Advanced Parallel Programming
Exercise 1

Fabian Czappa (fabian.czappa@tu-darmstadt.de)

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Summer term 2025
15.05.2025

Please solve the following tasks by 15.05.2025. The results are not graded, but a solution is discussed on 15.05.2025.

Task 1: A generic class

This task will focus on teaching you how to use generics in C++. The items are very finely grained. Do not let yourself be distracted in case you already know quite a lot.

1. Write a class that is templated by three generic types[1].
2. Give your class three non-static data members[2]—one of each of the generic types.
3. Give your class a static data member that is suitable for counting.
4. Write a constructor[3] for your class that takes three arguments—one of each of the generic types—all as constant references[4] and copies then to its members. Increment your counter.
5. Write a constructor for your class that takes three arguments—one of each of the generic types— as rvalues and moves[5] them to its members. Increment your counter.
6. Write a constructor for your class that takes one argument—a tuple[6] of three values. The types inside the tuple should be the three generic types of the class. Copy/Move the members of the tuple to the members of your class—depending on how you accept the tuple as argument. Increment your counter.
7. Read yourself how to use structured bindings in C++[7], and change your constructor from above to use structured bindings.
8. Define the copy[8] and move[9] constructors for your class. For each invocation, increment the counter.
9. Define the copy[10] and move[11] assignment operators for your class. For each invocation, increment the counter.
10. Define an equality operator[12] for your class. Two instances of your class are equal if, and only if, the first data members of both instances are equal.
11. Define a the smaller/larger operators for your class. Two instances of your class are smaller/larger if, and only if, the same relation holds for their first data members.

[1] https://en.cppreference.com/w/cpp/language/templates
[2] https://en.cppreference.com/w/cpp/language/data_members
[3] https://en.cppreference.com/w/cpp/language/constructor
[4] https://en.cppreference.com/w/cpp/language/reference
[5] https://en.cppreference.com/w/cpp/utility/move
[6] https://en.cppreference.com/w/cpp/utility/tuple
[7] https://en.cppreference.com/w/cpp/language/structured_binding
[8] https://en.cppreference.com/w/cpp/language/copy_constructor
[9] https://en.cppreference.com/w/cpp/language/move_constructor
[10] https://en.cppreference.com/w/cpp/language/copy_assignment
[11] https://en.cppreference.com/w/cpp/language/move_assignment
[12] https://en.cppreference.com/w/cpp/language/operator_comparison

12. In C++, you can have multiple functions with the same name, that differ only in the types of the arguments. As non-static member functions have an implicit first parameter (the object instance), the place to declare the specifiers of the type are behind the closing ‘)’, before the opening ‘{’[13]. Implicitly, every function has a ‘&’ modifier (i.e., you always call functions on references); this is also the case if you declare a function as constant.

    For each data member of your class, define the following two functions: First, a function that returns the member as a constant reference; this function should apply only to constant references of the class type. Second, a function that returns the member as an rvalue; this function should apply only to rvalues of the class type. For the second function, you can move the member element out of the class.

13. Define a static member function that returns the counter.

Solution:

```cpp
#pragma once                                                        // 1

#include <tuple>                                                    // 6
#include <utility>                                                  // 5

template <typename T1, typename T2, typename T3>                    // 1
class GenericClass {                                                // 1

public:
    GenericClass(const T1& t1, const T2& t2, const T3& t3)          // 4
        : member1(t1), member2(t2), member3(t3) {                   // 4
        counter++;                                                  // 4
    }                                                               // 4

    GenericClass(T1&& t1, T2&& t2, T3&& t3)                         // 5
        : member1(std::move(t1)),                                   // 5
        member2(std::move(t2)),                                     // 5
        member3(std::move(t3)) {                                    // 5
        counter++;                                                  // 5
    }                                                               // 5

    GenericClass(const std::tuple<T1, T2, T3>& tuple)               // 6
        : member1(std::get<0>(tuple)),                              // 6
        member2(std::get<1>(tuple)),                                // 6
        member3(std::get<2>(tuple)) {                               // 6
        counter++;                                                  // 6
    }                                                               // 6

    GenericClass(std::tuple<T1, T2, T3>&& tuple) {                  // 7
        auto&& [val1, val2, val3] = tuple;                          // 7
        member1 = std::move(val1);                                  // 7
        member2 = std::move(val2);                                  // 7
        member3 = std::move(val3);                                  // 7
        counter++;                                                  // 7
    }                                                               // 7

    GenericClass(const GenericClass& other)                         // 8
        : member1(other.member1), member2(other.member2),           // 8
        member3(other.member3) {                                    // 8
        counter++;                                                  // 8
    }                                                               // 8

    GenericClass(GenericClass&& other) noexcept                     // 8
        : member1(std::move(other.member1)),                        // 8
        member2(std::move(other.member2)),                          // 8
        member3(std::move(other.member3)) {                         // 8
        counter++;                                                  // 8
    }                                                               // 8
```

