template/typename

When to (not) use them and why

WHY std::get IS A FREE FUNCTION?

```
std::vector<char> vector{‘a’, ‘b’, ‘c’};
std::cout << vector.at(1);          // fine, prints ‘b’

std::tuple<char, char, char> tuple{‘a’, ‘b’, ‘c’};
std::cout << std::get<1>(tuple);    // fine, prints ‘b’
std::cout << tuple.get<1>();        // this doesn’t work
```

What is the rationale behind not having a member function template?

```
template<std::size_t index>
/* return type */ get() const;
```

SIMPLIFIED IMPLEMENTATION

```
template<class... Ts> class my_tuple {
public:
    template<std::size_t index>
    auto get() {
        return std::get<index>(tuple_);
    }

private:
    std::tuple<Ts...> tuple_;
};
```

SAMPLE USAGE

```
template<std::size_t index>
void print_nth(my_tuple<char, char, char> tuple) {
    std::cout << tuple.get<index>();    // this works...
}
```

Let's generalize:

```
template<std::size_t index, class... Ts>
void print_nth(my_tuple<Ts...> tuple) {
    std::cout << tuple.get<index>();    // ...but this doesn’t, why?
}
```

ERROR MESSAGES

Live example: https://godbolt.org/z/qjh55o

GCC error message:

```
expected primary-expression before ')' token
    std::cout << tuple.get<index>();
                                  ^
```

Clang error message is more helpful:

```
missing 'template' keyword prior to dependent template name 'get'
    std::cout << tuple.get<index>();
                       ^  ~~~~~~~~
```

DEPENDENT NAMES

```
template<std::size_t index, class... Ts>
void print_nth(my_tuple<Ts...> tuple) {
    std::cout << tuple.get<index>();
}
```

Names can be **dependent** and **non-dependent**. The name `get` is a dependent name – it depends on template parameters `Ts...`.

The general rule is: a dependent name is **not** considered to be a template name unless the disambiguation keyword `template` is used.

```
tuple.get < index > ();
          ^ “<“ means “less than“
```

TWO-PHASE NAME LOOKUP

The C++ standard requires that all names in templates are resolved in two phases.

The first phase corresponds to an **uninstantiated** template, when no information about template parameters is available. A compiler cannot assume that `get` is a template name, because `my_tuple` could be specialized:

```
template<>
class my_tuple<some_type> {
public:
    static constexpr int get = 0;
};
```

THE DIRECT SOLUTION

We have to add the `template` keyword before a dependent name:

```
template<std::size_t index, class... Ts>
void print_nth(my_tuple<Ts...> tuple) {
    std::cout << tuple.template get<index>();
}

template<std::size_t index, class... Ts>
void print_nth(my_tuple<Ts...>* tuple) {
    std::cout << tuple->template get<index>();
}
```

HOW CAN WE AVOID template?

```
auto get(std::size_t index) {
    return std::get<index>(tuple_);  // doesn’t work, index is not
}                                    // a constant expression
```

We need something like a constant expression function parameter:

```
auto get(constexpr std::size_t index) {
    return std::get<index>(tuple_);  // not yet in the language,
}                                    // see P1045
```

INTEGRAL CONSTANT WRAPPERS

Let’s wrap an index value into a type:

```
    template<std::size_t index>
    auto get(std::integral_constant<std::size_t, index>) {
        return std::get<index>(tuple_);
    }

template<std::size_t index, class... Ts>
void print_nth(my_tuple<Ts...> tuple) {
    std::cout << tuple.get(
        std::integral_constant<std::size_t, index>{});
}
```

INTEGRAL CONSTANT WRAPPERS

We can simplify code with a type alias and a variable template:

```
template<std::size_t index>
using my_size_t = std::integral_constant<std::size_t, index>;

template<std::size_t index>
inline constexpr auto my_size_c = my_size_t<index>{};

    template<std::size_t index>
    auto get(my_size_t<index>) const;
```

And then: `std::cout << tuple.get(my_size_c<index>);`

CONSTANT WRAPPERS IN BOOST

This is already available in Boost.Mpl and Boost.Hana:

`boost::mpl::size_t<index>`

`boost::hana::size_t<index>` and `boost::hana::size_c<index>`

Boost.Hana provides its own tuples and more powerful tools:

```
using namespace boost::hana::literals;
auto tuple = boost::hana::make_tuple(‘a’, ‘b’, ‘c’);
std::cout << boost::hana::reverse(tuple)[0_c];  // prints ‘c’
```

