Dirk Praetorius,
Gregor Gantner

Sommersemester 2018
08.05.2018

Übungen zur Vorlesung
Einführung in das Programmieren für TM

Serie 7

Aufgabe 7.1. 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 ein `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. Finden Sie weiters heraus, wieviel Speicher die Typen `short`, `int` und `long` auf dem Übungsserver verbrauchen.

Aufgabe 7.2. *Bubblesort* ist ein Sortier-Algorithmus: Man vergleicht aufsteigend jedes Element eines Arrays x_j mit seinem Nachfolger x_{j+1} und – falls notwendig – vertauscht die beiden. Nach dem ersten Durchlauf muss zumindest das letzte Element bereits am richtigen Platz sein. Der nächste Durchlauf muss also nur noch bis zur vorletzten Stelle gehen, usw. Wie viele geschachtelte Schleifen braucht dieses Vorgehen? Schreiben Sie eine Funktion `bubblesort`, die ein gegebenes Array $x \in \mathbb{R}^n$ mittels Bubble-Sort aufsteigend sortiert, d.h. $x_1 \leq x_2 \leq \cdots \leq x_n$. Schreiben Sie darüber hinaus ein Hauptprogramm, in dem Sie den Vektor x und die Länge n einlesen, `bubblesort` aufrufen und das Ergebnis ausgeben. Wie groß ist der Aufwand ihrer Funktion? Speichern Sie den Source-Code unter `bubblesort.c` in das Verzeichnis `serie07`.
Hinweis: Verwenden Sie Vektoren mit (pseudo-)zufälligen Koeffizienten um Ihre Implementierung empirisch auf Korrektheit zu testen. Zufallszahlen zwischen 0 und N (`int`) können Sie folgendermaßen erzeugen: Zunächst binden Sie die Headerdateien `stdlib.h` und `time.h` in Ihr Programm ein. Danach können Sie in einer beliebigen Funktion mit

```
srand( (unsigned) time(NULL) );
zufallszahl = rand() % (N+1);
```

eine Zufallszahl zwischen 0 und N generieren. Die Variable `zufallszahl` ist dabei vom Typ `int`.

Aufgabe 7.3. Schreiben Sie eine Funktion `double* merge(double* a, int m, double* b, int n)`, die zwei aufsteigend sortierte Vektoren $a \in \mathbb{R}^m$ und $b \in \mathbb{R}^n$ so vereinigt, dass der resultierende Vektor $c \in \mathbb{R}^{m+n}$ 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 Vektoren a und b bereits sortiert sind. 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. Testen Sie ihr Programm mit entsprechenden Beispielen! Speichern Sie den Source-Code unter `merge.c` in das Verzeichnis `serie07`.

Aufgabe 7.4. Schreiben Sie eine rekursive Funktion `void mergesort(double* x, int n)`, die einen Vektor $x \in \mathbb{R}^n$ mittels des *Mergesort*-Algorithmus aufsteigend sortiert. Gehen Sie dabei nach folgender Strategie vor:

- Ist die Länge $n \leq 2$, so wird der Vektor $x \in \mathbb{R}^n$ explizit sortiert.
- Ist die Länge $n > 2$, halbiert man x in zwei Teilvektoren y und z. Man ruft rekursiv `mergesort` für y und z auf und vereinigt die beiden sortierten Teilvektoren wieder zu einem sortierten Vektor. Dabei soll explizit ausgenutzt werden, dass die Teilvektoren sortiert sind.

Schreiben Sie darüber hinaus ein Hauptprogramm, in dem Sie den Vektor x und die Länge n einlesen, `mergesort` aufrufen und das Ergebnis ausgeben. Wie haben Sie Ihren Code auf Korrektheit getestet? Speichern Sie den Source-Code unter `mergesort.c` in das Verzeichnis `serie07`.
Bonus: Wie groß ist der Aufwand ihrer Funktion?

Aufgabe 7.5. Man realisiere Mergesort aus Aufgabe 7.4, ohne im Rekursionsschritt Vektoren der halben Länge anzulegen, sondern verwende Pointer-Arithmetik: Ist `x` der Basepointer auf das Array x (d.h. der Pointer auf x_0), so ist `x+k` der Basepointer auf x_k. Man muss im Rekursionsschritt also lediglich den Basepointer auf x_0, den Startindex k und den Endindex ℓ eines Teilfeldes von x übergeben. Für das sortierte Ergebnisfeld ist *einmalig* am Anfang ein dynamisches Array anzulegen. Weiterer Hilfsspeicher ist *nicht* nötig. Wie haben Sie Ihren Code auf Korrektheit getestet? Speichern Sie den Source-Code unter `mergesort2.c` in das Verzeichnis `serie07`.

