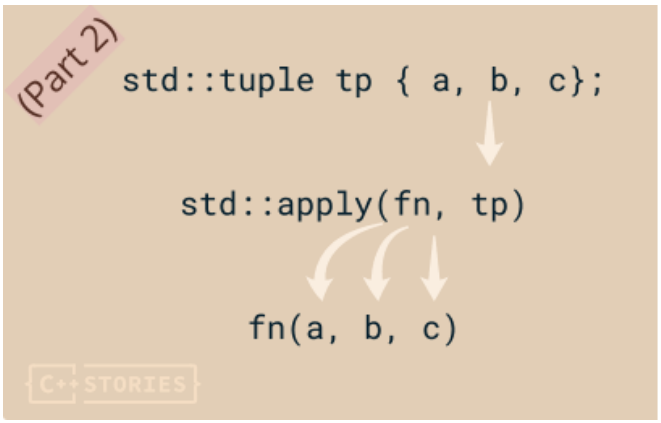
Last Update: 14 February 2022

C++ Templates: How to Iterate through std::tuple: std::apply and More (.)



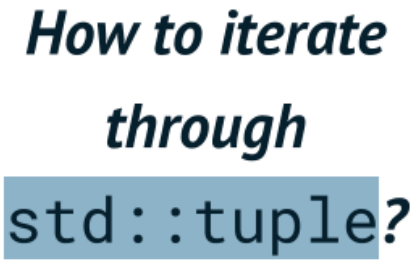


Table of Contents

(https://www.cppstories.com/index.xml) (https://twitter.com/fenbf)
(https://www.linkedin.com/in/bartlomiejfilipek)

In the previous article on the tuple iteration, we covered the basics. As a result, we implemented a function template that took a tuple and could nicely print it to the output. There was also a version with `operator <<`.

Today we can go further and see some other techniques. The first one is with `std::apply` from C++17, a helper function for tuples. Today's article will also cover some strategies to make the iteration more generic and handle custom callable objects, not just printing.

> *This is the second part of the small series. See the first article here (https://www.cppstories.com/2022/tuple-iteration-basics/) where we discuss the basics.*

std:apply approach

A handy helper for `std::tuple` is the `std::apply` function template that came in C++17. It takes a tuple and a callable object and then invokes this callable with parameters fetched from the tuple.

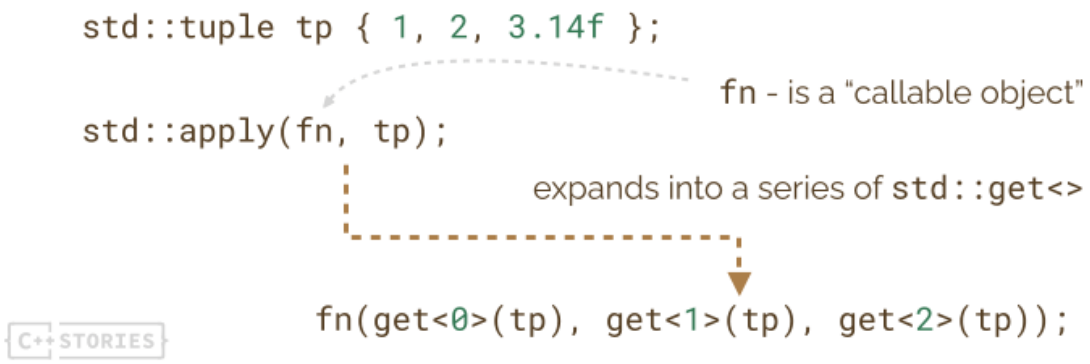
Here's an example:

```
#include <iostream>
#include <tuple>

int sum(int a, int b, int c) {
    return a + b + c;
}

void print(std::string_view a, std::string_view b) {
    std::cout << "(" << a << ", " << b << ")\n";
}

int main() {
    std::tuple numbers {1, 2, 3};
    std::cout << std::apply(sum, numbers) << '\n';

    std::tuple strs {"Hello", "World"};
    std::apply(print, strs);
}
```

Play @Compiler Explorer (https://godbolt.org/z/vvc8fPTxx)

As you can see, `std::apply` takes `sum` or `print` functions and then "expands" tuples and calls those functions with appropriate arguments.

