CS244, STELLENBOSCH UNIVERSITY

TUT 2: POINTERS IN C

29 JULY 2022

> *. . . methods are more important than facts.* The educational value of a problem given to a student depends mostly on how often the thought processes that are invoked to solve it will be helpful in later situations. It has little to do with how useful the answer to the problem may be. On the other hand, a good problem must also motivate the students; they should be interested in seeing the answer. Since students differ so greatly, I cannot expect everyone to like the problems that please me.
>
> —Donald E. Knuth

INSTRUCTIONS

- Compile your code with the `-Wall`, `-ansi`, and `-pedantic` switches.
- You should adhere to the programming practice spelled out in the tutorial (and solutions) of last week. In particular, there must not be any warnings, you must split your solution into several files where necessary, and you should test your programs.

EXERCISES

1. Write your own version of the standard library function `strlen` that returns the length of the string specified as parameter. As a matter of fact, write two:
   (a) The first must treat the string as a character array.
   (b) The second must compute the string length using pointer arithmetic.
2. Write your own version of the standard library function `strcmp(s, t)`, which compares the character strings `s` and `t`, and returns negative, zero, or positive if `s` is lexicographically less than, equal to, or greater than `t`. Provide two versions:
   (a) One that uses array indices, and
   (b) another that uses pointer arithmetic.
3. Write your own version of the standard library function `strcpy` that copies the contents of the second string to the first string. You may assume that the target array has enough space. Once again, write two versions:
   (a) One that uses array indices, and
   (b) another that uses pointer arithmetic.
4. Write the function `strend(s, t)`, which returns 1 if the string `t` occurs at the end of the string `s`, and 0 otherwise.
5. Write the function `strdup(s)` which returns a copy of the string `s`. Your function must reserve space for the copy with a call to the library function `malloc`. Your test program must use the library function `free` to deallocate space reserved with `malloc`. Use the command-line utility `valgrind` to test for memory leaks; the flag `-help` lists all the available flags, and `-leak-check=full` results in a detailed report of any leaks found.

6. Write a quicksort implementation `qsort(int a[], int i, int j)` that (recursively) quicksorts (into increasing order) the `int` array `a` between the specified indices. Use your own `swap` function to swap the values at two indices. Write a test program which reads the integers to be sorted from a file, the name of which is provided as command line argument.

7. Write your own (slightly different) version of the library function `qsort(void *a[], int left, int right, int (*comp)(void *, void *))` that (recursively) quicksorts `a` between the indices `left` and `right`, using the specified function pointer as comparison function. Test your implementation with your own `strcmp` function, and write another comparison that, for example, is able to compare two numbers.

8. Once an array is sorted, it is trivial to find a search key using a binary search. Implement binary search as a function. Try and use function pointers to make the function as general as possible. Try and write a test program which reads the integers to be sorted from a file, the name of which is provided as command line argument.

9. Compare the approach to **polymorphism**[1] in Exercise 7 with that of the solutions to Tutorial 1. In your consideration, pay particular attention to whether you consider a particular approach to be **static** (resolved at compile time) or **dynamic** (resolved at runtime). How does this compare to the method overloading, generic types, and subtyping found in Java?

10. It is also possible to sort data with a binary tree. Write a program that sorts data by first inserting each datum into an ordered binary tree. Use a C `struct` to create the structure for a tree node, which should contain pointers to the value stored at the node as well as the left and right subtrees. It might be useful to create a function `tnalloc` that wraps a call to `malloc` for allocation. Note that this method is not in-place, and depending on your implementation, you will either have to overwrite the original array of values, or create a sorted copy. Include a function in your implementation that destroys your binary tree after the sort has completed. This means, all space taken up by the tree must be `freed`.

[1] https://en.wikipedia.org/wiki/Polymorphism_(computer_science)