Dirk Praetorius,
Gregor Gantner

Sommersemester 2018
08.05.2018

Übungen zur Vorlesung
Einführung in das Programmieren für TM

Serie 7

Aufgabe 7.1. As for the contents of variables of elementary type (`double,int,...`), you can print out the content of a pointer with help of `printf`. The place-holder `%p` is used for addresses (which are the contents of pointers!). The output is system-dependent, but mostly in hexadecimal numbers. Write a function `void charPointerDist(char* startaddress, char* endaddress)` that prints out the following three values tabularly:

- Starting address
- End address
- Distance (difference) between both addresses (take care of the place-holder in `printf`!)

Since arrays are stored connectedly, the distance between two successive elements corresponds to the memory used for the specific datatype. Test your function with a `char`-array `c[2]` and the following calls:

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

Then, write a function `void doublePointerDist(double* startaddress, double* endaddress)` and test it with a `double`-array. Compare the differences between the results of the two functions. Find out how much memory is used for the types `short`, `int`, and `long` on the `lva.student` server.

Aufgabe 7.2. The *bubblesort* algorithm is a sorting algorithm which works as follows: Run through the entries of a given vector $x \in \mathbb{R}^n$ several times. For every run, each entry x_j of x is compared to its successor x_{j+1}. If $x_j > x_{j+1}$, then the two entries x_j and x_{j+1} are swapped. After the first complete run through the vector, one knows that (at least) the last element is sorted correctly, i.e., the last element x_n is the maximum of the vector. Thus, in the next run, one only has to go up to the last-but-one entry of the vector (and so on). How many loops do you need for this algorithm? Write a function `bubblesort` which sorts a given vector $x \in \mathbb{R}^n$ in ascending order using this algorithm, i.e., $x_1 \leq x_2 \leq \cdots \leq x_n$. Additionally, write a main program, which reads the vector x and its length n from the keyboard, sorts it with `bubblesort` and prints to the screen the sorted vector. What is the computational cost of your function?
Hint: Use vectors with (pseudo-)random coefficients to empirically verify the correctness of your implementation. Random numbers between 0 and N (`int`) can be created as follows: First include the header files `stdlib.h` and `time.h` into your program. The following code lines

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

generate a random number between 0 and N. The variable `randnumber` has the type `int`. Save your source code as `bubblesort.c` into the directory `serie07`.

Aufgabe 7.3. Write a function `double* merge(double* a, int m, double* b, int n)` which merges two vectors $a \in \mathbb{R}^m$ and $b \in \mathbb{R}^n$, which are sorted in ascending order, into the vector $c \in \mathbb{R}^{m+n}$ in such a way that c is also sorted in ascending order, e.g., the choices $a = (1, 3, 3, 4, 7)$ and $b = (1, 2, 3, 8)$ lead to $c = (1, 1, 2, 3, 3, 3, 4, 7, 8)$. The algorithm should exploit the fact that the input vectors a and b are sorted. Write a main program that reads $m, n \in \mathbb{N}$ as well as $a \in \mathbb{R}^m$ and $b \in \mathbb{R}^n$ from the keyboard and prints to the screen the resulting vector $c \in \mathbb{R}^{m+n}$. Test your implementation in an appropriate way! Save your source code as `merge.c` into the directory `serie07`.

Aufgabe 7.4. Write a recursive function `void mergesort(double* x, int n)` which sorts a vector $x \in \mathbb{R}^n$ in ascending order using the *mergesort* algorithm. Use the following strategy:

- If $n \leq 2$, then the vector $x \in \mathbb{R}^n$ is explicitly sorted.
- If $n > 2$, then the vector x is split into two subvectors y and z of half length. Then the function `mergesort` is recursively called for y and z. Finally, y and z are merged into a sorted vector. Use explicitly the fact, that y and z are already sorted at that moment.

Write a main program, which reads the vector x and its length n from the keyboard, sorts it with `mergesort` and prints to the screen the sorted vector. Test your program accurately! Save your source code as `mergesort.c` into the directory `serie07`.
Bonus: What is the computational cost of your function?

Aufgabe 7.5. Implement the mergesort algorithm from Exercise 7.4 without allocating additional vectors in the recursion step. Instead, use pointer arithmetic: If `x` is the base-pointer of the array x (i.e. the pointer to x_0), then `x+k` is the base-pointer of x_k. Hence for the recursion step, it is sufficient, to simply have the base-pointer to x_0, the starting index k and the ending index ℓ of a part of x as input parameters. For the sorted final array you can *uniquely* allocate dynamic memory at the beginning. No additional memory is needed. How did you test the correctness of your code? Save your source code as `mergesort2.c` into the directory `serie07`.

Aufgabe 7.6. Write a function `void quicksort(double* x, int n)`, which sorts a vector $x \in \mathbb{R}^n$ in ascending order using the *quicksort* algorithm. Choose an arbitrary pivot element from x, e.g., x_1. Then, x is split into two parts, $x^{(<)}$ and $x^{(\geq)}$, and the pivot element x_1: $x^{(<)}$ contains all the elements $< x_1$, while $x^{(\geq)}$ contains all the elements $\geq x_1$. Finally, $x^{(<)}$ and $x^{(\geq)}$ are recursively sorted and the resulting sorted vector are merged. The direct implementation of this algorithm, however, requires additional storage. To circumvent this, proceed as follows: Starting with $j = 2$, search for an element $x_j \geq x_1$, i.e., x_j belongs to $x^{(\geq)}$. Then, starting with $k = n$, search for an element $x_k < x_1$, i.e., x_k belongs to $x^{(<)}$. In that case, swap x_j and x_k. If j and k coincide, then x has already the form $(x_1, x^{(<)}, x^{(\geq)})$. With one additional swap, the form $(x^{(<)}, x_1, x^{(\geq)})$ is obtained immediately. It remains to sort $x^{(<)}$ and $x^{(\geq)}$ recursively. Write a main program, that reads the vector x and the length n from the keyboard and calls the function. Test your implementation appropriately. Save your source code as `quicksort.c` into the directory `serie07`.
Bonus: What is the computational cost of your function?

Aufgabe 7.7. Write a structure `Date` for the storage of all dates from 01.01.1900 (January 1, 1900). The structure consists of three `int`-members (day, month, and year). Write the functions

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

as well as the mutator functions

- `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)`.

Moreover, implement the function `int isMeaningful(Date* date)`, which determines whether a given date is admissible (the function returns the value `1` if the date is admissible, the value `0` otherwise). For instance, the date 31.02.2013 is not admissible (don't forget to consider leap years!). Finally, write a main program to test your implementation in an appropriate way. Save the source code, split into a header file `datum.h` and `datum.c`, into the directory `serie07`.

Aufgabe 7.8. Write a structure `Person` for the storage of sensitive personal information. The structure consists of four members: `firstname` (`char*`), `surname` (`char*`), `address` (`Address*`), and `birthday` (`Date*`). Implement also all necessary functions to work with the structure. Use the structure `Date` from Exercise 7.7 as well as the structure `Address` from the lecture (Slide 183). Write the function `Person* whoIsOlder(Person* a, Person* b)`, which compares the age of two persons and returns the younger. Test your implementation accurately! Save the source code, split into a header file `person.h` and `person.c`, into the directory `serie07`.