Template Metaprogramming and <type_traits>

Joe Jevnik

April 4, 2018

Boston C++

Outline

1. Template types
2. Template functions
3. Template variables
4. Template `using`
5. Parameter kinds
6. Advanced template usage
7. Computation model of templates
8. Overview of `<type_traits>`
9. Extras

Template Types

Template Types

```cpp
class vec2d_int {
private:
    const int m_x;
    const int m_y;

public:
    vec2d_int(int, int);
    int x() const;
    int y() const;
    int dot(const vec2d_int&) const;
};
```

Template Types

```
class vec2d_float {
private:
    const float m_x;
    const float m_y;

public:
    vec2d_float(float, float);
    float x() const;
    float y() const;
    float dot(const vec2d_float&) const;
};
```

Template Types

```
class vec2d_double {
private:
    const double m_x;
    const double m_y;

public:
    vec2d_double(double, double);
    double x() const;
    double y() const;
    double dot(const vec2d_double&) const;
};
```

Template Types

```
int vec2d_int::dot() {
    return m_x * other.m_x + m_y * other.m_y;
}
```

Template Types

```
int vec2d_int::dot() {
    return m_x * other.m_x + m_y * other.m_y;
}

float vec2d_float::dot() {
    return m_x * other.m_x + m_y * other.m_y;
}
```

Template Types

```
int vec2d_int::dot() {
    return m_x * other.m_x + m_y * other.m_y;
}

float vec2d_float::dot() {
    return m_x * other.m_x + m_y * other.m_y;
}

double vec2d_double::dot() {
    return m_x * other.m_x + m_y * other.m_y;
}
```

Template Types

```
template<typename T>
class vec2d {
private:
    const T m_x;
    const T m_y;

public:
    vec2d(const T&, const T&);
    const T& x() const;
    const T& y() const;
    T dot(const vec2d&) const;
};
```

Template Types

```
template<typename T>
T vec2d<T>::dot(const vec2d<T>& other) const {
    return m_x * other.m_x + m_y * other.m_y;
}
```

Template Types

```
vec2d<int> int_vector(2, 4);
vec2d<float> float_vector(2.5, 4.5);
vec2d<double> double_vector(2.5, 4.5);
```

Template Types

```
struct not_a_numeric_type {};

vec2d<not_a_numeric_type> v(not_a_numeric_type{},
                            not_a_numeric_type{});
not_a_numeric_type dotted = v.dot(v);
```

Template Types

```
scratch.cc: In instantiation of
    T vec2d<T>::dot(const vec2d<T>&) const
    [with T = not_a_numeric_type]:
scratch.cc:31:40:   required from here
scratch.cc:21:20: error: no match for operator*
    (operand types are
     const not_a_numeric_type and
     const not_a_numeric_type)
        return m_x * other.m_x + m_y * other.m_y;
               ~~~~~~~~~~~~~~~
```

Template Types

```
class z5 {
private:
    std::uint8_t m_value;

public:
    z5(int);
    z5 operator+(const z5&) const;
    z5 operator*(const z5&) const;
};

vec2d<z5> v(1, 2);
z5 dotted = v.dot(v);
```

Value Parameters

```
template<typename T>
class vec3d {
private:
    const T m_x;
    const T m_y;
    const T m_z;

public:
    vec3d(const T&, const T&, const T&);
    const T& x() const;
    const T& y() const;
    const T& z() const;
    vec3d dot(const vec3d& other) const;
};
```

Value Parameters

```
template<typename T>
class vec4d {
private:
    const T m_axis_0;
    const T m_axis_1;
    const T m_axis_2;
    const T m_axis_3;

public:
    vec4d(const T&, const T&, const T&, const T&);
    const T& axis_0() const;
    // ...
    vec4d dot(const vec4d& other) const;
};
```

Value Parameters

```cpp
#include <array>

template<typename T, std::size_t size>
class vecnd {
private:
    const std::array<T, size> m_data;

public:
    vecnd(const std::array<T, size>&>);
    const T& operator[](std::size_t) const;
    T dot(const vecnd&) const;
};
```

Value Parameters

```
template<typename T, std::size_t size>
T vecnd<T, size>::dot(const vecnd<T, size>& other) const {
    T sum = 0;
    for (std::size_t ix = 0; ix < size; ++ix) {
        sum += (*this)[ix] * other[ix];
    }
    return sum;
}
```

