Dirk Praetorius,
Thomas Führer

Sommersemester 2014
11.04.2014

Übungen zur Vorlesung
Einführung in das Programmieren für TM

Serie 6

Aufgabe 6.1. Schreiben Sie Funktionen `int countValueInRow(int** matrix, int m, int n, int val, int row)` und `int countValueInColumn(int** matrix, int m, int n, int val, int col)`, die zu einer gegebenen $m \times n$-Matrix die Anzahl der Einträge in der gegebenen Zeile/Spalte zurückliefern, die mit `val` übereinstimmen. Speichern Sie den Source-Code unter `countValueInRowCol.c` in das Verzeichnis `serie06`.

Aufgabe 6.2. Die Frobeniusnorm einer Matrix $A \in \mathbb{R}^{m\times n}$ ist durch

$$\|A\|_F := \Big(\sum_{j=1}^{m}\sum_{k=1}^{n} A_{jk}^2\Big)^{1/2}$$

definiert. Schreiben Sie eine Funktion `frobeniusnorm`, die für gegebene Matrix A und gegebene Dimensionen $m, n \in \mathbb{N}$ die Frobeniusnorm berechnet. Schreiben Sie ferner ein aufrufendes Hauptprogramm, in dem die Zeilen- und Spaltendimensionen $m, n \in \mathbb{N}$ und A eingelesen werden und $\|A\|_F$ ausgegeben wird. Die Matrix A soll dabei als dynamische Matrix (vom Typ `double**`) realisiert werden. Speichern Sie den Source-Code unter `frobeniusnorm.c` in das Verzeichnis `serie06`.

Aufgabe 6.3. In vielen mathematischen Bibliotheken werden Matrizen $A \in \mathbb{R}^{m\times n}$ spaltenweise gespeichert, d.h. in Form eines Vektors $a \in \mathbb{R}^{mn}$, wobei $a_{j+km} = A_{jk}$ gilt, wenn die Indizierung (wie in C üblich) bei 0 beginnt. Für eine Matrix $A \in \mathbb{R}^{m\times n}$ ist die Zeilensummennorm durch

$$\|A\| = \max_{j=1,\dots,m} \sum_{k=1}^{n} |A_{jk}|$$

gegeben. Schreiben Sie eine Funktion `zeilensummennorm`, die die Zeilensummennorm einer Matrix A berechnet, die spaltenweise gespeichert ist. Schreiben Sie ein aufrufendes Hauptprogramm, in dem A eingelesen und $\|A\|$ ausgegeben wird. Benutzen Sie ein dynamisches Array zur Speicherung der Matrix. Speichern Sie den Source-Code unter `zeilensummennorm.c` in das Verzeichnis `serie06`.

Aufgabe 6.4. Schreiben Sie eine Funktion `merge`, die zwei aufsteigend sortierte Felder $a \in \mathbb{R}^m$ und $b \in \mathbb{R}^n$ so vereinigt, dass das resultierende Feld $c \in \mathbb{R}^{m+n}$ ebenfalls aufsteigend sortiert ist, z.B. soll $a = (1, 3, 3, 4, 7)$ und $b = (1, 2, 3, 8)$ als Ergebnis $c = (1, 1, 2, 3, 3, 3, 4, 7, 8)$ liefern. Dabei soll ausgenutzt werden, dass die Felder a und b bereits sortiert sind. Schreiben Sie die Funktion so, das neben dem Base-Pointer des Vektors c die Längen $m, n \in \mathbb{N}$ übergeben werden. Bei Übergabe gelte $c_j = a_j$ für $j = 0, \dots, m-1$ und $c_j = b_{j-m}$ für $j = m, \dots, m+n-1$, d.h. bei Eingabe gilt $c = (a, b)$. Der Vektor c soll dann geeignet überschrieben werden. In der Funktion darf temporärer Speicher der Länge $m+n$ angelegt werden. Schreiben Sie ein aufrufendes Hauptprogramm, in dem $m, n \in \mathbb{N}$ sowie $a \in \mathbb{R}^m$ und $b \in \mathbb{R}^n$ eingelesen werden und $c \in \mathbb{R}^{m+n}$ ausgegeben wird. Speichern Sie den Source-Code unter `merge.m` in das Verzeichnis `serie06`.

