HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2018
LAB 3: MORE PROGRAMMING IN C AND COMPUTATIONAL COMPLEXITY

The aim of this lab is to continue practicing the fundamental concepts of the programming language C. In this lab you will write your own code or complete provided code. This lab also includes a part about computational complexity.

Note. You are not required to write makefiles in order to compile your code.

Log into a system and download the lab tar-ball `Lab03_More_C.tar.gz` from the Student Portal. Save it and unpack it as you learned in Lab 1.

1. Multidimensional arrays

Task 1:

Write a C program to display the matrix below:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ -1 & 0 & 1 & 1 & 1 \\ -1 & -1 & 0 & 1 & 1 \\ -1 & -1 & -1 & 0 & 1 \\ -1 & -1 & -1 & -1 & 0 \end{bmatrix}$$

Note. Use statically allocated two-dimensional arrays. First store the matrix as a two-dimensional array in memory, and then use the values from the array to print the matrix.

Task 2:

In this task you should complete a code for a simple minesweeper game. Open the source file `minesweeper.c` located in the directory `Task-2`. Run the program without parameters to read game rules.

The game field is represented by a two-dimensional array of size $n \times m$, where n and m are input parameters. The number of bombs k is also an input parameter.

Your task is to allocate dynamically memory for the field before the game starts and deallocate memory after the end of the game, see comments in the code.

Date: January 22, 2018.

2. Pointers to functions

Short summary of how function pointers work

Example: we have a function `int myfunc(int* x, double y)`.

- Declaration of a function pointer `foo` to a function with two input parameters of types `int*` and `double` respectively, and return value of type `int`:

  `int (*foo)(int*, double);`

- Assigning a value to the declared pointer:

  `foo = &myfunc;` or `foo = myfunc;` (both ways are allowed)

- Invoking the function pointed to by the pointer `foo`:

  `foo(x, y);` or `(*foo)(x, y);` (both ways are allowed)

Task 3:

You are given the following two functions:

```
void print_int_1(int x) {
  printf("Here is the number: %d\n", x);
}
void print_int_2(int x) {
  printf("Wow, %d is really an impressive number!\n", x);
}
```

Write a C program which creates a function pointer to that kind of functions and calls `print_int_1` using this function pointer, then set the pointer to point to `print_int_2` instead, and use the function pointer to call that function. Does it work? Note that the same function pointer can be used to refer to different functions, as long as the return type and argument types are the same.

Task 4:

Study the `qsort` function for sorting arrays, for example here:
http://www.cplusplus.com/reference/cstdlib/qsort/

It has the declaration

```
void qsort (void* base, size_t num, size_t size, int (*compar)(const void*,const void*));
```

with the following arguments:

- `base`
  Pointer to the first object of the array to be sorted, converted to a void*.

- `num`
  Number of elements in the array pointed to by base.

- `size`
  Size in bytes of each element in the array.

- `compar`
  Pointer to a function that compares two elements. This function is called

repeatedly by `qsort` to compare two elements. It shall follow the following prototype: `int compar (const void* p1, const void* p2);`

Note that `compar` returns an int. The function defines the order of the elements by returning a value

< 0 The element pointed to by p1 goes before the element pointed to by p2
0 The element pointed to by p1 is equivalent to the element pointed to by p2
> 0 The element pointed to by p1 goes after the element pointed to by p2

Write a C program which sorts an array of real numbers in descending order using the `qsort` function. You should write a function `CmpDouble` which follows the prototype of `compar` and compares two elements of the array, such that the following code sorts elements of the array `arrDouble` in descending order

```
double arrDouble[] = {9.3, -2.3, 1.2, -0.4, 2, 9.2, 1, 2, 0};
int arrDoubleLen = sizeof(arrDouble) / sizeof(double);
qsort (arrDouble, arrDoubleLen, sizeof(double), CmpDouble);
```

Note: If you like, you can add some `printf` statements inside your `CmpDouble` function to see explicitly what is happening each time it is called, printing e.g. "Now the following two values are compared...".