Value Parameters

```
vecnd<int, 4> v({1, 2, 3, 4});
```

Template Functions

Member Functions

```
vec2d<int> v(1, 2);
vec2d<float> u(1.5, 2.5);
int dotted = v.dot(u);
```

Member Functions

```
scratch.cc:84:26: error: no matching function for
    call to vec2d<int>::dot(vec2d<float>&)
     int dotted = v.dot(u);
                          ^
scratch.cc:20:10: note: candidate:
    T vec2d<T>::dot(const vec2d<T>&) const
    [with T = int]
     T dot(const vec2d& other) const {
       ^~~
scratch.cc:20:10: note:
    no known conversion for argument 1 from
    vec2d<float> to const vec2d<int>&
```

Member Functions

```cpp
template<typename T>
class vec2d {
private:
    const T m_x;
    const T m_y;

public:
    // ...

    template<typename U>
    T dot(const vec2d<U>& other) const {
        return m_x * other.m_x + m_y * other.m_y;
    }
};
```

Member Functions

```
scratch.cc: In instantiation of
    T vec2d<T>::dot(const vec2d<U>&) const
    [with U = float; T = int]:
scratch.cc:78:26:   required from here
scratch.cc:16:28: error: const float vec2d<float>::m_x
    is private within this context
         return m_x * other.m_x + m_y * other.m_y;
                      ~~~~~~~~~
scratch.cc:6:13: note: declared private here
     const T m_x;
             ^~~
```

Member Functions

```
template<typename U>
T dot(const vec2d<U>& other) const {
    return m_x * other.x() + m_y * other.y();
}
```

```
vec2d<int> v(1, 2);
vec2d<float> u(1.5, 2.5);
int dotted = v.dot(u);  // 6
float manual = 1 * 1.5 + 2 * 2.5;  // 6.5
```

```
template<typename U>
auto dot(const vec2d<U>& other) const {
    return m_x * other.x() + m_y * other.y();
}
```

```
vec2d<int> v(1, 2);
vec2d<float> u(1.5, 2.5);
auto dotted = v.dot(u);  // 6.5 :: float
```

Free Functions

```cpp
template<typename T>
std::string f(T) {
    return "template function";
}

std::string f(long) {
    return "non-template function";
}

f(0);  // template function
f(0L);  // non-template function
```

Template Variables

Template Variables

```cpp
template<int value>
int half = value / 2;

int a = half<4>;  // 2
int b = half<5>;  // 2 (integer division)
```

Template Variables

```
template<typename Key, typename Value>
std::map<Key, Value> cache;

cache<int, float>[0] = 1.5;

(std::addressof(cache<int, float>) ==
 std::addressof(cache<int, float>))  // true
```

Template `using`

Template `using`

```
template<typename T>
using const_ref = const T&;

// const_ref<int> -> const int&
// const_ref<const int> -> const int&
// const_ref<const int&> -> const int&
```

Parameter Kinds

Parameter Kinds

- `typename` or `class`

Parameter Kinds

- `typename` or `class`
- fundamental integral types (`int`, `unsigned int`, etc.)

Parameter Kinds

- `typename` or `class`
- fundamental integral types (`int`, `unsigned int`, etc.)
- pointer types (static storage)

Parameter Kinds

- `typename` or `class`
- fundamental integral types (`int`, `unsigned int`, etc.)
- pointer types (static storage)
- function values

Parameter Kinds

- `typename` or `class`
- fundamental integral types (`int`, `unsigned int`, etc.)
- pointer types (static storage)
- function values
- `auto`

Parameter Kinds

- `typename` or `class`
- fundamental integral types (`int`, `unsigned int`, etc.)
- pointer types (static storage)
- function values
- `auto`
- `template*`

Advanced Template Usage

Explicit Template Specialization

```cpp
template<typename T>
struct s {
    static std::string f() {
        return "T = ?";
    }
};

template<>
struct s<int> {
    static std::string f() {
        return "T = int";
    }
};
```

Explicit Template Specialization

```
s<float>::f();  // T = ?
s<void>::f();   // T = ?
s<int>::f();    // T = int
```

Incomplete Types

```
template<typename T>
struct s;

template<>
struct s<int> {
    static std::string f() { return "T = int"; }
};

template<>
struct s<float> {
    static std::string f() { return "T = float"; }
};
```

Incomplete Types

```
s<float>::f(); // T = float
s<int>::f();   // T = int
s<void>::f();  // error: incomplete type s<void>
               // used in nested name specifier
```

