySizeException(
"Invalid array size: " + rows + " x " + cols);
    }

    nRows = rows;
    nCols = cols;
    val = new double[nRows * nCols];
  }
```

```
  /** returns shape (= number of rows and columns) of the matrix */
  public int[] shape() {
    int[] sizes = new int[2];
    sizes[0] = nRows;
    sizes[1] = nCols;
    return sizes;
  }

  /** sets Matrix(i,j) to v */
  public void set(int i, int j, double v) {
    // check indices
    if (i > 0 && i <= nRows && j > 0 && j <= nCols) {
      val[i - 1 + nRows * (j - 1)] = v;
    } else {
      throw new IndexOutOfBoundsException(
        "Invalid index: (" + i + "," + j + ")" );
    }
  }

  /** returns Matrix(i,j) */
  public double get(int i, int j) {
    // check indices
    if (i > 0 && i <= nRows && j > 0 && j <= nCols) {
      return val[i - 1 + nRows * (j - 1)];
    } else {
      throw new IndexOutOfBoundsException(
"Invalid index: (" + i + "," + j + ")" );
    }
  }

  /** result = matmul(this, b) */
  public void mult(Matrix b, Matrix result) {
    int[] dimA = shape();
    int[] dimB = b.shape();
    int[] dimC = result.shape();

    // check dimensions
    if (dimA[1] != dimB[0]) {
      throw new IllegalArgumentException(
        "Invalid matrix dimensions for multiplication: ("
  + dimA[0] + "," + dimA[1] + ") x ("
  + dimB[0] + "," + dimB[1] + ")"  );
    } else if (dimA[0] != dimC[0] || dimB[1] != dimC[1]) {
      throw new IllegalArgumentException(
        "Invalid matrix dimensions for result: is ("
  + dimC[0] + "," + dimC[1] + "), should be ("
```

```
  + dimA[0] + "," + dimB[1] + ")" );
    }

    int m = dimA[0];
    int l = dimA[1];
    int n = dimB[1];

    for (int i = 1; i <= m; i++) {
      for (int j = 1; j <= n; j++) {
result.val[i - 1 + m * (j - 1)] = 0.0;
for (int k = 1; k <= l; k++) {
  result.val[i - 1 + m * (j - 1)] +=
    val[i - 1 + m * (k - 1)] * b.val[k - 1 + l * (j - 1)];
}
      }
    }
  }
}
```

Klasse SquareMatrix

```
/** SquareMatrix:
 *    quadratic matrix of doubles
 */

public class SquareMatrix extends Matrix {

  /** creates an n x n square matrix */
  public SquareMatrix(int n) {
    super(n,n);
  }

  /** matrix multiplication of square matrices gives a square matrix */
  public void mult(SquareMatrix b, SquareMatrix result) {
    super.mult(b, result);
  }

  /** returns the trace of a square matrix */
  public double trace() {
    double tr = 0.0;
    for (int i = 0; i < nRows; i++) {
      tr += val[i + nRows * i];
    }
    return tr;
  }
}
```

Klasse TestMatrix

```
/** TestMatmul:
 *   simple class to test the matrix multiplikation
 */

public class TestMatrix {

  private SquareMatrix a, b, c;

  public TestMatrix(int n) {
    a = new SquareMatrix(n);
    b = new SquareMatrix(n);
    c = new SquareMatrix(n);

    // initialize a and b
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
a.set(i, j, i + j);
b.set(i, j, i - j);
      }
      b.set(i, i, 1.0);
    }
  }

  public void run() {
    a.mult(b, c);
  }

  public void check() {
    int n = a.shape()[0];

    double sp_th = n * (n + 1);
    double sp = c.trace();

    double c12_th = (n * n * n - 7 * n + 9) / 3.0;
    double c12 = c.get(1, 2);

    System.out.println("trace (expected)  : " + sp_th);
    System.out.println("trace             : " + sp);
    System.out.println("c(1,2) (expected) : " + c12_th);
    System.out.println("c(1,2)            : " + c12);
  }

  static public void main(String[] args) {
    TestMatrix tm = new TestMatrix(300); // initializes everything
    tm.run();
```

```
    tm.check();
  }
}
```

A.5 C++

A.5.1 Matrixmultiplikation

Klasse Matrix, Schnittstelle

```
/* Matrix:
 *  matrix of doubles
 *  secure version with checks
 *  organized like Fortran array:
 *    indices running from 1 to n
 *    stored in column order
 */

class Matrix {

  public:

  // creates a new matrix with given number of rows and columns
  Matrix(int rows, int cols);

  // delete a matrix
  ~Matrix();

  // returns shape (= number of rows and columns) of the matrix
  int* shape();

  // sets Matrix(i,j) to v
  void set(int i, int j, double v);

  // returns Matrix(i,j)
  double get(int i, int j);

  // result = matmul(this, b)
  void mult(Matrix &b, Matrix &result);

  protected:

  int  nRows;
  int  nCols;         // number of colums and rows
  double  *val;       // vector of values
};
```

Klasse Matrix, Implementierung

```
/* Matrix.C:
 *  matrix of doubles
 *  implementation
 */

#include "Matrix.H"
#include <strstream.h>

// creates a new matrix with given number of rows and columns
Matrix::Matrix(int rows, int cols) {