Task 5:

Write a C program which sorts an array of strings in alphabetic order using the `qsort` function. You should write a function `CmpString` which follows the prototype of `compar` and compare two elements of the array, such that the following code sorts elements of the array `arrStr` in alphabetic order:

```
char *arrStr[] = {"daa", "cbab", "bbbb", "bababa", "ccccc", "aaaa"};
int arrStrLen = sizeof(arrStr) / sizeof(char *);
qsort(arrStr, arrStrLen, sizeof(char *), CmpString);
```

Hint. Use the `strcmp` function for comparison of strings.

Hint: Note that the `CmpString` function should get pointers to the elements in the array which should be compared. In our case we get pointers to strings (char**). To help understanding what is happening, it may again be helpful to add some `printf` statements inside your `CmpString` function, showing which strings are being compared.

3. Binary search tree

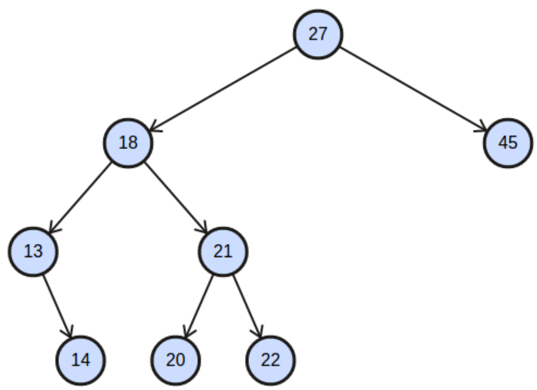
A binary tree is a tree in which each node is allowed to have no more than two children, usually called left and right. A binary search tree (BST) is a special kind of binary tree where the data is structured in some way using additional requirements:

- there must be no nodes with the same value

- for every node its right subtree (the tree that has its right child as root) should consist only of nodes with values greater than its value

Use this BST Visualizer for creating examples and better understanding:
http://btv.melezinek.cz/binary-search-tree.html

One example of a BST:



The code required for this section is located in the directory `Task-6`. This code will be used in tasks 6, 7 and 8.

In this section we will implement a simple plant database for a botanical garden. We will use a binary search tree where each node represents a plant. Each plant has a unique ID represented by an integer number and a name represented by a string. Nodes in the binary search tree are sorted by the plant ID.

Each node is given by a structure

```
typedef struct tree_node
{
int ID;
char * name;
struct tree_node *left;
struct tree_node *right;
} node_t;
```

The function `print_bst` located in the file `bst.c` for each node prints ID, name and IDs of the left and right child nodes.

At the start the tree is empty:

```
node_t *root = NULL;
```

Task 6:

Implement a function `insert` which inserts the information about the plant into the tree such that the properties of the binary search tree are preserved.

For example insert nodes (ID, name):

- 445, sequoia
- 162, fir
- 612, baobab
- 845, spruce
- 862, rose
- 168, banana
- 225, orchid
- 582, chamomile

then the `print_bst` function should give:

```
445 sequoia:    L162,R612
162 fir:        R168
168 banana:     R225
225 orchid:
612 baobab:     L582,R845
582 chamomile:
845 spruce:     R862
862 rose:
```

where for example line `445 sequoia:  L162,R612` means that the node with ID 445 and plant name "sequoia" has two child nodes: the left child with ID 162 and right child with ID 612.

Create this tree using the BST Visualizer.

Note. In each node memory for a structure member `name` should be allocated dynamically when the node is created. If you want, you can use the function `strdup(const char *s)` from the header file `string.h`. The function allocates sufficient memory for a copy of the string s, copies s, and returns a pointer to the copy. Alternatively, you can allocate memory yourself by calling `malloc` and then copy the data of the string yourself.

Note: Note that the `insert` function should allocate memory dynamically for the `name` string; it should not just copy the input pointer. If you only set `name` to point to the input string, you are pointing to data that may change and/or no longer exist later. You need to use dynamic memory allocation for `name` to make your BST data structure independent. The simple usage in the `bst.c` program can work anyway since all the strings still exist in that case. For another usage case where it is critical that the strings are copied properly, see the `bst_test.c` program. There, strings are given by input from the user and the previous strings no longer exist in memory. In that case, what would happen if the `insert` function just copied the input pointer?