Incomplete Types

```
template<typename T>
struct s;

template<>
struct s<int> {
    static std::string f() { return "T = int"; }
};

template<>
struct s<float> {
    static int g(int a) { return a + 1; }
};
```

Partial Template Specialization

```
template<typename A, typename B>
struct s {
    static std::string f() { return "A = ? and B = ?"; }
};

template<typename A>
struct s<A, int> {
    static std::string f() { return "A = ? and B = int"; }
};

template<typename B>
struct s<int, B> {
    static std::string f() { return "A = int and B = ?"; }
};
```

Partial Template Specialization

```
s<float, float>::f();  // A = ? and B = ?
s<float, int>::f();    // A = ? and B = int
s<int, float>::f();    // A = int and B = ?
```

Partial Template Specialization

```
s<int, int>::f();
```

Partial Template Specialization

```
scratch.cc:120:29: error: ambiguous template
    instantiation for struct s<int, int>
     std::cout << s<int, int>::f() << '\n';
                             ^~
scratch.cc:98:8: note: candidates are:
    template<class A> struct s<A, int>
    [with A = int]
 struct s<A, int> {
        ^~~~~~~~~
scratch.cc:105:8: note:
    template<class B> struct s<int, B>
    [with B = int]
 struct s<int, B> {
        ^~~~~~~~~
```

Partial Template Specialization

```
template<typename T>
struct s;

template<typename ElementType>
struct s<std::vector<ElementType>> {
    using type = ElementType;
};

using a = std::vector<int>;
using element_type = typename s<a>::type;  // int
```

Variadic Templates

```
using namespace std::string_literals;

print_all("I typed", '<', '<', 100, " times today"s.);
print_all("Never ", L"again.");
```

Variadic Templates

```
void print_all();

template<typename A>
void print_all(const A&);

template<typename A, typename B>
void print_all(const A&, const B&);

template<typename A, typename B, typename C>
void print_all(const A&, const B&, const C&);

// ...
```

Variadic Templates

```
void print_all();

template<typename T, typename... Ts>
void print_all(const T& first, const Ts&... rest);
```

Variadic Templates

```
void print_all() {
}

template<typename T, typename... Ts>
void print_all(const T& first, const Ts&... rest) {
    std::cout << first;
    print_all(rest...);
}
```

Parameter Packs

```
template<typename... Ts>
void f(Ts... args) {

    g(args...);  // g(args[0], args[1], ...)










}
```

Parameter Packs

```
template<typename... Ts>
void f(Ts... args) {

    g(args...);  // g(args[0], args[1], ...)
    g((1 + args)...);  // g(1 + args[0], 1 + args[1], ...)









}
```

Parameter Packs

```
template<typename... Ts>
void f(Ts... args) {

    g(args...);  // g(args[0], args[1], ...)
    g((1 + args)...);  // g(1 + args[0], 1 + args[1], ...)
    g((args + 1)...);  // g(args[0] + 1, args[1] + 1, ...)







}
```

Parameter Packs

```
template<typename... Ts>
void f(Ts... args) {

    g(args...);  // g(args[0], args[1], ...)
    g((1 + args)...);  // g(1 + args[0], 1 + args[1], ...)
    g((args + 1)...);  // g(args[0] + 1, args[1] + 1, ...)
    g(h(args)...);  // g(h(args[0]), h(args[1]), ...)




}
```

Parameter Packs

```
template<typename... Ts>
void f(Ts... args) {

    g(args...);  // g(args[0], args[1], ...)
    g((1 + args)...);  // g(1 + args[0], 1 + args[1], ...)
    g((args + 1)...);  // g(args[0] + 1, args[1] + 1, ...)
    g(h(args)...);  // g(h(args[0]), h(args[1]), ...)
    std::tuple<Ts...> t(args...);




}
```

Parameter Packs

```
template<typename... Ts>
void f(Ts... args) {

    g(args...);  // g(args[0], args[1], ...)
    g((1 + args)...);  // g(1 + args[0], 1 + args[1], ...)
    g((args + 1)...);  // g(args[0] + 1, args[1] + 1, ...)
    g(h(args)...);  // g(h(args[0]), h(args[1]), ...)
    std::tuple<Ts...> t(args...);
    using type = std::tuple<const Ts&...>;
    // type = std::tuple<const Ts[0]&, const Ts[1]&, ...>


}
```