[13] See https://isocpp.org/wiki/faq/const-correctness#const-overloading

```
    GenericClass& operator=(const GenericClass& other) {            // 9
        if (this != &other) {                                       // 9
            member1 = other.member1;                                // 9
            member2 = other.member2;                                // 9
            member3 = other.member3;                                // 9
            counter++;                                              // 9
        }                                                           // 9
        return *this;                                               // 9
    }                                                               // 9

    GenericClass& operator=(GenericClass&& other) {
        if (this != &other) {                                       // 9
            member1 = std::move(other.member1);                     // 9
            member2 = std::move(other.member2);                     // 9
            member3 = std::move(other.member3);                     // 9
            counter++;                                              // 9
        }                                                           // 9
        return *this;                                               // 9
    }

    bool operator==(const GenericClass& other) const {              // 10
        return member1 == other.member1;                            // 10
    }                                                               // 10

    bool operator<(const GenericClass& other) const {               // 11
        return member1 < other.member1;                             // 11
    }                                                               // 11

    bool operator>(const GenericClass& other) const {               // 11
        return member1 > other.member1;                             // 11
    }                                                               // 11

    const T1& getMember1() const& {                                 // 12
        return member1;                                             // 12
    }                                                               // 12

    const T2& getMember2() const& {                                 // 12
        return member2;                                             // 12
    }                                                               // 12

    const T3& getMember3() const& {                                 // 12
        return member3;                                             // 12
    }                                                               // 12

    T1&& getMember1()&& {                                           // 12
        return std::move(member1);                                  // 12
    }                                                               // 12

    T2&& getMember2()&& {                                           // 12
        return std::move(member2);                                  // 12
    }                                                               // 12

    T3&& getMember3()&& {                                           // 12
        return std::move(member3);                                  // 12
    }                                                               // 12

    static unsigned int getCounter() {                              // 13
        return counter;                                             // 13
    }                                                               // 13

private:
    T1 member1;                                                     // 2
    T2 member2;                                                     // 2
    T3 member3;                                                     // 2

    static inline unsigned int counter = 0;                         // 3
```

```cpp
public:
    template <std::size_t Index>
    [[nodiscard]] constexpr auto& get()& {
        if constexpr (Index == 0) {
            return member1;
        }
        if constexpr (Index == 1) {
            return member2;
        }
        if constexpr (Index == 2) {
            return member3;
        }
    }

    template <std::size_t Index>
    [[nodiscard]] constexpr const auto& get() const& {
        if constexpr (Index == 0) {
            return member1;
        }
        if constexpr (Index == 1) {
            return member2;
        }
        if constexpr (Index == 2) {
            return member3;
        }
    }

    template <std::size_t Index>
    [[nodiscard]] constexpr auto&& get()&& {
        if constexpr (Index == 0) {
            return std::move(member1);
        }
        if constexpr (Index == 1) {
            return std::move(member2);
        }
        if constexpr (Index == 2) {
            return std::move(member3);
        }
    }

};

namespace std {
    template <typename T1, typename T2, typename T3>
    struct tuple_size<::GenericClass<T1, T2, T3>> {
        static constexpr size_t value = 3;
    };

    template <typename T1, typename T2, typename T3>
    struct tuple_element<0, ::GenericClass<T1, T2, T3>> {
        using type = T1;
    };

    template <typename T1, typename T2, typename T3>
    struct tuple_element<1, ::GenericClass<T1, T2, T3>> {
        using type = T2;
    };

    template <typename T1, typename T2, typename T3>
    struct tuple_element<2, ::GenericClass<T1, T2, T3>> {
        using type = T3;
    };

} // namespace std
```

Listing 1: A generic class

Task 2: A generic container for APP exercises

In this task, you will help me design a program to store the APP exercises. An exercise is a triple of values:

1. An exercise id, usually an unsigned integer of 32 bit[14]
2. A text for the task description, usually a string of unknown length[15]
3. a text for the solution, usually a string of unknown length

So, we can represent an exercise as the class from the previous task with the three types as proposed. The items are very finely grained. Do not let yourself be distracted in case you already know quite a lot.