Task 7:

Implement a function `delete_tree` which recursively deletes all the nodes in the tree.

Hint. The empty tree is represented by a NULL pointer. After deleting the tree, we should set the `root` pointer to NULL inside the `delete_tree` function. In general, it is a good practice to set a pointer to NULL after calling function `free`. Since we want the `delete_tree` function to be able to modify the pointer, the function has as an input parameter of type `node_t**` (a pointer to a pointer to a `node_t`).

Hint. Do not forget to free memory allocated for names.

Task 8:

Implement a function `search` which searches the tree to find the name of a plant with a given ID.

Example:

```
search(root, 168);
search(root, 467);
```

Output:

```
Plant with ID 168 has name banana
Plant with ID 467 does not exist!
```

4. Time measuring

Task 9:

The code for this task is located in the `Task-9` directory.

The program finds the minimum and maximum values in a square matrix. The code consists of the following functions:

- allocate_matrix (Allocate memory for a matrix)
- deallocate_matrix (Free the allocated memory of a matrix)
- fill_matrix (Fill the matrix with random floating-point numbers in the range 0 to 10)
- print_matrix (Display the matrix)
- get_min_value (Return the minimum element value of the matrix)
- get_max_value (Return the maximum element value of the matrix)

1. Run the code for different matrix sizes, e.g. 100, 500, 1000, 5000, and 10000. Use the `time` command to measure the time required to run the entire program:

```
time ./matrix
```

where `matrix` is the name of the executable file. Note the user, system, and wall times.

2. Add another matrix allocation and deallocation by adding the following two lines in the code:

```
double** matrixB = allocate_matrix(n);
deallocate_matrix(matrixB,n);
```

How does this affect the system time?

3. Use the `gettimeofday()` function from `time.h` to measure the time needed to execute each part of the program, and use `printf` statements to output the timings. *Hint:* it can be convenient to write a small help function like this:

```
double get_wall_seconds(){
struct timeval tv;
gettimeofday(&tv, NULL);
double seconds = tv.tv_sec + (double)tv.tv_usec/1000000;
return seconds;
}
```

4. Repeat step 3, but now use `clock_gettime()` (remember to link with `-lrt`) if you're on Solaris or Linux, or `clock_get_time()` if you have a Mac. The following url contains a useful code snippet: https://gist.github.com/jbenet/1087739

5. Complexity

Task 10:

A C program `permutations.c` in the directory `Task-10` is counting all permutations of a given string. You can run the program on a string "absndha" using command:

```
./a.out absndha
```

where `a.out` is the executable file.

Which *space* complexity has this algorithm for searching all permutations of a string? Which *time* complexity has this algorithm for searching all permutations of a string? Run the program for strings of various lengths and measure running times.

Task 11:

The bubble sort algorithm for sorting an array is implemented in the source file `bubble_sort.c` in the directory `Task-11`. The program allocates memory for an array of length n and fills it with random numbers. Then in the function `bubble_sort()` the array is sorted. You can run the program using the command:

```
./bubble n
```

where `bubble` is the executable file.

Take a look in the function `bubble_sort()`. What is the time complexity of the algorithm? Run the program for various array lengths and measure execution time.

Task 12:

In this task you will compare the "bubble sort" and "merge sort" algorithms. Merge sort is implemented in the source file `merge_sort.c` in the directory `Task-12`. You can run the program using the command:

```
./merge n
```

where `merge` is the executable file. The time complexity of merge sort is $O(n \log n)$, where n is the length of the array. Compare the execution time of the bubble and merge sort algorithms. Which algorithm would you choose?

Task 13:

Write a code doing matrix-matrix multiplication. What is the space complexity? What is the time complexity? Compare theoretical expectations to your observed timings. Do the timings behave as you expected?

6. Extra part

In case you did not yet solve the extra part in the end of Lab 2, go back and look at that again. If you are stuck, ask your teacher for a hint!