A new look
on template meta-programming

Meeting C++ 2018, Berlin

Ivan Čukić

ivan@cukic.co
http://cukic.co

kde.org

Disclaimer

> Make your code readable. Pretend the next person who looks at your code is a psychopath and they know where you live.
>
> Philip Wadler

METAS

Meta-programming

It is very difficult to make something **not** to be Turing-complete

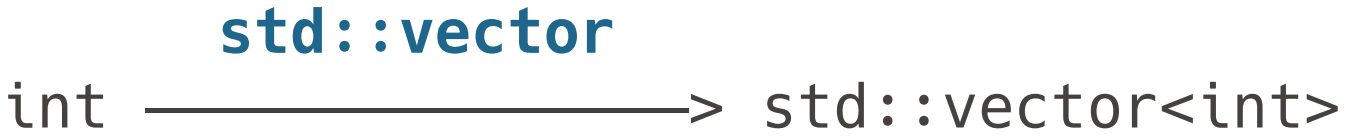
---

a famous C++ guru

Meta-functions

- Evaluated at compile-time
- No input or output
- Not only for mapping values to values

Meta-functions



Terse syntax

```
template <typename T>
using inner_type_t =
          typename T::value_type;
```

```
inner_type_t<std::vector<int>>
```

Terse syntax

```
template <typename T>
using inner_type_t =
          typename T::value_type;
```

function argument

```
inner_type_t<std::vector<int>>
```

Terse syntax

```
template <typename T>
using inner_type_t =
          typename T::value_type;
```

function name

```
inner_type_t<std::vector<int>>
```

Terse syntax

```
template <typename T>
using inner_type_t =
          typename T::value_type;
```

function result

```
inner_type_t<std::vector<int>>
```

Terse syntax

```
template <typename T>
using inner_type_t =
          typename T::value_type;
```

```
inner_type_t<std::vector<int>>
```

function call

General syntax

```
template <typename T>
class inner_type {
    using type = typename T::value_type;
};
```

```
inner_type<std::vector<int>>
inner_type<std::vector<int>>::type
```

General syntax

```
template <typename T>
class inner_type {
    using type = typename T::value_type;
};
```

function argument

```
inner_type<std::vector<int>>
inner_type<std::vector<int>>::type
```

General syntax

```
template <typename T>
class inner_type {
    using type = typename T::value_type;
};
```

function name

```
inner_type<std::vector<int>>
inner_type<std::vector<int>>::type
```

General syntax

```
template <typename T>
class inner_type {
    using type = typename T::value_type;
};
```

function result

```
inner_type<std::vector<int>>
inner_type<std::vector<int>>::type
```

General syntax

```
template <typename T>
class inner_type {
    using type = typename T::value_type;
};
```

```
inner_type<std::vector<int>>
inner_type<std::vector<int>>::type
```

function call

General syntax

```
template <typename T>
class inner_type {
    using type = typename T::value_type;
};
```

```
inner_type<std::vector<int>>
inner_type<std::vector<int>>::type
```

accessing the result

General syntax

```
template <typename T,
          typename Expected = void>

class expect_inner {
    using type =
            typename T::value_type;
    constexpr bool value =
            std::is_same_v<type, Expected>;
};
```

function arguments with defaults

```
expect_inner<std::vector<int>>::type
expect_inner<std::vector<int>>::value
```

General syntax

```
template <typename T,
          typename Expected = void>

class expect_inner {
    using type =                                  multiple
            typename T::value_type;               named
    constexpr bool value =                        function
            std::is_same_v<type, Expected>;       results
};
```

```
expect_inner<std::vector<int>>::type
expect_inner<std::vector<int>>::value
```

General syntax

```
template <typename T,
          typename Expected = void>

class expect_inner {
    using type =
            typename T::value_type;
    constexpr bool value =
            std::is_same_v<type, Expected>;
};
```

```
expect_inner<std::vector<int>>::type
expect_inner<std::vector<int>>::value
```

accessing named results

Terse accessors

```
template <typename T,
          typename Expected = void>
class expect_inner {
    using type = …
    constexpr bool value = …
};
```

```
template <typename T, typename E = void>
using expect_inner_t =
    typename expect_inner<T, E>::type;

template <typename T, typename E = void>
constexpr bool expect_inner_v =
    expect_inner<T, E>::value;
```

Meta-functions

Functions that work on types.