1. For an easier life, define a type alias[16] for it (so you do not have to type as much).
2. Store the exercises in a vector[17].
3. Define a function that takes a vector of exercises by constant reference and checks if the exercises are sorted by using the smaller operator. The function should return a boolean indicating the sortedness.
4. Define a function that takes a vector of exercises by constant reference. If the vector is not sorted, throw a value and complain to the programmer. If the vector is sorted, return a vector of all exercise ids that occur more than once[18].
5. Define a function that takes a vector of exercises by reference and sorts its[19].
6. Define a function that takes a vector of exercises by constant reference. If the vector is not sorted, throw a value and complain to the programmer. If the vector is sorted, return a vector of the exercise ids that are missing (i.e., for which there exists two exercises, one of which has a higher id, and one of which has a lower id).
7. We saw in the refresher that moving values is oftentimes better than copying. Read yourself what even better way exists by emplacing items[20].
8. Play around with your class! Check if the functionality is correct. Also inspect how many times a class was constructed/copied/moved.

Solution:

```cpp
#include "generic-class.hpp"                                                // 1

#include <algorithm>                                                        // 5
#include <cstdint>                                                          // 1
#include <iostream>                                                         // 8
#include <string>                                                           // 1
#include <unordered_set>                                                    // 4
#include <vector>                                                           // 2

using exercise_t = GenericClass<std::uint32_t, std::string, std::string>;   // 1
using exercises_t = std::vector<exercise_t>;                                // 2

bool are_sorted(const exercises_t& exercises) {                             // 3
    for (auto i = std::size_t{ 0 }; i + 1 < exercises.size(); i++) {        // 3
        const auto smaller = exercises[i] < exercises[i + 1];               // 3
        const auto equal = exercises[i] == exercises[i + 1];                // 3
        if (!(smaller || equal)) {                                          // 3
            return false;                                                   // 3
        }                                                                   // 3
    }                                                                       // 3
```

