



Wintersemester 2007/08

**Einführung in die Informatik für
Naturwissenschaftler und Ingenieure
(alias Einführung in die Programmierung)
(Vorlesung)**

Prof. Dr. Günter Rudolph
Fakultät für Informatik
Lehrstuhl für Algorithm Engineering



Inhalt

- Zeiger für dynamischen Speicher
- Anwendungen



Aufgabe:

Lese zwei Vektoren reeller Zahlen der Länge n ein.

$$a = (a_1, \dots, a_n)' \quad b = (b_1, \dots, b_n)'$$

Berechne das Skalarprodukt ... $\sum_{i=1}^n a_i \cdot b_i$

... und gebe den Wert auf dem Bildschirm aus!

Lösungsansatz:

Vektoren als Arrays von Typ `double`.

n darf höchstens gleich der Arraygröße sein! Testen und ggf. erneute Eingabe!



```
#include <iostream>
using namespace std;

int main() {
    unsigned int const nmax = 100;
    unsigned int i, n;
    double a[nmax], b[nmax];

    // Dimension n einlesen und überprüfen
    do {
        cout << "Dimension ( n < " << nmax << " ): ";
        cin >> n;
    } while ( n < 1 || n > nmax);
```

(Fortsetzung folgt ...)

Datendefinition `double a[nmax]` OK,
weil `nmax` eine Konstante ist!

Ohne `const`: Fehlermeldung!
z.B. „Konstanter Ausdruck erwartet“

Intermezzo: Sinn & Zweck des ISO Standards für C++

Der aktuelle GNU C++ Compiler erlaubt folgendes:

```
#include <iostream>
int main() {
    int n;
    std::cin >> n;
    double a[n];
    a[0] = 3.14;
    return 0;
}
```

Aber: Der Microsoft C++ Compiler (VS 2003) meldet einen Fehler!

Variable Arraygrenzen sind nicht Bestandteil des C++ Standards!

Verwendung von Compiler-spezifischen Spracherweiterungen führt zu Code, der **nicht portabel** ist! ☹️ 🚫 ☠️

Das ist nicht wünschenswert!

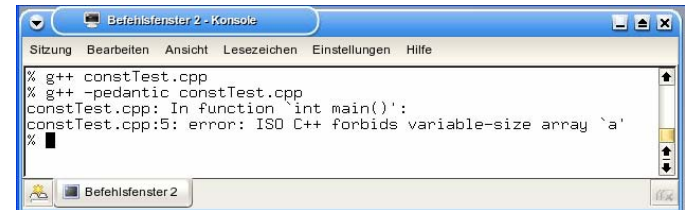
Intermezzo: Sinn & Zweck des ISO Standards für C++

```
#include <iostream>
int main() {
    int n;
    std::cin >> n;
    double a[n];
    a[0] = 3.14;
    return 0;
}
```

Also: Bei Softwareentwicklung nur Sprachelemente des C++ Standards verwenden!

Bei GNU Compiler: Option `-pedantic`

C++ Standard ISO/IEC 14882-2003
z.B. als PDF-Datei erhältlich für 30\$
<http://webstore.ansi.org/>



```
Befehlsfenster 2 - Konsole
Sitzung Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
% g++ constTest.cpp
% g++ -pedantic constTest.cpp
constTest.cpp: In function 'int main()':
constTest.cpp:5: error: ISO C++ forbids variable-size array `a`
% █
```

Kapitel 4: Zeiger

(... Fortsetzung)

```
// Vektor a einlesen
for (i = 0; i < n; i++) {
    cout << "a[" << i << "]=" << " ";
    cin >> a[i];
}
// Vektor b einlesen
for (i = 0; i < n; i++) {
    cout << "b[" << i << "]=" << " ";
    cin >> b[i];
}
// Skalarprodukt berechnen
double sp = 0.;
for (i = 0; i < n; i++)
    sp += a[i] * b[i];
// Ausgabe
cout << "Skalarprodukt = " << sp << endl;
return 0;
}
```

Anmerkung:

Fast identischer Code!

Effizienter mit Funktionen!

→ nächstes Kapitel!

Kapitel 4: Zeiger

Unbefriedigend bei der Implementierung:

Maximale festgelegte Größe des Vektors!

→ Liegt an der unterliegenden Datenstruktur Array:

Array ist **statisch**, d.h.

Größe wird zur Übersetzungszeit **festgelegt** und ist **während der Laufzeit** des Programms **nicht veränderbar!**

Schön wären **dynamische** Datenstrukturen, d.h.