Here's a diagram showing how it works:



Ok, but how does it relate to our problem?

The critical thing is that `std::apply` hides all index generation and calls to `std::get<>`. That's why we can replace our printing function with `std::apply` and then don't use `index_sequence`.

The first approach - working?

The first approach that came to my mind was the following - create a variadic function template that takes `Args...` and pass it to `std::apply`:

```
template <typename... Args>
void printImpl(const Args&... tupleArgs) {
    size_t index = 0;
    auto printElem = [&index](const auto& x) {
        if (index++ > 0)
            std::cout << ", ";
        std::cout << x;
    };

    (printElem(tupleArgs), ...);
}

template <typename... Args>
void printTupleApplyFn(const std::tuple<Args...>& tp) {
    std::cout << "(";
    std::apply(printImpl, tp);
    std::cout << ")";
}
```


Looks… fine… right?

The problem is that it doesn't compile :)

GCC or Clang generates some general error which boils down to the following line:

```
candidate template ignored: couldn't infer template argument '_Fn
```

But how? Why cannot the compiler get the proper template parameters for `printImpl`?

The problem lies in the fact that out `printImpl` is a variadic function template, so the compiler has to instantiate it. The instantiation doesn't happen when we call `std::apply`, but inside `std::apply`. The compiler doesn't know how the callable object will be called when we call `std::apply`, so it cannot perform the template deduction at this stage.

We can help the compiler and pass the arguments:

```
#include <iostream>
#include <tuple>

template <typename... Args>
void printImpl(const Args&... tupleArgs) {
    size_t index = 0;
    auto printElem = [&index](const auto& x) {
        if (index++ > 0)
            std::cout << ", ";
        std::cout << x;
        };

    (printElem(tupleArgs), ...);
}

template <typename... Args>
void printTupleApplyFn(const std::tuple<Args...>& tp) {
    std::cout << "(";
    std::apply(printImpl<Args...>, tp); // <<
    std::cout << ")";
}

int main() {
    std::tuple tp { 10, 20, 3.14};
    printTupleApplyFn(tp);
}
```

Play @Compiler Explorer (https://godbolt.org/z/WTscsnvnz).

In the above example, we helped the compiler to create the requested instantiation, so it's happy to pass it to `std::apply`.

But there's another technique we can do. How about helper callable type?

```
struct HelperCallable {
    template <typename... Args>
    void operator()(const Args&... tupleArgs)  {
        size_t index = 0;
        auto printElem = [&index](const auto& x) {
            if (index++ > 0)
                std::cout << ", ";
            std::cout << x;
        };

        (printElem(tupleArgs), ...);
    }
};

template <typename... Args>
void printTupleApplyFn(const std::tuple<Args...>& tp) {
    std::cout << "(";
    std::apply(HelperCallable(), tp);
    std::cout << ")";
}
```

Can you see the difference?

Now, what we do, we only pass a `HelperCallable` object; it's a concrete type so that the compiler can pass it without any issues. No template parameter deduction happens. And then, at some point, the compiler will call `HelperCallable(args...)`, which invokes `operator()` for that struct. And it's now perfectly fine, and the compiler can deduce the types. In other words, we deferred the problem.

