

## Exercise Sheet 10

---

### Exercise 1: Variadic Templates: Tuples (10 points)

The lecture discussed a custom implementation of arrays using recursive data types:

- a zero-element array is simply an empty struct
- an  $(N+1)$ -element array is an  $N$ -element array (public inheritance) storing an additional element
- the number of elements  $N$  and the type of elements  $T$  are template parameters
- a templated `entry` method returns the  $M$ -th entry,  $0 \leq M < N$ , of the array

Back then we noted that tuples could also be implemented this way, but only if variadic templates are available for the arbitrarily long list of stored types. With variadic templates having been introduced in the lecture, you are tasked with implementing such a recursive variadic class template that defines tuples.

Proceed as follows:

- Define a class template `Tuple` that accepts an arbitrary number of types, but *don't* provide an implementation. Every possible case is handled through a specialization, but you will see that you need this general declaration, because otherwise you will not be able to match the template parameter signatures.
- A tuple containing  $N + 1$  variables of up to  $N + 1$  different types consists of a variable of the first type and a tuple of the remaining  $N$  variables with associated types (again via public inheritance). Create a template specialization for the case  $N > 0$ , with an appropriate data member, base class specification, and constructor.
- Write a method function template `entry<int M>()` that returns the  $M$ -th entry ( $0 \leq M < N$ ). Two cases are possible:
  - $M = 0$ : The zeroth entry is the data member, so we can simply return that.
  - $M \neq 0$ : The requested entry is part of the base class, so we forward the call and return what the base class gives us.

Use SFINAE to differentiate between these two cases, and handle them separately. Note that we don't actually know the returned type in the second case. Even knowing the return type of the direct base class is not enough, because the actual data member might be in some base class of the base class. One possible solution would be an internal template metaprogram that extracts the correct type. Is there a simpler solution?

- The class `std::tuple` has two different access methods: via index as above, or via requested type, if the latter is unique across the tuple. Provide this functionality for the custom tuple class, i.e., write a method function template `entry<U>()` that returns the entry of type `U`. There are two possibilities:
  - The data member has type `U` and that type doesn't appear in the base class: simply return the data member as above.
  - The data member doesn't have type `U`: hand the request over to the base class.

Use SFINAE to differentiate between these two cases, and handle them separately. Note that we don't cover the case where `U` appears more than once explicitly — it's okay if this just results

in a compilation error. You will need information about contained types: write an internal class template `struct contains<U>` that exports `true` if the tuple or its base classes contain a data member of type `U`, else `false`. Use normal logic operators (`&&` and `||`) in the SFINAE and internal struct, or the C++17 class templates `std::conjunction` and `std::disjunction` if you want.

- (e) Provide a base case, which is an empty tuple. Just as with the custom arrays, this is essentially an empty struct, but you will have to provide a base case for the internal `contains` struct.

The complete implementation contains four different `entry` methods, and the SFINAEs make sure that exactly one of them matches in any situation, whether an index is passed as parameter or a type. Note that in contrast to our custom implementation, the class `std::tuple` uses a free function template named `std::get` for access. The main reason is that our version becomes slightly awkward to use within templates, where one has to specify that the method is, indeed, also a template: `t.template entry<2>()` or similar.

## Exercise 2: Concurrency with Threads

(10 points)

Use threads to implement a parallelized scalar product of two vectors. You may use any vector class: the numerical ones of the lecture, a `std::vector`, or even a plain old C-style array. Alternatively, you may provide a function template to handle all these separate cases simultaneously.

Create the following four versions, each operating on a subset of the vector components:

- (a) A version using mutexes and locks, directly adding the occurring products to the result.
- (b) A second version using mutexes and locks, computing the local scalar product of the indices belonging to the thread, and then adding those local products to the result.
- (c) A variant of the first version, using atomics instead of mutexes.
- (d) A variant of the second version, using atomics instead of mutexes.

The main program should divide any given pair of two vectors into segments of more or less equal size and hand them two a matching number of worker threads. Test your four versions on large vectors, and measure the required time. What happens for larger numbers of threads, or what do you expect would happen, in case the number of parallel threads you can start is very limited?

Assume for a moment that the number of threads is so large that even the second and the fourth version suffer from congestion (this is a real problem, albeit in the context of message passing on very large super clusters). What could be done to alleviate the problem? You don't need to implement the solution.

In a real numerical program, the scalar product would be used for subsequent computations. An example is the Conjugate Gradients method, where the expression for the step direction of the scheme contains a scalar product. In a parallelized version of such a program, the threads would not just compute the scalar product, but perform other operations before and after. It is obviously very important to make sure that the scalar product has been fully computed before using the result in other computations.

- (e) Create a synchronization point (barrier) for the first two versions, e.g., using counters and condition variables or something similar. After this point, have each thread print the scalar product as a stand-in for further computations, and check that all threads report the same value.
- (f) Inform yourself about memory order models and their consequences, and try to create a similar barrier for the atomics versions, e.g., using atomic counter variables and a Boolean flag that uses `load` and `store`. Print and check the results as above.