Größe wird zur Übersetzungszeit **nicht festgelegt** und ist **während der Laufzeit** des Programms **veränderbar!**

Das geht mit **dynamischem Speicher** ...

... und **Zeigern!**

Exkurs: Dynamischer Speicher

Erzeugen und Löschen eines Objekts zur Laufzeit:

1. Operator `new` erzeugt Objekt
2. Operator `delete` löscht zuvor erzeugtes Objekt

Beispiel: (Erzeugen)

```
int *xp = new int;
double *yp = new double;

struct PunktT {
    int x, y;
};

PunktT *pp = new PunktT;
```

Beispiel: (Löschen)

```
delete xp;
delete yp;

delete pp;
```

```
int n = 10;
int *xap = new int[n];
PunktT *pap = new PunktT[n];

delete[] xap;
delete[] pap;
```

variabel, nicht konstant!

Exkurs: Dynamischer Speicher

Bauplan:

```
Datentyp *Variable = new Datentyp;      (Erzeugen)
delete Variable;                        (Löschen)
```

Bauplan für Arrays:

```
Datentyp *Variable = new Datentyp[Anzahl]; (Erzeugen)
delete[] Variable; (Löschen)
```

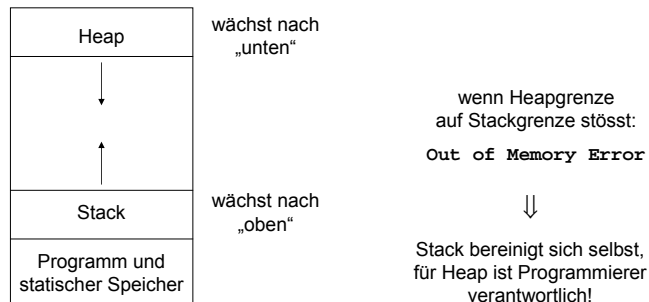
Achtung:

Dynamisch erzeugte Objekte müssen auch wieder gelöscht werden!
Keine automatische Speicherbereinigung!

Exkurs: Dynamischer Speicher

Wo wird Speicher angelegt?

⇒ im **Freispeicher** alias **Heap** alias **dynamischen Speicher**



Kapitel 4: Zeiger

Zurück zur **Beispielaufgabe:**

```
unsigned int const nmax = 100;
unsigned int i, n;
double a[nmax], b[nmax];

do {
    cout << "Dimension ( n < " << nmax << " ): ";
    cin >> n;
} while ( n < 1 || n > nmax);
```

vorher:
statischer Speicher

```
unsigned int i, n;
double *a, *b;

do {
    cout << "Dimension: ";
    cin >> n;
} while ( n < 1);
a = new double[n];
b = new double[n];
```

nachher:
dynamischer Speicher

Kapitel 4: Zeiger

Nicht vergessen:

Am Ende angeforderten dynamische Speicher wieder freigeben!

```
delete[] a;
delete[] b;
return 0;
}
```

Sonst „Speicherleck“!

⇒ Programm terminiert möglicherweise anormal mit Fehlermeldung!

Kapitel 4: Zeiger

Beispiel für programmierten Absturz:

```
#include <iostream>
using namespace std;

int main() {
    unsigned int const size = 100 * 1024;
    unsigned short k = 0;

    while (++k < 5000) {
        double* ptr = new double[size];
        cout << k << endl;
        // delete[] ptr;
    }
    return 0;
}
```

bei $k \approx 2500$ sind
2 GB erreicht
⇒ Abbruch! ⚡

Kapitel 4: Zeiger

Projekt: Matrix mit dynamischem Speicher (Größe zur Laufzeit festgelegt!)

Vorüberlegungen:

Speicher im Rechner ist **linear**!

⇒ Rechteckige / flächige Struktur der Matrix linearisieren!



n Zeilen, m Spalten $\Rightarrow n \times m$ Speicherplätze!

Kapitel 4: Zeiger

Projekt: Matrix mit dynamischem Speicher (Größe zur Laufzeit festgelegt!)

```
double **matrix;
matrix = new double*[zeilen];
matrix[0] = new double[zeilen * spalten];
for (i = 1; i < zeilen; i++)
    matrix[i] = matrix[i-1] + spalten;
```

Zugriff wie beim
zweidimensionalen
statischen Array:
 $matrix[2][3] = 2.3;$