[14] https://en.cppreference.com/w/cpp/types/integer
[15] https://en.cppreference.com/w/cpp/string/basic_string
[16] https://en.cppreference.com/w/cpp/language/type_alias
[17] https://en.cppreference.com/w/cpp/container/vector
[18] https://en.cppreference.com/w/cpp/container/unordered_set
[19] https://en.cppreference.com/w/cpp/algorithm/sort
[20] https://en.cppreference.com/w/cpp/container/vector/emplace_back

```cpp
                                                                                   // 3
    return true;                                                                   // 3
}                                                                                  // 3

std::vector<std::uint32_t> get_duplicate_ids(const exercises_t& exercises) {       // 4
    if (!are_sorted(exercises)) {                                                  // 4
        throw "I complain!";                                                       // 4
    }                                                                              // 4
                                                                                   // 4
    auto duplicate_ids = std::vector<std::uint32_t>{};                             // 4
    auto seen_ids = std::unordered_set<std::uint32_t>{};                           // 4
                                                                                   // 4
    for (const auto& [id, _1, _2] : exercises) {                                   // 4
        if (!seen_ids.contains(id)) {                                              // 4
            seen_ids.emplace(id);                                                  // 4
            continue;                                                              // 4
        }                                                                          // 4
                                                                                   // 4
        if (duplicate_ids.empty()) {                                               // 4
            duplicate_ids.emplace_back(id);                                        // 4
            continue;                                                              // 4
        }                                                                          // 4
                                                                                   // 4
        const auto last_seen_id = duplicate_ids.back();                            // 4
        if (last_seen_id != id) {                                                  // 4
            duplicate_ids.emplace_back(id);                                        // 4
            continue;                                                              // 4
        }                                                                          // 4
    }                                                                              // 4
                                                                                   // 4
    return duplicate_ids;                                                          // 4
}                                                                                  // 4

void sort_exercises(exercises_t& exercises) {                                      // 5
    std::sort(exercises.begin(), exercises.end());                                 // 5
}                                                                                  // 5

std::vector<std::uint32_t> get_missing_ids(const exercises_t& exercises) {         // 6
    if (!are_sorted(exercises)) {                                                  // 6
        throw "I complain again!";                                                 // 6
    }                                                                              // 6
                                                                                   // 6
    auto missing_ids = std::vector<std::uint32_t>{};                               // 6
    if (exercises.empty()) {                                                       // 6
        return missing_ids;                                                        // 6
    }                                                                              // 6
                                                                                   // 6
    auto expected_id = exercises.front().getMember1();                             // 6
                                                                                   // 6
    for (const auto& [id, _1, _2] : exercises) {                                   // 6
        if (id <= expected_id) {                                                   // 6
            expected_id++;                                                         // 6
            continue;                                                              // 6
        }                                                                          // 6
                                                                                   // 6
        while (id > expected_id) {                                                 // 6
            missing_ids.emplace_back(expected_id);                                 // 6
            expected_id++;                                                         // 6
        }                                                                          // 6
                                                                                   // 6
        expected_id++;                                                             // 6
    }                                                                              // 6
                                                                                   // 6
    return missing_ids;                                                            // 6
}                                                                                  // 6

int main() {
    auto exercise_2 = exercise_t{ 2, "This is the second exercise", "This is the second solution" };
    std::cout << "Total calls: " << exercise_t::getCounter() << '\n';

    auto exercises = exercises_t{ };
    std::cout << "Total calls: " << exercise_t::getCounter() << '\n';

    auto exercise_3 = exercise_t{ 3, "This is the third exercise", "This is the third solution" };
    exercises.emplace_back(exercise_3);
    std::cout << "Total calls: " << exercise_t::getCounter() << '\n';

    auto exercise_1 = exercise_t{ 1, "This is the first exercise", "This is the first solution" };
    exercises.push_back(exercise_1);
    std::cout << "Total calls: " << exercise_t::getCounter() << '\n';

    exercises.emplace_back(4, "This is the forth exercise", "This is the forth solution");
    std::cout << "Total calls: " << exercise_t::getCounter() << '\n';

    exercises.push_back(exercise_t{ 5, "This is the fifth exercise", "This is the fifth solution" });
    std::cout << "Total calls: " << exercise_t::getCounter() << '\n';

    exercises.emplace_back(std::make_tuple(3, "This is the third exercise", "This is the second version of the third solution"));
    std::cout << "Total calls: " << exercise_t::getCounter() << '\n';

    std::cout << "The order of exercises is:\n";
    for (const auto& [id, task, solution] : exercises) {
        std::cout << "Exercise " << id << ":\n";
        std::cout << "\t" << task << '\n';
        std::cout << "\t" << solution << '\n';
    }
    std::cout << "Total calls: " << exercise_t::getCounter() << '\n';

    if (are_sorted(exercises)) {
        std::cout << "The exercises are sorted\n";
    }
    else {
        std::cout << "The exercises are not sorted\n";
    }

    sort_exercises(exercises);

    if (are_sorted(exercises)) {
        std::cout << "The exercises are sorted\n";
    }
    else {
        std::cout << "The exercises are not sorted\n";
    }

    std::cout << "The order of exercises is:\n";
    for (const auto& [id, task, solution] : exercises) {
        std::cout << "Exercise " << id << ":\n";
        std::cout << "\t" << task << '\n';
        std::cout << "\t" << solution << '\n';
    }
    std::cout << "Total calls: " << exercise_t::getCounter() << '\n';

    const auto duplicate_ids = get_duplicate_ids(exercises);
    std::cout << "The duplicate ids are:\n";
    for (const auto& id : duplicate_ids) {
        std::cout << id << '\n';
    }
    std::cout << "Total calls: " << exercise_t::getCounter() << '\n';

    const auto missing_ids = get_missing_ids(exercises);
    std::cout << "The missing ids are:\n";
    for (const auto& id : missing_ids) {
        std::cout << id << '\n';
    }
    std::cout << "Total calls: " << exercise_t::getCounter() << '\n';

    return 0;
```

```
}
```

Listing 2: Exercises en masse