Aufgabe 6.5. Schreiben Sie eine rekursive Funktion `mergesort`, die ein Feld a aufsteigend sortiert und das sortierte Feld zurückgibt. Gehen Sie dabei nach folgender Strategie vor:

- Hat a Länge ≤ 2, so wird das Feld a explizit sortiert.
- Hat a Länge > 2, halbiert man a in zwei Teilfelder b und c. Man ruft rekursiv `mergesort` für b und c auf und vereinige die sortierten Teilfelder mittels `merge` aus Aufgabe 6.4.

Machen Sie sich das Vorgehen anhand des Beispiels $a = (1, 3, 5, 2, 7, 1, 1, 3)$ klar. Testen Sie das Programm entsprechend.
Bemerkung: Falls a die Länge $2n+1$ für $n \geq 1$ hat, dann teilt man a in die zwei Teilfelder b und c, wobei b die Länge $n+1$ und c die Länge n hat. Für diese Aufgabe können Sie *Pointer-Arithmetik* verwenden, d.h. falls `a` ein Feld und `p` ein Pointer ist, welcher die Adresse von `a[k]` enthält, dann ist `p+n` die Adresse von `a[k+n]` (d.h. `*(p+n)` stimmt mit `a[k+n]` überein). Man beachte, dass `a` der Base-Pointer ist, der die Adresse von `a[0]` enthält.

Aufgabe 6.6. Genauso wie der Inhalt von Variablen elementaren Datentyps kann auch der Inhalt eines Pointers mittels `printf` ausgegeben werden. Man verwendet hier `%p` als Platzhalter für Adressen. Die Ausgabe dafür erfolgt systemabhängig meist in Hexadezimaldarstellung. Schreiben Sie eine Funktion `void charPointerAbstand(char* anfangsadresse, char* endadresse)`, welche folgende drei Werte tabelliert:

- Anfangsadresse
- Endadresse
- Abstand (Differenz) der beiden Adressen (Platzhalter im `printf` beachten!)

Da Arrays zusammenhängend im Speicher liegen, entspricht der Abstand zweier aufeinanderfolgender Elemente genau dem Speicherverbrauch des entsprechenden Datentyps. Testen Sie Ihre Funktion für einen `char`-Array `c[2]` mit den beiden Aufrufen:

```
charPointerAbstand(&c[0],&c[1]);
charPointerAbstand(c,c+1);
```

Schreiben Sie nun nach obiger Manier eine Funktion `void doublePointerAbstand(double* anfangsadresse, double* endadresse)`, testen diese mit einem `double`-Array und vergleichen die unterschiedlichen Ergebnisse.
Optional: Finden Sie heraus, wieviel Speicher die Typen `short`, `int` und `long` auf dem Übungsserver verbrauchen.

Aufgabe 6.7. Wo liegen die Fehler im folgenden Programm?

```
#include <stdio.h>

void square(double* x)
{
  double* y;
  x=(*y)*(*x);
}

int main(){
  double x=2.1;
  square(&x);
  printf("x^2=%f\n",x);
  return 0;
}
```

Verändern Sie *nur* die Funktion `square`, so dass der Output des Codes den Erwartungen entspricht.

Aufgabe 6.8. Was ist der Unterschied und der Zusammenhang zwischen einer Variable und einem Pointer? Was könnten Vor- und Nachteile dieser Konstrukte sein?
Schreiben Sie eine Funktion `swap`, welche die Werte zweier Variablen `x` und `y` vertauscht. Warum funktioniert das folgende Vorgehen nicht?

```
void swap(double x, double y)
{
   double tmp;
   tmp = x;
   x = y;
   y = tmp;
}
```