Technische Universität München
Institut für Informatik
Dr. rer. nat. Tobias Weinzierl
Kaveh Rahnema

März 2013

# Kompaktkurs Einführung in die Programmierung

# **Übungsblatt 8:** Strukturtypen, Speicher reservieren

## Lernziele

- Verwendung von Strukturen.
- Umgang mit mehrdimensionalen Feldern (Arrays).

## 1) Strukturtypen zum Üben (T)

Gegeben sei die Strukturdefinition aus der Vorlesung

```
struct bestellung {
    int Anzahl;
    double Preis;
};
```

Was ist nach Ausführung der folgenden Anweisungen jeweils der Inhalt der Strukturvariablen und von `summe`?

i) Der Punkt-Operator:

```
struct bestellung b1, b2, b3;
double summe;
b1.Anzahl = 1;
b2.Anzahl = 2;
b3.Anzahl = b2.Anzahl;
b1.Preis = 10.0;
b2.Preis = 20.0;
b3.Preis = b1.Preis + b2.Preis;
summe =   b1.Anzahl*b1.Preis
        + b2.Anzahl*b2.Preis
+ b3.Anzahl*b3.Preis;
```

ii) Der Operator `->`:

```
struct bestellung b1, b2, *bp;
double summe = 100;
bp = &b1;
(*bp).Anzahl = 5;
b1.Preis = 10.0;
bp = &b2;
bp -> Anzahl = 25;
bp -> Preis = (summe - b1.Anzahl*b1.Preis) / bp->Anzahl;
```

## 2) Zeichenketten kopieren (P)

Schreiben Sie eine Funktion

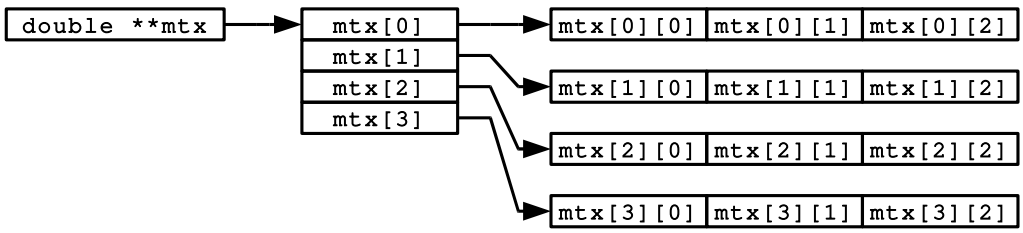
```
char *copy(char *s)
```

die die übergebene Zeichenkette kopiert. Dazu soll sie zunächst Speicherplatz für eine neue Zeichenkette passender Größe reservieren (mit `new` bzw. `malloc`), dann die Zeichen von `s` dorthin kopieren (vergessen Sie nicht das '`\0`'-Zeichen am Ende!) und einen Zeiger auf das erste Zeichen der neuen Zeichenkette zurückgeben.

## 3) Matrix-Vektor-Multiplikation (P)

Matrizen lassen sich gut mit zweidimensionalen Feldern realisieren, diese wiederum, um hinsichtlich der Bereichsgrenzen flexibel zu bleiben, über Zeiger und mittels `new` bzw. `malloc` dynamisch reserviertem Speicher (im Gegensatz zu Deklarationen wie z.B. `double[10][10]`).

Für ein zweidimensionales Feld wird zunächst ein Feld von Zeigern angelegt (z.B. ein Zeiger pro „Zeile“ des Feldes). Jeder Zeiger dieses Feldes wird dann auf ein weiteres Feld (für die jeweilige Zeile) gesetzt:




Schreiben Sie ein Programm, das Matrizen und Vektoren anlegt, eine Matrix-Vektor-Multiplikation durchführt sowie entsprechende Augaben aufruft.

i) Schreiben Sie dazu zwei Funktionen

```
double *null_vector(int n)
double **null_matrix(int n)
```

die nach obigem Schema einen Vektor der Länge $n$ bzw. eine $n \times n$-Matrix erzeugen, alle Elemente auf 0 setzen und als Ergebnis einen entsprechenden Zeiger liefern.

ii) Schreiben Sie Funktionen

```
double *test_vector(int n)
double **test_matrix(int n)
```

die ebenfalls eine Matrix bzw. einen Vektor erzeugen (dazu können Sie die Funktionen aus dem ersten Teil verwenden), aber zusätzlich die Elemente mit geeigneten Werten versehen (was das ist, bleibt Ihnen überlassen, aber man soll sich später damit von der Funktion Ihrer Multiplikation überzeugen können, 0 ist also nicht geeignet.)

iii) Schreiben Sie Funktionen

```
void print_vector(double *vec, int n)
void print_matrix(double **mtx, int n)
```

die eine Matrix bzw. einen Vektor ausdrucken. Die Größe wird dabei im Parameter `n` übergeben. Wenn Sie Wert auf Ästhetik legen, können Sie sich dazu kundig machen, wie man mit der Bibliotheksfunktion `printf()` in C bzw. über `std::cout` in C++ die Ausgabe so formatiert, dass die Elemente einer Spalte untereinander stehen.

iv) Schreiben Sie eine Funktion

```
double *mult(double **mtx, double *vec, int n)
```

die die Multiplikation einer Matrix `mtx` mit einem Vektor `vec` durchführt. Deren Größe wird wieder mit `n` übergeben. Der Rückgabewert soll der Ergebnisvektor sein, d.h. ein Zeiger auf das (neu erzeugte) Feld.

v) Schreiben Sie ein Hauptprogramm, das die entstandenen Funktionen in geeigneter Weise aufruft.

vi) ★ Achten Sie darauf, dass keine Speicherlecks entstehen indem Sie Daten die auf dem Freispeicher alloziiert wurden geeignet wieder freigeben.

## 4) Dichte Matrix-Matrix-Multiplikation (★)

Ziel dieser Aufgabe ist es eine Multiplikation zweier dichtbesetzter Matrizen

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} \tag{1}$$

beliebiger Dimension mit Hilfe von OpenMP zu parallelisieren.

i) Realisieren Sie eine geeignete Initialisierung Ihrer Matrizen $A \in \mathbb{N}^{n \times n}$, $B \in \mathbb{N}^{n \times n}$ und $C \in \mathbb{N}^{n \times n}$.

ii) Implementieren Sie das Matrix-Matrix Produkt (1) zunächst ohne OpenMP, d.h. eine sequentielle Version.

iii) Implementieren Sie nun eine zusätzliche Variante von (1), dessen Berechnung mit Hilfe von OpenMP parallel abläuft. Die Parallelisierung soll hierbei entlang der Zeilen geschehen, d.h. jeder Thread soll für einen (zusammenhängenden) Teil von Zeilen von $C$ verantwortlich sein.

iv) Implementieren Sie Zeitmessungen für die sequentielle und parallele Version ihrer Produkte. Können Sie Unterschiede bzw. Geschwindigkeitsvorteile erkennen? Wählen Sie z.B. als Dimension $n = 1000$.