Aufgabe 7.6. Schreiben Sie eine rekursive Funktion `void quicksort(double* x, int n)`, die einen Vektor $x \in \mathbb{R}^n$ mittels des *Quicksort*-Algorithmus aufsteigend sortiert. Man wählt willkürlich ein Pivotelement aus der zu sortierenden Liste x, z.B. x_1. Dann zerlegt man die Liste in zwei Teillisten $x^{(<)}$ und $x^{(\geq)}$ und das Pivotelement x_1: $x^{(<)}$ enthält dabei alle Elemente $< x_1$, $x^{(\geq)}$ enthält nur Elemente $\geq x_1$. $x^{(<)}$ und $x^{(\geq)}$ werden rekursiv sortiert. Anschließend wird das Ergebnis zusammengesetzt. Eine direkte Implementierung dieses Algorithmus hat allerdings den Nachteil, dass zusätzlicher Speicher benötigt wird. Um dies zu vermeiden, gehen Sie nun folgendermaßen vor: Beginnend mit $j = 2$ sucht man ein Element $x_j \geq x_1$, d.h. x_j gehört zu $x^{(\geq)}$. Ferner sucht man beginnend bei $k = n$ ein Element $x_k < x_1$, d.h. x_k gehört zu $x^{(<)}$. In diesem Fall vertauscht man x_j und x_k. Wenn sich die Zähler j und k treffen, liegt die Liste x in der Form $(x_1, x^{(<)}, x^{(\geq)})$ vor. Mit einer weiteren Vertauschung erreicht man sofort die Gestalt $(x^{(<)}, x_1, x^{(\geq)})$. Es müssen nur noch $x^{(<)}$ und $x^{(\geq)}$ rekursiv sortiert werden. Schreiben Sie darüberhinaus ein Hauptprogramm, in dem Sie den Vektor x und und die Länge n einlesen und `quicksort` aufrufen. Testen Sie Ihren Code entsprechend! Speichern Sie den Source-Code unter `quicksort.c` in das Verzeichnis `serie07`.
Bonus: Wie groß ist der Aufwand ihrer Funktion?

Aufgabe 7.7. Schreiben Sie einen Strukturdatentyp `Date` zur Speicherung aller Daten ab 01.01.1900. Die Struktur besteht aus drei `int`-Members (Tag, Monat und Jahr). Schreiben Sie die Funktionen

- `Date* newDate(int d, int m, int y)`,
- `Date* delDate(Date* date)`,

sowie die sechs Zugriffsfunktionen

- `void setDateDay(Date* date, int d)`,
- `void setDateMonth(Date* date, int m)`,
- `void setDateYear(Date* date, int y)`,
- `int getDateDay(Date* date)`,
- `int getDateMonth(Date* date)`,
- `int getDateYear(Date* date)`.

Schreiben Sie die Funktion `int isMeaningful(Date* date)`, die die Zulässigkeit eines Datums überprüft (Rückgabewert `1` bei Zulässigkeit, sonst `0`). Das Datum 31.02.2013 z.B. ist nicht zulässig (Schaltjahre nicht vergessen!). Schreiben Sie ferner ein aufrufendes Hauptprogramm, um Ihre Implementierung entsprechend zu testen. Speichern Sie den Source-Code, aufgeteilt in Header-Datei `datum.h` und `datum.c`, in das Verzeichnis `serie07`.

Aufgabe 7.8. Schreiben Sie einen Strukturdatentyp `Person` zur Speicherung von personenbezogenen Daten. Die Struktur besteht aus vier Members: `firstname` (`char*`), `surname` (`char*`), `address` (`Address*`) und `birthday` (`Date*`). Schreiben Sie außerdem alle nötigen Funktionen um mit dieser Struktur arbeiten zu können. Verwenden Sie die Struktur `Date` aus Aufgabe 7.7 sowie die Struktur `Address` aus der Vorlesung (Folie 186). Schreiben Sie die Funktion `Person* whoIsOlder(Person* a, Person* b)`, welche von zwei gegebenen Personen die ältere Person zurückgibt. Testen Sie Ihre Implementierung entsprechend! Speichern Sie den Source-Code, aufgeteilt in Header-Datei `person.h` und `person.c`, in das Verzeichnis `serie07`.