So we know that the code works fine with a helper callable type… so how about a lambda?

```
#include <iostream>
#include <tuple>

template <typename TupleT>
void printTupleApply(const TupleT& tp) {
    std::cout << "(";
    std::apply([](const auto&... tupleArgs) {
                size_t index = 0;
                auto printElem = [&index](const auto& x) {
                    if (index++ > 0)
                        std::cout << ", ";
                    std::cout << x;
                };

                (printElem(tupleArgs), ...);
            }, tp
        )
    std::cout << ")";
}

int main() {
    std::tuple tp { 10, 20, 3.14, 42, "hello"};
    printTupleApply(tp);
}
```

Play @Compiler Explorer (https://godbolt.org/z/xvvacxz8c).

Also works! I also simplified the template parameters to just `template <typename TupleT>`.

As you can see, we have a lambda inside a lambda. It's similar to our custom type with `operator()`. You can also have a look at the transformation through C++ Insights: this link (https://cppinsights.io/s/399f54bc)

Printing simplification

Since our callable object gets a variadic argument list, we can use this information and make the code simpler.

Thanks PiotrNycz (https://github.com/fenbf/cppstories-discussions/issues/75#issuecomment-1039487071) for pointing that out.

The code inside the internal lambda uses `index` to check if we need to print the separator or not - it checks if we print the first argument. We can do this at compile time:

```
#include <iostream>
#include <tuple>

template <typename TupleT>
void printTupleApply(const TupleT& tp) {
    std::apply
        (
            [](const auto& first, const auto&... restArgs)
            {
                auto printElem = [](const auto& x) {
                    std::cout << ", " << x;
                };
                std::cout << "(" << first;
                (printElem(restArgs), ...);
            }, tp
        );
    std::cout << ")";
}

int main() {
    std::tuple tp { 10, 20, 3.14, 42, "hello"};
    printTupleApply(tp);
}
```

Play @Compiler Explorer (https://godbolt.org/z/eMv718d9E).

This code breaks when tuple has no elements - we could fix this by checking its size in `if constexpr`, but let's skip it for now.

Making it more generic

So far we focused on printing tuple elements. So we had a "fixed" function that was called for each argument. To go further with our ideas, let's try to implement a function that takes a generic callable object. For example:

```
std::tuple tp { 10, 20, 30.0 };
printTuple(tp);
for_each_tuple(tp, [](auto&& x){
    x*=2;
});
printTuple(tp);
```

Let's start with the approach with index sequence:

```
template <typename TupleT, typename Fn, std::size_t... Is>
void for_each_tuple_impl(TupleT&& tp, Fn&& fn,
std::index_sequence<Is...>) {
    (fn(std::get<Is>(std::forward<TupleT>(tp))), ...);
}

template <typename TupleT, typename Fn,
       std::size_t TupSize =
std::tuple_size_v<std::remove_cvref_t<TupleT>>>
void for_each_tuple(TupleT&& tp, Fn&& fn) {
    for_each_tuple_impl(std::forward<TupleT>(tp), std::forward<Fn>(fn),
                        std::make_index_sequence<TupSize>{});
}
```

What happens here?

First, the code uses universal references (forwarding references) to pass tuple objects. This is needed to support all kinds of use cases - especially if the caller wants to modify the values inside the tuple. That's why we need to use `std::forward` in all places.

But why did I use `remove_cvref_t`?

On std::decay and remove ref

As you can see in my code I used:

```
std::size_t TupSize = std::tuple_size_v<std::remove_cvref_t<TupleT>>
```

This is a new helper type from the C++20 trait that makes sure we get a "real" type from the type we get through universal reference.

Before C++20, you can often find `std::decay` used or `std::remove_reference`.

Here's a good summary from a question about tuple iteration link to Stackoverflow (https://stackoverflow.com/questions/41199709/is-there-a-tuple-for-each-that-returns-a-tuple-of-all-values-returned-from-the/41200494?noredirect=1#comment125571672_41200494):

> *As T&& is a forwarding reference, T will be `tuple<...>&` or `tuple<...> const&` when an lvalue is passed in; but `std::tuple_size` is only specialized for `tuple<...>`, so we must strip off the reference and possible const. Prior to C++20's addition of `std::remove_cvref_t`, using `decay_t` was the easy (if overkill) solution.*

Generic `std::apply` version

We discussed an implementation with index sequence; we can also try the same with std::apply. Can it yield simpler code?