- Transforming types to types
- Mapping types to values
- Mapping values to types

What are they useful for?

Deducing types

```
template <typename C>
??? sum(const C& collection)
{
    return std::accumulate(…);
}
```

What is the return type?

Deducing types

- Add a template parameter for the user to explicitly state the return type
- Add the `init` function argument like `std::accumulate` has
- Or do something better…

Deducing types

```
template <typename C,
          typename Val = inner_type_t<C>>
Val sum(const C& collection)
{
    const auto begin = std::begin(collection);
    const auto end = std::end(collection);

    if (begin == end) {
        return Val{}; // or throw an exception
    } else {
        const auto init = *begin;
        return std::accumulate(++begin, end, init);
    }
}
```

Works for any sane sequence collection.

Deducing types

What about collections that do not have a `value_type` nested type definition?

Deducing types

```
template <typename T>
using inner_type_t =
        decltype(*std::begin(std::declval<T>()));
```

Works for any sequence collection with iterators. But...

Writing out the exact types

```
template <typename ...T>
class print_types;
```

```
print_types<
    inner_type_t<std::vector<bool>>,
    inner_type_t<std::vector<int>>
    >();
```

Writing out the exact types

```
template <typename ...T>
class print_types;
```

```
print_types<
    inner_type_t<std::vector<bool>>,
    inner_type_t<std::vector<int>>
    >();

gcc: invalid use of incomplete type
'class print_type<bool, const int&>'
```

Deducing types

```
template <typename T>
using inner_type_t =
    std::remove_cvref_t<
        decltype(*std::begin(std::declval<T>()))
    >;
```

Now it works correctly for any sequence collection with iterators.
But...

REQUIREMENTS

Choice

Which implementation of `inner_type` is better – the one that relies on `value_type` or the one that relies on iterators?

Duck-oriented programming

```
template <typename C>
concept HasValueType =
    requires { typename C::value_type; };
```

Note: This is **not** a proper concept.

Duck-oriented programming

```
template <typename C>
concept HasBeginIterator =
    requires(C c) { *std::begin(c); };
```

Note: This is **not** a proper concept.

Duck-oriented programming

```
template <typename C>
    requires HasValueType<C>
auto sum(C c) {
    …
}

template <typename C>
    requires not HasValueType<C>
             and HasBeginIterator<C>
auto sum(C c) {
    …
}
```

Duck-oriented programming

```
template <typename T>
auto sum(const T& collection)
{
    if constexpr (HasValueType<T>) {
        …
    } else if constexpr (HasBeginIterator<T>) {
        …
    }
}
```

Duck-oriented programming

```
template <typename T>
auto inner_type_helper(T&& t)
{
    if constexpr (HasValueType<T>) {
        return std::declval<typename T::value_type>();

    } else if constexpr (HasBeginIterator<T>) {
        return *std::begin(t);

    } else {
        static_assert(
            fail<T>,
            ”Unable to deduce the value type”);
    }
}
```

Duck-oriented programming

```
template <typename T>
using inner_type =
    decltype(inner_type_helper(std::declval<T>()));
```

```
int[] xs { 1, 2, 3 };

print_types<
    inner_type<std::vector<bool>>,      bool
    inner_type<std::vector<int>>,       int
    inner_type<decltype(xs)>            int
    >
```

Duck-oriented programming

```
template <typename Begin, typename End>
concept Range =
    std::is_copy_constructible_v<Begin>
    and requires (Begin b, End e) {
        *b;
        ++b;
        b != e;
        b == e;
    };
```

Back to the drawing board

Compilers? GCC?

INTO THE VOID

The void

```
template <typename...>
using void_t = void;
```

Discarding overloads

The result is not important.

It is important whether the result exists or not.

If the compiler fails to perform a substitution during overload resolution for templates, it just removes the offending overload (SFINAE).