FREE FUNCTIONS

We can avoid `template` keyword by using free functions (the approach from the standard library):

```
template<class... Ts> class my_tuple;

template<std::size_t index, class... Ts>
auto get(my_tuple<Ts...>) {
    /* ... */
}

my_tuple<char, char, char> tuple;
std::cout << get<1>(tuple);
```

ARGUMENT-DEPENDENT LOOKUP

Sometimes we can omit namespace qualification:
ADL – argument dependent lookup:

```
std::vector<char> vec{‘a’, ‘b’, ‘c’};
std::cout << /* std:: */ size(vec);  // fine, ADL finds std::size

std::tuple<char, char, char> tuple{‘a’, ‘b’, ‘c’};
std::cout << get<0>(tuple);          // ADL doesn’t work, need std::
```

Starting with C++20, ADL will work with function templates:

```
std::cout << get<0>(tuple);          // Works in C++20, P0846
```

DEPENDENT TYPES

Let’s add a metafunction to return n-th element type:

```
template<class... Ts> class my_tuple {
public:
    template<std::size_t index>
    using element_type =
        std::tuple_element_t<index, std::tuple<Ts...>>;

private:
    std::tuple<Ts...> tuple_;
};
```

SAMPLE USAGE

```
template<std::size_t index>
void print_nth_type(my_tuple<char, char, char> tuple) {
    using T = decltype(tuple);  // = my_tuple<char, char, char>
    std::cout << typeid(T::element_type<index>).name();  // works
}

template<std::size_t index, class... Ts>
void print_nth(my_tuple<Ts...> tuple) {
    using T = decltype(tuple);  // = my_tuple<Ts...>
    std::cout << typeid(T::element_type<index>).name();  // fails
}
```

THE DIRECT SOLUTION

The same rule works for dependent types: a dependent name is **not** considered to be a type name unless the disambiguation keyword `typename` is used.

We have to add the `template` keyword before a dependent template name and a `typename` keyword before a dependent type:

```
template<std::size_t index, class... Ts>
void print_nth_type(my_tuple<Ts...> tuple) {
    using T = decltype(tuple);
    using Element_type = typename T::template element_type<index>;
    std::cout << typeid(Element_type).name();
}
```

HOW CAN WE AVOID typename?

We can avoid typename keyword by using out of class type alias:

```
template<std::size_t index, class Tuple>
using element_type = /* implementation */;

template<std::size_t index, class... Ts>
void print_nth_type(my_tuple<Ts...> tuple) {
    using T = decltype(tuple);
    std::cout << typeid(element_type<index, T>).name();
}
```

WHEN typename IS NOT NEEDED?

There are two exceptions to the general rule (before C++20):

```
template<class T>
class derived : T::type {     // (1) base class
public:
    derived() : T::type()     // (2) member initializer list
    {}
};
```

Moreover, typename is not allowed in these contexts!

TWO-PHASE LOOKUP AND MSVC

Before MSVS 2017 15.3, two-phase name lookup was not implemented in MSVC. Templates were consumed as a stream of tokens and all names were resolved only during template instantiation.

As a result, no `typename` and `template` keywords were need for dependent types and template names because at that time there was enough context to distinguish values, types and template names. This can drastically change meaning of some code (examples can be found here).

Compiler switch in MSVS 2017:

`/permissive-` enables two-phase name lookup

Starting with MSVS 2019, two-phase name lookup is enabled by default.

`/permissive` disables two-phase name lookup

WHAT'S NEW IN C++20?

In contexts where only a type name can appear, no `typename` is required.

Works:

```
template<class T>
T::value_type foo();

template<class T>
struct my_struct {
    T::value_type data_;
};
```

Doesn't work:

```
template<class T>
int foo(T::value_type);

template<class T>
void foo() {
    T::value_type data;
}
```

SOME REFERENCES

- E.Bendersky. *Dependent name lookup for C++ templates*
  https://eli.thegreenplace.net/2012/02/06/dependent-name-lookup-for-c-templates
- *Two-phase name lookup for C++ templates – Why?*
  https://stackoverflow.com/q/12561544
- *T.Gani, S.Lavavej et al. Two-phase name lookup support comes to MSVC*
  https://devblogs.microsoft.com/cppblog/two-phase-name-lookup-support-comes-to-msvc/
- D.Stone. *`constexpr` function parameters*
  https://wg21.link/p1045
- N.Ranns, D.Vandevoorde. *Down with `typename`!*
  https://wg21.link/p0634
- J.Spicer. *ADL and function templates that are not visible*
  https://wg21.link/p0846