Here's my try:

```
template <typename TupleT, typename Fn>
void for_each_tuple2(TupleT&& tp, Fn&& fn) {
    std::apply
    (
        [&fn](auto&& ...args)
        {
            (fn(args), ...);
        }, std::forward<TupleT>(tp)
    );
}
```

Look closer, I forgot to use `std::forward` when calling `fn`!

We can solve this by using template lambdas available in C++20:

```
template <typename TupleT, typename Fn>
void for_each_tuple2(TupleT&& tp, Fn&& fn) {
    std::apply
    (
        [&fn]<typename ...T>(T&& ...args)
        {
            (fn(std::forward<T>(args)), ...);
        }, std::forward<TupleT>(tp)
    );
}
```


Play @Compiler Explorer (https://godbolt.org/z/jn1MxcEqh)

Additionally, if you want to stick to C++17, you can apply `decltype` on the arguments:

```
template <typename TupleT, typename Fn>
void for_each_tuple2(TupleT&& tp, Fn&& fn) {
    std::apply
    (
        [&fn](auto&& ...args)
        {
            (fn(std::forward<decltype(args)>(args)), ...);
        }, std::forward<TupleT>(tp)
    );
}
```

Play with code @Compiler Explorer (https://godbolt.org/z/rMehff8zo).

Return value

https://godbolt.org/z/1f3Ea7vsK (https://godbolt.org/z/1f3Ea7vsK)

Summary

It was a cool story, and I hope you learned a bit about templates.

The background task was to print tuples elements and have a way to transform them. During the process, we went through variadic templates, index sequence, template argument deduction rules and tricks, std::apply, and removing references.

I'm happy to discuss changes and improvements. Let me know in the comments below the article about your ideas.

See the part one here: C++ Templates: How to Iterate through std::tuple: the Basics - C++ Stories (https://www.cppstories.com/2022/tuple-iteration-basics/).

References:

- Effective Modern C++ (https://amzn.to/3GMx8Uu) by Scott Meyers
- C++ Templates: The Complete Guide (2nd Edition) (http://amzn.to/2wtoENU) by David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor

Similar Articles:

- C++ Templates: How to Iterate through std::tuple: the Basics (/2022/tuple-iteration-basics/)
- constexpr vector and string in C++20 and One Big Limitation (/2021/constexpr-vecstr-cpp20/)
- C++ Return: std::any, std::optional, or std::variant? (/2021/sphero-cpp-return/)
- C++20 Oxymoron: constexpr virtual (/2021/constexpr-virtual/)
- 12 Different Ways to Filter Containers in Modern C++ (/2021/filter-cpp-containers/)

Tags: cpp (/tags/cpp), cpp17 (/tags/cpp17), cpp20 (/tags/cpp20), standard library (/tags/standard-library), templates (/tags/templates),

11 Comments - *powered by utteranc.es*

iPMSoftware commented on Feb 14, 2022

Hi Bartlomiej. I was doing some iteration over tuple like last week too.
You can check it on
https://github.com/iPMSoftware/CppFunAndLearn/blob/master/template_recursive_iteration_over_tuple/m
p

PiotrNycz commented on Feb 14, 2022

This HelperCallable looks familiar, it looks like generic lambda,
So I'd just use lambda in such case:

```
auto printImpl = [](const auto&... tupleArgs) {
    size_t index = 0;
    auto printElem = [&index](const auto& x) {
        if (index++ > 0)
            std::cout << ", ";
```

```
        std::cout << x;
        };

    (printElem(tupleArgs), ...);
};
```

But actually this is not a real problem -- since you have types (Args...) for printImpl to instantiate:

```
template <typename... Args>
void printTupleApplyFn(const std::tuple<Args...>& tp) {
    std::cout << "(";
    std::apply(printImpl<Args...>, tp); // (!)
    std::cout << ")";
}
```

fenbf commented on Feb 14, 2022

Thanks @iPMSoftware, I'll see your implementation. Do you have some different techniques used there?