Parameter Packs

```
template<typename... Ts>
void f(Ts... args) {

    g(args...);  // g(args[0], args[1], ...)
    g((1 + args)...);  // g(1 + args[0], 1 + args[1], ...)
    g((args + 1)...);  // g(args[0] + 1, args[1] + 1, ...)
    g(h(args)...);  // g(h(args[0]), h(args[1]), ...)
    std::tuple<Ts...> t(args...);
    using type = std::tuple<const Ts&...>;
    // type = std::tuple<const Ts[0]&, const Ts[1]&, ...>
    using type = std::tuple<const_ref<Ts>...>;

}
```

Parameter Packs

```
template<typename... Ts>
void f(Ts... args) {

    auto right_sum = (... + args);
    // args[0] + (args[1] + (args[2] + ...))









}
```

Parameter Packs

```
template<typename... Ts>
void f(Ts... args) {

    auto right_sum = (... + args);
    // args[0] + (args[1] + (args[2] + ...))
    auto left_sum = (args + ...);
    // ((args[0] + args[1]) + args[2] + ...)




}
```

Parameter Packs

```
template<typename... Ts>
void f(Ts... args) {

    auto right_sum = (... + args);
    // args[0] + (args[1] + (args[2] + ...))
    auto left_sum = (args + ...);
    // ((args[0] + args[1]) + args[2] + ...)
    auto right_sum_with_init = (1 + ... + args);
    // 1 + (args[0] + (args[1] + (args[2] + ...)))



}
```

Parameter Packs

```
template<typename... Ts>
void f(Ts... args) {

    auto right_sum = (... + args);
    // args[0] + (args[1] + (args[2] + ...))
    auto left_sum = (args + ...);
    // ((args[0] + args[1]) + args[2] + ...)
    auto right_sum_with_init = (1 + ... + args);
    // 1 + (args[0] + (args[1] + (args[2] + ...)))
    auto left_sum_with_init = (args + ... + 1);
    // ((args[0] + args[1]) + args[2] + ... + 1)

}
```

Optional Definitions

```cpp
template<typename T, bool = (sizeof(T) > 8)>
struct container {
    static std::string f() { return "large T"; }
};

template<typename T>
struct container<T, false> {
    static std::string f() { return "small T"; }
};

container<int>::f();  // "small T"
container<std::array<int, 10>>::f();  // "large T"
```

`decltype` and `std::declval`

`decltype`

keyword that returns the type of an expression

`decltype` **and** `std::declval`

`decltype`

- keyword that returns the type of an expression
- does not execute argument

decltype and std::declval

decltype

keyword that returns the type of an expression

does not execute argument

```
int a;
using type_of_a = decltype(a);  // int

struct s { /* ... */ };
using result = decltype(s.f(a));  // result of s.f(a)
```

`decltype` **and** `std::declval`

`std::declval`

function (not keyword) that returns an instance of an arbitrary type `std::declval`

- function (not keyword) that returns an instance of an arbitrary type
- may only be used in a `decltype`, cannot be executed

`decltype` and `std::declval`

`std::declval`

- function (not keyword) that returns an instance of an arbitrary type
- may only be used in a `decltype`, cannot be executed

```
struct s;
using result = decltype(std::declval<s>() + 1);
// result of operator+(s, int)
```

Capability Based Dispatching

```
template<typename T, typename = void>
struct s {
    static auto f(const T&) { return 0; }
};

template<typename T>
struct s<T, decltype(std::declval<const T>().f(), void())>
    static auto f(const T& ob) { return ob.f(); }
};
```

Capability Based Dispatching

```
struct value {
    int f() const { return 1; }
};

s<const char*>::f("no f() method");  // 0
s<value>::f(value{});  // 1
```

Computation Model

Overview

- functional paradigm

Overview

- functional paradigm
- pure

Overview

- functional paradigm
- pure
- evaluates to a C++ program without templates

C++ Templates vs. Haskell

```
template<typename T>
struct element_type;
```

C++ Templates vs. Haskell

```haskell
import Data.Map

data CxxType = CxxInt
             | CxxFloat
             | CxxDouble
             | CxxVector CxxType
             | CxxCompleteStruct (Map String Member)
             | CxxIncompleteStruct

data Member = MemberType CxxType
            | MemberFunction [CxxType]
```

C++ Templates vs. Haskell

```haskell
data ElementTypeTemplate = ElementTypeTemplate CxxType

element_type instantiation =
    case instantiation of
        ElementTypeTemplate _ -> CxxIncompleteStruct
```

C++ Templates vs. Haskell

```
template<typename T>
struct element_type<std::vector<T>> {
    using type = T;
};
```