Detecting nested types

```
template <typename T,                                    1
          typename = void_t<>>
struct has_value_type
    : std::false_type {};

template <typename T>                                              2
struct has_value_type<
        T,
        void_t<typename T::value_type>>
    : std::true_type {};
```

Detecting nested types

```
if constexpr (has_value_type<T>()) {
    …
} else {
    …
}
```

Adding more constraints

```
template <typename T,
          typename = void_t<>>
struct is_iterable : std::false_type {};

template <typename T>
struct is_iterable<
    T,
    void_t<decltype(*std::begin(std::declval<T>())),
           decltype(std::end(std::declval<T>())),
           …
          >
    > : std::true_type {};
```

DETECTION

The detector backend

Invented by Walter E. Brown:

```
template <typename Def, typename Void,                    1
          template<typename...> typename Op,
          typename... Args>
struct DETECTOR {
    using value_t = std::false_type;
    using type = Def;
};

template <typename Def,                                   2
          template<typename...> typename Op,
          typename... Args>
struct DETECTOR<Def, void_t<Op<Args...>>,
          Op, Args...> {
    using value_t = std::true_type;
    using type = Op<Args...>;
};
```

(exposition only meta-function)

Detection

```
template <template<typename...> typename Op,
          typename... Args>
using is_detected =
    typename DETECTOR<nonesuch, void, Op, Args...>::value_t;

template <template<typename...> typename Op,
          typename... Args>
using detected_t =
    typename DETECTOR<nonesuch, void, Op, Args...>::type;
```

Detection

```
template <typename T>
using nonmember_begin = decltype(std::begin(std::declval<T>()));


if constexpr (is_detected_v<nonmember_begin, std::string>) {
    …
}
```

Detection

```
template <typename T>
using nonmember_end = decltype(std::end(std::declval<T>()));


if constexpr (is_detected_v<nonmember_begin, std::string> &&
               is_detected_v<nonmember_end, std::string>) {
    …
}
```

Detection

```
template <typename T>
using dereference = decltype(*(std::declval<T>()));


if constexpr (is_detected_v<free_begin, std::string> &&
               is_detected_v<free_end, std::string> &&
               is_detected_v<dereference,
                      detected_t<free_begin, std::string>>) {
    …
}
```

Detection

```
template <typename T>
using increment = decltype(++(std::declval<T>()));

template <typename T>
using copy_assign = decltype(std::declval<T&>() =
                                    std::declval<const T&>());

template <typename T>
constexpr bool is_input_iterator =
                            is_detected_v<dereference, T> &&
                            is_detected_v<increment, T> &&
                            is_detected_v<copy_assign, T>;
```

Detection

```
template <typename T, typename U = T>
using eq_compare = decltype(
        (std::declval<T>() == std::declval<U>()) &&
        (std::declval<T>() != std::declval<U>()));

template <typename R>
constexpr bool is_input_range =
    is_input_iterator<detected_t<free_begin, R>> &&

    is_detected_v<eq_compare,
            detected_t<free_begin, R>,
            detected_t<free_end, R>
        >;
```

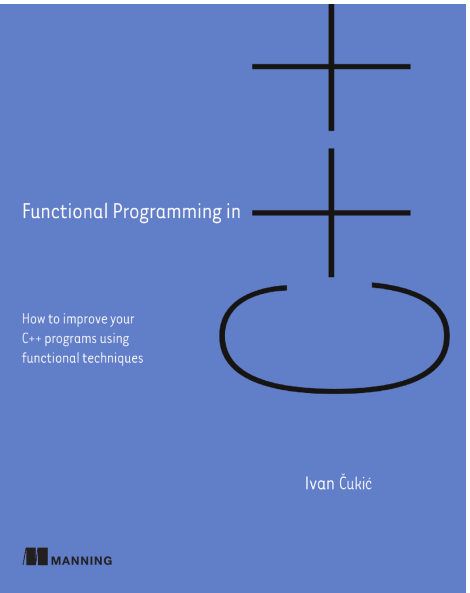
Detection

```
template <typename Def, template<typename...>
          typename Op,  typename... Args>
using detected_or =
          DETECTOR<Default, void, Op, Args...>;

template <typename Expected,
          template <typename...> typename Op,
          typename... Args>
using is_detected_exact =
          is_same<Expected, detected_t<Op, Args...>>;

template <typename To,
          template<typename...> typename Op,
          typename... Args>
using is_detected_convertible =
          is_convertible<detected_t<Op, Args...>, To>;
```

Questions

Kudos (in chronological order):

Friends at **KDE**
Saša Malkov and **Zoltan Porkolab**



MANNING PUBLICATIONS
Functional Programming in C++
cukic.co/to/fp-in-cpp

Discount code **ctwmeetingcpp18**
(40% off all books and videos)