Advanced Programming Tutorial
Worksheet 3: Functions

Keywords: recursion, operator overloading, function templates

Prepare!

Before coming to the tutorial, and so that the time is enough to solve the exercise, revise the slides of the lecture unit "Functions" and answer the following questions:

- ☐ How do we define functions? What is the difference between arguments and return value?
- ☐ What happens when we call a function?
- ☐ What is the difference between pass-by-value and pass-by-reference?
- ☐ What do we need to be careful about when we return by reference?
- ☐ What is function overloading? How does it work?
- ☐ What are anonymous functions? How do we define one?
- ☐ What is a recursive function? Write down an example. When does it stop?

Exercise 1: Greatest Common Divisor

In this task we consider the greatest common divisor algorithm which is defined as follows:

$$gcd(a, 0) = a \tag{1}$$
$$gcd(a, b) = gcd(b, a \texttt{ mod } b) \tag{2}$$

Implement this algorithm in 2 (3) different variants:

1. Recursively. You are allowed to use temporary variables, but no static or global ones.
2. Iteratively. Same constraints as in the recursive approach.
3. **Idea:** Iteratively, but this time without temporary variables. Hint: You can swap two variables using `std::swap` or using simple arithmetic (or binary) operations.

The number and type of arguments should remain the same, the name can differ. The return value should be the greatest common divisor of the provided input arguments. Hint: `int gcd(int a, int b);`

Exercise 2: Overloading Operators

You are implementing a Linear Algebra library in C++. For this, you will build upon the previous worksheet where we used nested `for` loops to compute a matrix-vector product. You will be building upon the the solution of the previous worksheet, `matvec.cpp`.

1. You now want to print both `int` and `double` matrices. For this, overload the `print_vector` and `print_matrix` functions to support both `int` and `double` matrices.

2. Your friend Alice asks if you will allow printing matrices of type `float` as well. You decide that instead of overloading for just `int` and `double`, you will support printing vectors/matrices of any type `T` using function templates. Implement templated versions of `print_vector` and `print_matrix` to achieve this.

   Idea: Reuse `print_vector` inside `print_matrix`.

3. You already know that you can write `for` loops to compute matrix-vector products. However, for convenience, you want to write mathematical expressions for matrix-vector products instead. That is, you want to be able to write the following code:

```
std::vector<std::vector<double>> Matrix;
std::vector<double> vec;

// later

std::vector<double> mat_vec_product = Matrix * vec;
```

   In C++, you can do this by overloading `operator*`, i.e., the multiplication operator. This is just a function with the name `operator*` which defines what to do when you write expressions involving the multiplication symbol. There are similar functions for other arithmetic operators too. Here's an example of using `operator+` to define vector addition:

```
std::vector<double> operator+ (const std::vector<double>& vec_a,
                               const std::vector<double>& vec_b){
    assert( vec_a.size() == vec_b.size() ); // Complain if sizes are not
    auto N = vec_a.size();                  // compatible (debug only)

    std::vector<double> vec_sum(N, 0.0);
    for (auto i = 0; i < N; i++){
        vec_sum[i] = vec_a[i] + vec_b[i];
    }
    return vec_sum;
}

int main(){
    std::vector<double> vec_a = {1, 2, 3};
    std::vector<double> vec_b = {4, 5, 6};

    auto res = vec_a + vec_b;
}
```

   Use this example as a guide to overload `operator*` to allow writing code as in the example above.

   Idea: Allow the same for matrices and vectors of `int` type.

4. You now want to check your implementation of `operator*` from 3 for correctness. Implement a function with the signature `bool test_matrix_vector_product()` that tests your

implementation of `operator*` by comparing its return value with a reference solution for at least one pre-defined scenario of your choice. Inform the user whether the test succeeded or failed and in case of failure, print the reference and the result vectors.

5. **Idea:** Implement the same tests using doctest (or any other unit testing framework).

Advice / C++ Core Guidelines

The C++ Core Guidelines are a "collaborative effort led by Bjarne Stroustrup, much like the C++ language itself", which aims to "help people to use modern C++ effectively". You can always refer to them if you are confused about best practices. Take a look at the following, for example:

- F.3: Keep functions short and simple
- F.16: For "in" parameters, pass cheaply-copied types by value and others by ref. to const
- F.17: For "in-out" parameters, pass by reference to non-const
- F.51: Where there is a choice, prefer default arguments over overloading

Epilogue

Recursion can often lead to faster and more elegant implementations, but be aware of the memory consumption. The tail recursion optimization of C++ reduces the used memory. Regarding overloading, the important is that you understand that overloaded versions are functions with different names in the binary. Remember that we can also use templates (and template specialization) to automatically generate different versions. While overloading for basic types is not that interesting, it is very useful when dealing with more complex types.