  // error handling
  if (rows < 0 || cols < 0) {
    ostrstream myString;
    myString << "Invalid array size: " << rows << " x " << cols ;
    throw myString.str();
  }

  nRows = rows;
  nCols = cols;
  val = new double[nRows * nCols];
}

Matrix::~Matrix() {
  delete[] val;
}

// returns shape (= number of rows and columns) of the matrix
int* Matrix::shape() {
  int * sizes = new int[2];
  sizes[0] = nRows;
  sizes[1] = nCols;
  return sizes;
}

// sets Matrix(i,j) to v
void Matrix::set(int i, int j, double v) {
  // check indices
  if (i > 0 && i <= nRows && j > 0 && j <= nCols) {
    val[i - 1 + nRows * (j - 1)] = v;
  } else {
    ostrstream myString;
    myString << "Invalid index: (" << i << "," << j << ")";
    throw myString.str();
  }
```

```
}

// returns Matrix(i,j)
double Matrix::get(int i, int j) {
  // check indices
  if (i > 0 && i <= nRows && j > 0 && j <= nCols) {
    return val[i - 1 + nRows * (j - 1)];
  } else {
    ostrstream myString;
    myString << "Invalid index: (" << i << "," << j << ")";
    throw myString.str();
  }
}

// result = matmul(this, b)
void Matrix::mult(Matrix &b, Matrix &result) {
  int  *dimA, *dimB, *dimC;
  dimA = shape();
  dimB = b.shape();
  dimC = result.shape();

  // check dimensions
  if (dimA[1] != dimB[0]) {
    ostrstream myString;
    myString << "Invalid matrix dimensions for multiplication: ("
     << dimA[0] << "," << dimA[1] << ") x ("
     << dimB[0] << "," << dimB[1] << ")";
    throw myString.str();
  } else if (dimA[0] != dimC[0] || dimB[1] != dimC[1]) {
    ostrstream myString;
    myString << "Invalid matrix dimensions for result: is ("
     << dimC[0] << "," << dimC[1] << "), should be ("
     << dimA[0] << "," << dimB[1] << ")";
    throw myString.str();
  }

  int m = dimA[0];
  int l = dimA[1];
  int n = dimB[1];

  for (int i = 1; i <= m; i++) {
for (int j = 1; j <= n; j++) {
  result.val[i - 1 + m * (j - 1)] = 0.0;
  for (int k = 1; k <= l; k++) {
result.val[i - 1 + m * (j - 1)] +=
  val[i - 1 + m * (k - 1)] * b.val[k - 1 + l * (j - 1)];
  }
```

```
}
  }
}
```

Klasse SquareMatrix, Schnittstelle

```
// SquareMatrix:
//    quadratic matrix of doubles


#include "Matrix.H"

class SquareMatrix : public Matrix {

  public:

  // creates an n x n square matrix
  SquareMatrix(int n);

  // delete a SquareMatrix
  ~SquareMatrix();

  // matrix multiplication of square matrices gives a square matrix
  void mult(SquareMatrix &b, SquareMatrix &result);

  // returns the trace of a square matrix
  double trace();
};
```

Klasse SquareMatrix, Implementierung

```
// SquareMatrix:
//    quadratic matrix of doubles
//    implementation

#include "SquareMatrix.H"

// creates an n x n square matrix
SquareMatrix::SquareMatrix(int n) : Matrix(n,n) {
}

// delete a SquareMatrix
SquareMatrix::~SquareMatrix() {
  delete[] val;
}
```

```
// matrix multiplication of square matrices gives a square matrix
void SquareMatrix::mult(SquareMatrix &b, SquareMatrix &result) {
  Matrix::mult(b, result);
}

// returns the trace of a square matrix
double SquareMatrix::trace() {
  double tr = 0.0;
  for (int i = 0; i < nRows; i++) {
tr += val[i + nRows * i];
  }
  return tr;
}
```

Klasse TestMatrix

```
/** TestMatmul:
 *   simple class to test the matrixmultiplikation
 */

#include <iostream.h>
#include "SquareMatrix.H"

class TestMatrix {

  public:

  TestMatrix(int n) : a(n), b(n), c(n) {
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
a.set(i, j, i + j);
b.set(i, j, i - j);
      }
      b.set(i, i, 1.0);
    }
  }

  void run() {
    a.mult(b, c);
  }

  void check() {
    int n = a.shape()[0];
```

```
    double sp_th = n * (n + 1);
    double sp = c.trace();

    double c12_th = (n * n * n - 7 * n + 9) / 3.0;
    double c12 = c.get(1, 2);

    cout << "Spur theoretisch: " << sp_th  << endl;
    cout << "Spur            : " << sp     << endl;
    cout << "c(1,2) theor.   : " << c12_th << endl;
    cout << "c(1,2)          : " << c12    << endl;
  }

  private:

  SquareMatrix a, b, c;
};


int main(int argc, char* argv[]) {
  try {
    TestMatrix tm(500); // initialisiert alles
    tm.run();
    tm.check();
  } catch (char* info) {
    cerr << "EXCEPTION: " << info << endl;
    return -1;
  }
}
```