OBJECT ORIENTED PROGRAMMING

SEMINAR 1

OBJECTIVES

The main purpose of this seminar is to acquaint students with basic C programming concepts. We'll learn how to approach, solve and implement (**in C, not C++!**) simple programming problems. You'll also learn about primitive data types, statements, functions and two ways of passing parameters to a function: pass by value (the default) and pass by pointer.

PROPOSED PROBLEMS

1. Write a C program to check if a number is prime. We'll use *test driven development* (TDD) when solving the problem. TDD might seem counterintuitive as it reverses the canonical process of software development, where you first write the code and then create the tests for the code. In test driven development you start by writing the tests and then incrementally modify the code such that all the tests pass. To sum up, in TDD:
    a. you start by reading, **understanding**, and processing the problem.
    b. you write a new test (*assert* might prove useful, use #include <assert.h>);
    c. run all your tests (all the tests except the newly added one should pass);
    d. write the code such that your new test passes;
    e. all tests should now pass;
    f. if needed, do some refactoring;
    g. go back to point b (repeat the process).

    So first we need to understand the definition of a prime number:

    A natural number is prime if it has only two factors 1 and itself.

    We need to write a function that checks if a number is prime or not. What is the input and the output of this function? Think about the primitive data types that you should use (let's say bye bye to those programs where we use *int*s for everything!).
    What are the preconditions, postconditions and invariants of this function?

    Now think about the test cases and iteratively add the test cases and implement the code until you solve the problem.

2. Write a function that finds and returns the first *n* prime numbers. It turns out that there is a well known ancient algorithm for this: the *Sieve of Eratosthenes*

The **output** of this function should be **an array** containing the prime numbers smaller than or equal to *n,* as well as the number of elements that fulfill this property. **(Don't just print the prime numbers in the function!**).

The steps of the algorithm are the following:

Sieve of Eratosthenes Find all the prime numbers smaller or equal to a number n
1. Generate a sequence with all the prime numbers from 2 to n 2. Start with $p = 2$, the smallest prime number 3. Mark all the multiples of p in this sequence ($2p$, $3p$, $4p$ …) 4. Move to the smallest number greater than p and repeat starting from point 3. 5. All the elements left unmarked in the sequence will be the prime numbers smaller or equal to *n*.

For example, if we want to generate all the prime numbers up to 12:
We first generate the numbers from 2 to 12:

2	3	4	5	6	7	8	9	10	11	12

Start from $p = 2$. Mark all the multiples of 2 (except 2).

2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~	11	~~12~~

Move to the next $p = 3$ (a prime number), and mark all its multiples:

2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~	11	~~12~~

Move to the next $p = 5$ (a prime number), and mark all its multiples. None marked.
Move to the next $p = 7$ (a prime number), and mark all its multiples. None marked.
Move to the next $p = 11$ (a prime number), and mark all its multiples. None marked.
All the number that are not marked are prime numbers:

{2, 3, 5, 7, 11}, there are 5 prime numbers smaller or equal to 12.

Your function should pass all the tests provided in the starter code!

3. Pretty-print in a tabular form all the prime numbers number up to 1000. The numbers should be properly aligned in a "square matrix".
   https://www.cplusplus.com/reference/cstdio/printf/

4. The Goldbach conjecture states that any ***even whole number that is greater than 2 can be written as the sum of two prime numbers***. Write a function that takes as parameter a number *n* (less than or equal to 100), computes and returns the two prime numbers that, if added, give *n*. If the number passed as parameter is not in the expected range [3, 100] or it is not even, the function will return *false* (*#include<stdbool.h>*); in this case the prime numbers will be set to 0. The prime numbers that you return from the function should be ordered (the first prime number is always less than or equal to the second prime number).
   You can use a lookup table to store the prime number up until 100.

5. Write a function that approximates the square root of a real positive number *n* with precision *eps*. You should use a *divide et impera* approach for this. *Divide et impera* is a programming technique in which the base problem is broken down into smaller problems of the same type, until we reach a problem that is so simple that it can be solved directly. Then, the solutions to all these simpler subproblems are combined to get the solution of the original problem.
   You can treat this problem as a searching problem. We know that the square root of a real positive number is in the interval [0, n]. So, similar to binary search, we can successively half and search in this interval the number *s* which if squared it would give a number approximately equal to *n:* $|s * s - eps| \leq n$.
   Now modify the function such that it computes the n-th root of a number.

Use test driven development when solving this problem.

Recommended video:
https://www.youtube.com/watch?v=p8u_k2LIZyo