Dr. Ole Klein, Stefan Meggendorfer
IWR, Heidelberg University
Submission date 14.12.2021

Exercise Sheet 7

Exercise 1: Template Metaprogramming: Number Sequences (10 points)

Template metaprogramming might be an unusual programming technique, but it is Turing complete: this can be shown by simply implementing a Turing machine in C++ templates (you can find some online if you are interested). This shows that anything you can write as a normal program could, at least in principle, be achieved using template metaprogramming (see Church-Turing conjecture[1]). The main obstacles are the finite amount of recursive template levels the compiler allows, the fact that dynamic interaction with the user is impossible, the fact that floating-point numbers can't be template parameters, and of course the slightly awkward way things have to be written. The only way to provide some form of user input is through preprocessor macros, since the usual command line arguments and queries using `std::cin` are not known at compile time.

Solve the following two tasks with template metaprogramming:

(a) **Fibonacci Numbers.** Write both a template metaprogram and a normal function that compute the k-th element of the Fibonacci sequence for given $k \in \mathbb{N}$:

$$a_k := \begin{cases} 1 & \text{for } k = 0 \\ 1 & \text{for } k = 1 \\ a_{k-2} + a_{k-1} & \text{else} \end{cases}$$

Base your two implementations on the mathematical definition above, i.e., directly implement the given recursion formula. Then write a main function that calls both versions for the argument "`INDEX`", and prints the results. Here, `INDEX` is a preprocessor macro, which we use instead of command line arguments or user input to control which index k is used. You may define this index during the compilation process, e.g., for GCC:

```
g++ -D INDEX=<your value> -o main main.cc
```

Remember that you have to recompile your program if you want to change this index.

Measure both the compilation time and runtime of your program from $k = 5$ to $k = 50$ in steps of 5 (under Linux you can use the "`time`" command for this task). What do you observe? Plot both times over the index k and discuss your observations. What is the explanation for the asymptotics? There are significantly faster algorithms for the computation of Fibonacci numbers[2]. Would you suggest using one of these other algorithms for the template metaprogram, the function, or both?

(b) **Prime Numbers.** Write a second template metaprogram that prints the N-th prime number. Remember that a prime number is a natural number larger than one, that is only divisible by one and itself. Here are some steps that you may follow in your implementation:

- Start with a struct `is_prime<int P>` that exports a `const static bool` that is `true` for prime numbers and false for all other numbers. A simple implementation uses the signature

```
template<int P, int K = P - 1> struct is_prime;
```

that checks whether P is not divisible by any number smaller or equal to K (except for one, of course). A recursive definition of this function is straight-forward, using the modulo operator and simple Boolean algebra. The default case then provides the desired struct.

[1] https://en.wikipedia.org/wiki/Church%E2%80%93Turing_thesis
[2] https://www.nayuki.io/page/fast-fibonacci-algorithms

- Then write a second struct `next_prime<int N>` that exports the smallest prime P with $P \geq N$. If N is prime, then this is simply N itself, else we can use recursion again by exporting the smallest prime P with $P \geq (N + 1)$. Why is this recursion guaranteed to terminate? The recursion needs to branch on whether N itself is prime or not, but both conditionals (`if/else`) and the ternary operator (`X ? Y : Z`) would always instantiate the next recursion level, whether X is true or false, leading to infinite recursion. From C++17 onwards one could use the `constexpr if` construct to only instantiate one of the two branches, thereby solving the problem, but your code should be valid C++98/03. Once again, this can be solved with a slightly extended signature:

```
template<int P, bool B = is_prime<P>::value> struct next_prime;
```

  How can you use this second (auto-evaluated) parameter to get the desired branching behavior?

- Finally, write a struct `prime<int N>` that exports the N-th prime number, making use of the other structs you have defined. This definition is also recursive. What is the base case? How can you compute the correct prime number for all the other cases recursively?

Use the aforementioned preprocessor macro trick to evaluate your template metaprogram for different values of N, and check that the right sequence is created.

Exercise 2: Matrices: An Application of SFINAE (10 points)

One of the major drawbacks of class template specializations is the need to reimplement the whole class template, even if only a single method has to be modified. This leads to significant code duplication, especially for classes that provide a large number of methods, and inconsistencies between the different specializations may occur after some time if updates aren't copied to all the different specializations.

SFINAE can be used to solve this issue: there is no explicit specialization of the class template, instead there are simply two (or more) different versions of the one method that needs to be changed, with exactly one of them being instantiable for any parameterization of the class template. This technique can be used for both normal methods and methods that are function templates themselves: if the method is a template, then it simply gains an additional template parameter for the `std::enable_if`, and if it is an ordinary function it becomes a template with a single parameter instead.

(a) Apply SFINAE to the `print()` method of your matrix implementation. As discussed in a previous exercise, the default implementation is only correct if the elements of a given matrix are actual numbers, and doesn't work anymore if the matrix is actually a block matrix, i.e., a matrix with entries that are themselves matrices. Instead of redefining the whole matrix class for such cases, introduce two versions of the `print()` method, one that is instantiated when the entries are scalars, and one that is instantiated when the entries are matrices.

  You can reuse the function bodies if you have completed the previous exercise, else you will have to implement one of them. To determine whether a given matrix is a block matrix or not, check the following properties of the entries:

  - Does the type of the entries export a number of rows? (You may have to introduce a public `enum` in your matrix class to make this available if you haven't already done so.)
  - Does the type of the entries export a number of columns? (With an `enum` as above.)

  You may assume that anything exporting these two constants is actually a matrix (and not, say, a higher-dimensional tensor or similar). Make sure that exactly one of the conditions in the `std::enable_if` clauses is true for any potential entry data type.

(b) Assume for a moment that there are both matrix classes with public enums as above, and matrices that make the number of rows / columns available via methods (which is what a matrix implementation with dynamic size would do). How would your SFINAE have to change if it should work with

  - normal entries such as numbers (`double`, `int`, ...)

- matrices that export enums as treated by your implementation
- matrices that provide this information via methods instead

You don't have to actually implement the functionality needed for the checks: assume that all required structs are already provided, and write down what the signatures of the three implementations would look like.

(c) SFINAE can also be used to provide two or more competing (essentially non-specialized) templates, which can eliminate the need for several different function overloads. Consider the multiplication operators, there are usually three different types:

- with a scalar
- with a vector
- with another matrix

The first two of these have already been implemented: for concrete types, or as templates parameterized by container types. However, the code for these three types of multiplication is usually quite generic, and would work the same way for several different types of scalars, vectors, and matrices. This makes the implementation via completely abstract function templates attractive, i.e., with the type of the second argument as template parameter. *The problem:* all three types of multiplication would have the same function signature.

Use SFINAE to work around this problem:

- The existence resp. nonexistence of which methods can be used to characterize scalars, vectors, and matrices?
- Provide traits classes `is_vector<T>` and `is_matrix<T>` that can detect whether a given type matches the vector resp. matrix interface.
- Implement all three types of multiplication as free function templates, parameterized by the type of the second argument, and use SFINAE to select the correct template version for a given type of operand.

Modify one of your existing test cases to make sure that your SFINAE construct works and picks the right template in each case.