Dr. Ole Klein, Stefan Meggendorfer
IWR, Heidelberg University

Exercise Sheet 8

Exercise 1: C++ Quiz 4 (5 points)

Here are some additional questions from https://cppquiz.org for you to answer:

Question 1: https://cppquiz.org/quiz/question/163 (automatic type deduction)

Question 2: https://cppquiz.org/quiz/question/236 (operator auto)

Question 5: https://cppquiz.org/quiz/question/112 (move semantics)

Question 3: https://cppquiz.org/quiz/question/226 (move semantics II)

Question 4: https://cppquiz.org/quiz/question/116 (perfect forwarding)

Question 1 and 2 are about automatic type deduction, with the latter being more of a peculiarity. The other three questions are about rvalue references, forwarding references, and move semantics: what one might expect to happen, and what actually happens according to the standard.

Exercise 2: Automatic Type Deduction (5 points)

Take your most recent matrix class implementation (it doesn't actually matter which one as long as it is templatized). We would like to make use of automatic type deduction at several points within the matrix class, resp. classes.

(a) Replace the return type of all the methods with `auto`. At least one method each should be of the following form:

- trailing return type with explicit specification of the resulting type
- trailing return type, declaring the return type with `decltype` and the function arguments
- no trailing return type, simply deduct the resulting type from the return statements (requires compiling with C++14 or above)

(b) Check the attributes of the class: is it possible to replace some of their types with `auto`? If yes, which ones? If not, why not?

(c) Check the local variables of the methods, and replace the explicit type with `auto` wherever possible.

(d) After having converted all possible locations to `auto`: which instances of using `auto` are actually a good idea, if any? Are there locations where this modification doesn't add any value, in your opinion? Discuss.

Exercise 3: Constant Expressions (10 points)

In a previous exercise you were tasked with writing template metaprograms to compute two number sequences, namely the Fibonacci sequence and the prime number sequence. In this exercise we will reimplement these two sequences, but with constant expressions instead of templates.

Solve the following two tasks using `constexpr` functions. Note that these functions should be valid in C++11, i.e., each function must consist of a single return statement. This means that `if`/`else` branches can't be used, you will have to rely on the ternary operator (`x ? y : z`) instead. The value of this expression is y if the condition x is true, else z.

(a) **Fibonacci Numbers.** If you solved the previous exercise, you should already have a normal function that computes the Fibonacci sequence. Modify this function so that is becomes a valid C++11 `constexpr` function (the extent of modifications depends on your chosen implementation, of course). Simply implement the function directly if you didn't do the previous exercise. Here is a quick reminder of the mathematical definition of the k-th element, $k \in \mathbb{N}$:

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

(b) **Prime Numbers.** Reimplement the prime number sequence using `constexpr` functions. You can follow the outline from the previous exercise, if you like:

- Create a `constexpr` function `is_prime (int p, int k = p-1)` that checks whether p is not divisible by any number larger than one and smaller than or equal to k (recursive function).
- Write `constexpr` functions `next_prime(int n)` and `prime(int n)` that return the next prime after n and the n-th prime, respectively.
- Note that there is a fundamental difference to the previous exercise: the ternary operator wasn't suitable for the template metaprogram, since both "branches" would be instantiated in any case (infinite recursion), but it is actually guaranteed that only one of them will be executed in the end. This means you can use the ternary operator for this version, as mentioned above.

(c) **Comparisons.** Compare the implementation effort and code complexity of the template metaprogramming and `constexpr` versions for both sequences. Which version would you prefer? Measure the compilation times of all four codes (Fibonacci and prime numbers, template metaprogramming and `constexpr`) for different inputs. What do you observe? Discuss.

Exercise 4: Linked List with Smart Pointers (5 points)

Revisit the linked list exercise: we used raw pointers back then, but we might just as well use smart pointers instead. The behavior of the linked list class will change depending on the actual choice of smart pointer. Note that one doesn't assign `nullptr` to an invalid smart pointer, or compare against this value to check validity. Instead, invalid smart pointers are simply those that don't currently store a raw pointer and evaluate to `false` when used as a Boolean expression.

(a) Try replacing the raw pointers with `std::unique_ptr<Node>`. Is this straight-forward? If not, what exactly is problematic about the chosen interface of the linked list? Remember what happened when the list was copied. What happens / would happen, if unique pointers are used instead?

(b) Replace the raw pointers with `std::shared_ptr<Node>`. What happens in this case if we copy the list, and then modify one of the copies? And what happens if one of them or both go out of scope?

(c) Turn the shared pointer version of the linked list into a doubly linked list: introduce an additional method `previous` that returns the previous node instead of the next one, and modify the `Node` class accordingly. Note that simply using shared pointers for this would produce a silent memory leak, since the resulting loops of shared pointers would artificially keep the list alive after it goes out of scope. This means you have to use a `std::weak_ptr<Node>` instead.

(d) Discuss the benefits and drawbacks of smart pointers for this application. How would you personally implement a doubly linked list, and what would be your design decisions?