Exercise Sheet #5

Deadline: 20.05.2024, 12:00h

Problem 1 (*Lambda Expressions*) (10 points)

For all the following parts, provide the implementation and at least one meaningful example.

(a) Use a lambda expression that takes two `int`s and returns their sum to assign this returned value to a variable.

(① point)

(b) Use a lambda expression that captures an `int` from its enclosing function and takes an `int` parameter to directly assign the return value to a variable. Again, the lambda should return the sum of the captured variable and the parameter.

(① point)

(c) Instead of directly calling the lambda expressions, try to create function pointers from them. You should get an error message for the lambda expression that captures a variable but not for the one that only takes parameters. This is perfectly fine, since `C++` does not allow the conversion of a lambda expression that captures variables to a function pointer.

(① point)

(d) Using the variable capturing lambda expression, try to add a line that changes the value of the captured variable. You should get a compilation error that tells you that you are trying to assign a value to a read-only variable. Provide a simple fix for that.

(① point)

(e) An often encountered use case for lambda expressions is to pass them to functions expecting a small instruction how to operate. Write a function `print_vector` that takes an integer array and a lambda expression as input. The function should apply the lambda expression to each element of the array and print the result to the console.

Hint: To pass a lambda expression to a function you can use `std::function` (see here) or simply use function pointers, as we used previously on sheet #4.

(③ points)

(f) Lambda expressions can simplify your code and make it more readable. Write two versions of a function that takes an array of integers `int (&arr)[N]` and an `int &s` as input and adds all the elements of `arr` to `s`. The first version should work using a simple for-loop, while the second iterates over the vector using `std::for_each` (see here) and adds the elements of `arr` to `s` using a lambda expression.

(③ points)

Problem 2 (*Header Files and Namespaces*) (10 points)

For more complex projects and especially when developing code in a group, it can be useful to create a multi-file structure. Then there is exactly one executable file containing the `main` function and all functions, classes, structs etc. are loaded from external `C++` files and header files. E.g. a function is defined in a separate `C++` file (e.g. `test.cpp`) and only the function body, i.e. with name, argument types, return type, is defined in a header file of same name (`test.h`). The function in the external file can be loaded by including the corresponding header file (`#include "test.h"`). Notice that here quotation marks are used instead of angle brackets.

(a) Write a function that generates a random 4×4 array with integer entries in the range of $[0, 10]$, by receiving an existing empty array and changing its entries. You can use the `rand()` function to do that, but you will need to provide it with a seed, using `srand(time(0));`. Note that the seed should be initialized outside the function.

(③ points)

(b) Write a matrix multiplication function that takes as input two 4×4 arrays and prints the result of their multiplication.

(③ points)

(c) Write a `main` function in which you initialize two empty 4×4 arrays and call the random array generator on both. Print out both arrays and multiply them using the multiplication function.

(① point)

(d) In big projects it is helpful to separate your smaller functions (utility functions) to another file. Split your code into two parts: A `main.cpp` file containing only the `main` function, and a `utilities.cpp` file containing your other functions. Add a header file where you define a new namespace and declare the function bodies. Import the namespace (and your function in it) into the `main.cpp` code. When compiling with `g++`, make sure to use `g++ main.cpp utilities.cpp`. The process of combining different object files into a single executable is called *linking*.

(② points)

(e) Use doxygen to generate documentation for your project.

(① point)

Problem 3 (Advanced: *Gradient Descent Algorithm*) (10 points)

A frequently encountered task in machine learning is to find the local minima of a given $N \in \mathbb{N}$ dimensional function. In this exercise we want to implement the *Gradient Descent Algorithm*, a simple method to find local minima. Your goal is to implement the algorithm and try out your implementation with an example of your choice. The algorithm works as follows:
Let $U \subseteq \mathbb{R}^N$ and $F: U \to \mathbb{R}$ be a differentiable function. For every $x \in U$, the steepest descent of F is given by the gradient $\nabla F(x)$. Start by setting $x_1 \in U$. For $n = 1, 2, \ldots$, progress into the direction of the steepest descent, i.e. set

$$x_{n+1} = x_n - \alpha \nabla F(x_n),$$

where $\alpha > 0$ is a suitable step size. Terminate if $\|\alpha \nabla F(x_n)\| < \epsilon$, for an absolute tolerance $\epsilon > 0$.
Follow these steps which will guide you through the implementation:

- Define a function `steepest_descent` which takes the function under investigation, its respective gradient, a starting point x_1, a step size α and a tolerance ϵ as input and computes an approximation to a local minimum of the function using the gradient descent algorithm. Make sure that the function can handle arbitrary dimensionality N. Use a template to accept arbitrary vector space dimensionality. The function F and its gradient should be passed to the function as function pointers.

  Hint: To handle vectors either read up on `std:array` or use C-style arrays, pass them to the respective function by reference and modify them in-place.

  Hint: To make your code more readable, you can define types for the function pointers to F and $\mathrm{grad}\, F$. With that the full function declaration could look something like this:

```
template <size_t N>
using f_ptr = double (*)(double(&x)[N]);
template <size_t N>
using g_ptr = void (*)(double(&x)[N], double(&grad)[N]);

template <size_t N>
void steepest_descent(
    func_ptr<N> f, grad_ptr<N> g,
    double(&x)[N], double alpha, double tol)
{
    ...
}
```

- Test your implementation with an example for the function F of your choice and compare the result with your analytical prediction.