Exercise Sheet #4

Deadline: 13.05.2024, 12:00h

Can't get enough of `C++`? This series on YouTube is recommended to strengthen your understanding of the concepts, for lecture revision and to learn even more about `C++`.

Problem 1 (*Templates and Function Pointers*) (10 points)

Templates in `C++` are a powerful feature that allows you to write generic code that can work with different data types. We therefore do not have to rewrite a function if we want to pass different types to it. For the following tasks, implement the requested functions and show its functionality by providing at least one suitable example.

(a) Implement two functions `add` and `multiply` with a template type `T` for the arguments and the return value, which takes two numbers of the same template type (e.g. `int`, `double`) and return the sum and product of those numbers, respectively.

(② points)

On exercise sheet #3 we worked with pointers to variables. Just like variables, function are stored somewhere in memory – we can therefore also have function pointers. To retrieve the address of a function, we use the syntax `&function_name`. We can declare function pointers with the syntax `return_type (*funcPtr)(parameter_type, ...);`. In order to call the function to which the function pointer points to, we can use the syntax `(*funcPtr)(parameters);`.[1]

(b) We now want to write a function that prints the result of a mathematical operation of two numbers. Write a function `printOperationResult` with three parameters and return type `void`. Two parameters are numbers of template type `T` the third is a function pointer to a function taking itself two numbers of type `T` as input and returning a number of type `T`. The function `printOperationResult` should call the function associated with the third parameter (function pointer) with the first to parameters and print the result of this function call to the console. Test your function with the two functions from part (a) as input.

(④ points)

[1]It also works to just write `funcPtr(parameters);`

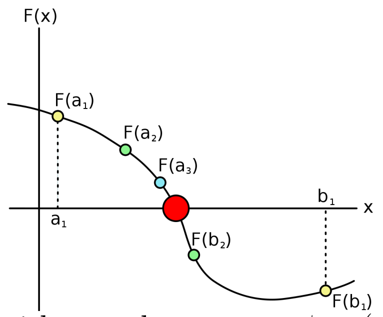
(c) Write a function that has a templated integer parameter and takes as an argument an array of template type `T` and length `N`. The function shall print all entries of the array and the size of the array.
Hint: The function declaration then looks like this:
```
template<typename T, int N> func_name (T (&array)[N]).
```
(① point)

(d) What happens if the function from part (b) is called with a dynamically allocated array, i.e. `int * a = new int[N]`. Why does this happen? Give your solution to the question in a comment of your code.
Hint: The functions with the appropriate data types are generated from the templates at compile time, what about dynamically allocated arrays?
(① point)

(e) Write a function similar to the one from part (c) that you can pass a two-dimensional $N \times M$ array and that prints the dimensions of the array and all entries to the console.
(② points)

Problem 2 (*Iterative Bisection Method*) (10 points)

The *Iterative Bisection Method* is a simple algorithm to find the roots of a function. Your goal is to implement the algorithm both iteratively and recursively, try out your implementation with a few examples and compare the two implementations. Here is how the algorithm works:



Let $F\colon [a,b] \to \mathbb{R}$ be a continuous function with exactly one root $\xi \in (a,b)$ and $F(a) \cdot F(b) < 0$. Start by setting $a_1 := a$ and $b_1 := b$. For $n = 1, 2, \dots$, set

$$x := \frac{a_n + b_n}{2}.$$

If $F(a_n) \cdot F(x) \leq 0$, set $a_{n+1} := a_n$ and $b_{n+1} := x$, else set $a_{n+1} := x$ and $b_{n+1} := b_n$. Terminate, if $\|b_{n+1} - a_{n+1}\| < \epsilon$. For a visualization of the algorithm see the figure to the right.
Follow these steps which will guide you through the implementation:

(a) Define a function `iterative_bisection` that takes a function pointer to the function F under investigation, starting values a_1 and b_1, an absolute tolerance ϵ and a maximal number of iterations as parameters, and implement the iterative bisection method in the function body. Terminate the algorithm if either the desired accuracy or the maximal number of iterations is reached and output the found value for the root, the remaining error range and the number of iterations until convergence to the console. Make sure to test the given starting values for validity and display appropriate error messages if something goes wrong.

(⑤ points)

(b) Now implement a function `iterative_bisection_recursive` that runs the same algorithm, but does so using recursion.

(④ points)

(c) Test both of your implementation with at least two different functions F and make sure they yield the same results.

(① point)

Problem 3 (Advanced: *Bitwise Operations*) (10 points)

In `C++` you can manipulate data on the level of its binary representation. The six bitwise operations are:

Operator	Symbol	Form	Operation
left shift	`<<`	`x << y`	all bits in `x` shifted left `y` bits
right shift	`>>`	`x >> y`	all bits in `x` shifted right `y` bits
bitwise NOT	`~`	`~ x`	all bits in `x` flipped
bitwise AND	`&`	`x & y`	each bit in `x` AND each bit in `y`
bitwise OR	`|`	`x | y`	each bit in `x` OR each bit in `y`
bitwise XOR	`^`	`x ^ y`	each bit in `x` XOR each bit in `y`

The goal of this exercise is to do basic algebra on the bitwise level.

- Write a function that takes two positive integers and returns their sum, not using `+`, `-`, `*` or `/`. You should be able to achieve this by manipulating the numbers on the bitwise level. You are also allowed to use the conventional syntax to control the program flow, as well as comparators (e.g. `x > y`). For a start, read up on how addition of numbers is done in binary arithmetic e.g. at https://en.wikipedia.org/wiki/Binary_number#Addition.

  Hint: It might be necessary to perform the carrying, as described in the reference, multiple times. You need to add the carried bits to the result, so recursion could be useful.

- Write a function that implements the multiplication of two positive integers with the same restrictions to bitwise operations. Read up on how to multiply binary numbers e.g. at https://en.wikipedia.org/wiki/Binary_number#Multiplication.

  Hint: You should use the adding function from the first part of the exercise.