Thanks @PiotrNycz - unfortunately even with Args... it doesn't compile: `std::apply(printImpl<Args...>, tp);` see here https://godbolt.org/z/YWEn3YEMq

PiotrNycz commented on Feb 14, 2022

@fenbf It compiles - you have just trivial error in functionname - see https://godbolt.org/z/WTscsnvnz ((printTupleApplyFn, not printTupleApply)) - missing Fn suffix

🚀 1

fenbf commented on Feb 14, 2022

@PiotrNycz ah, eh, wow! yes, that's the bug :) Thanks for pointing this out, I'll update the article

PiotrNycz commented on Feb 14, 2022

It is always nice to help.

As a bonus, one idea how to print nicely all of these tuple elementa w/o extra additional lambda/function r
index:

```
template <typename Tuple>
void printTupleApply(const Tuple& tp) {
    std::cout << "(";
```

```
    if constexpr (std::tuple_size_v<TupleT> >= 0u)
        std::apply([](const auto& a0, const auto&... a1) {
                std::cout << a0;
                ((std::cout << ", " << a1), ...);
            }, tp);
    std::cout << ")";
}
```

👍 1

fenbf commented on Feb 15, 2022

Thanks @PiotrNycz I've just updated the article with a new section with that code.

👍 1

PiotrNycz commented on Feb 15, 2022

Everybody makes mistakes, including me ;) It should be `> 0` not `>= 0` in my `if constexpr` condition. E noticed you skipped that part, a wise move - the more focused article, the better. BTW, nice book "C++17 detail" - I have my own copy.

fenbf commented on Feb 15, 2022

@PiotrNycz heh :) Having an empty tuple is probably very rare, maybe it could be even solved by having special function template specialization to report a warning in that case... or ignore it. It's probably only ne for generic code in some libraries.
thanks for a good opinion about the book.

👍 1

tahsinkose commented 3 months ago

Here is a quite simple application, which mimics a stream of heterogeneous data and still works with the compatible sub-content. It uses what has been discussed in this post and `if constexpr` :

```
std::tuple tp{10, 20, 3.14, "Hello", 42};
auto mult = [](auto&& x){
    if constexpr (std::is_arithmetic_v<std::remove_cvref_t<decltype(x)>>) {
```

```
        x *= 2;
      }
  };
  printTuple(tp);
  for_each_tuple(tp, mult);
  printTuple(tp);
```

I tried to do this with `concepts`, but couldn't manage as it reports `couldn't deduce template parameter 'Fn'` on `multiply` function. See the snippet:

```
emplate <class T, class U>
concept Multipliable = requires(T a, U b) {
  a*b;
};

template <class T, class U>
```

PiotrNycz commented 3 months ago

@tahsinkose In C++ a function template is not a thing you can pass as argument to other functions. You c pass function template instantiation (as well as objects, variables other functions (pointers to functions). H you need to wrap this function template with lambda:

```
for_each_tuple(tp, [] (auto&& ...args) {
         multiply(std::forward<decltype(args)>(args)...); } );
```

👍 1

Write Preview

Sign in to comment

Styling with Markdown is supported

Sign in with G

<< Modern C++ For Absolute Beginners, Book Review
(https://www.cppstories.com/2022/moderncpp-beginners-book/)

(https://www.cppstories.com/index.xml) (https://twitter.com/fenbf) C++ Templates: How to Iterate through std::tuple: the Basics >>
(https://www.linkedin.com/in/bartlomiejfilipek/) (https://www.cppstories.com/2022/tuple-iteration-basics/)