C++ Templates vs. Haskell

```haskell
data ElementTypeTemplate = ElementTypeTemplate CxxType

element_type instantiation =
    case instantiation of
        ElementTypeTemplate (CxxVector t) ->
            CxxCompleteStruct $ singleton
                                          "type"
                                          (MemberType t)
        ElementTypeTemplate _ -> CxxIncompleteStruct
```

Encoding Computation in Templates

```cpp
template<typename A, typename B>
struct eq {
    static constexpr bool value = false;
};

template<typename A>
struct eq<A, A> {
    static constexpr bool value = true;
};
```

Encoding Computation in Templates

```
template<auto head, typename Tail>
struct cond;

struct nil;

using list = cond<1, cond<2, cond<3, nil>>>;
```

Encoding Computation in Templates

```
template<template<auto> typename F, typename List>
struct map {
    using type = nil;
};

template<template<auto> typename F,
         auto head,
         typename Tail>
struct map<F, cons<head, Tail>> {
    using type = cons<F<head>::value,
                      typename map<F, Tail>::type>;
};
```

Encoding Computation in Templates

```
template<auto v>
struct f {
    static constexpr auto value = v + 1;
};

static_assert(eq<typename map<f, list>::type,
                 cons<2, cons<3, cons<4, nil>>>>::value);
```

Encoding Computation in Templates

```
template<template<auto> typename F, typename List>
using eager_map = typename map<F, List>::type;

static_assert(eq<typename map<f, list>::type,
                 eager_map<f, list>>::value);
```

Memoization

```
template<std::size_t n>
constexpr std::size_t fib = fib<n - 1> + fib<n - 2>;

template<>
constexpr std::size_t fib<0> = 1;

template<>
constexpr std::size_t fib<1> = 1;
```

Memoization

```
fib(n=0):  0.29 +- 0.008 seconds
fib(n=1):  0.30 +- 0.003 seconds
fib(n=2):  0.29 +- 0.002 seconds
...
fib(n=47): 0.30 +- 0.021 seconds
fib(n=48): 0.30 +- 0.013 seconds
fib(n=49): 0.29 +- 0.014 seconds
```

<type_traits>

Integral Constant

```
template<typename T, T v>
struct integral_constant {
    using value_type = T;
    static constexpr T value = v;
};

using false_type = integral_constant<bool, false>;
using true_type = integral_constant<bool, true>;
```

Type Categories

- `is_void`
- `is_null_pointer`
- `is_integral`
- `is_floating_point`
- `is_array`
- `is_enum`
- `is_union`
- `is_class`
- `is_function`
- `is_pointer`
- `is_lvalue_reference`
- `is_rvalue_reference`
- `is_member_object_pointer`
- `is_member_function_pointer`

Type Categories

```cpp
template<typename T>
struct is_array : public false_type {};

template<typename T>
struct is_array<T[]> : public true_type {};

template<typename T, std::size_t n>
struct is_array<T[n]> : public true_type {};

template<typename T>
struct is_pointer : public false_type {};

template<typename T>
struct is_pointer<T*> : public true_type {};
```

Composite Types

- is_fundamental
- is_arithmetic
- is_scalar
- is_object
- is_compound
- is_reference
- is_member_pointer

Type Properties

- `is_const`
- `is_volatile`
- `is_trivial`
- `is_trivially_copyable`
- `is_standard_layout`
- `is_pod`
- `is_literal_type`
- `has_unique_object_representations`
- `is_empty`
- `is_polymorphic`
- `is_abstract`
- `is_final`
- `is_aggregate`
- ...

Type Properties

- `has_unique_object_representations`: No padding in a struct.

Type Properties

- `has_unique_object_representations`: No padding in a struct.
- `is_empty`: No non-static members.

Type Properties

- `has_unique_object_representations`: No padding in a struct.
- `is_empty`: No non-static members.
- `is_polymorphic`: Has at least one virtual.

Relationships

- `is_same`
- `is_base_of`
- `is_convertible`

Manipulating Types

- remove_cv
- remove_const
- remove_volatile
- add_cv
- add_const
- add_volatile
- remove_reference
- add_lvalue_reference
- add_rvalue_reference
- add_pointer
- remove_pointer

Manipulating Types

```
template<typename T>
struct remove_const {
    using type = T;
};

template<typename T>
struct remove_const<const T> {
    using type = T;
};
```

Manipulating Types

```
template<typename T>
struct remove_reference {
    using type = T;
};

template<typename T>
struct remove_reference<T&> {
    using type = T;
};

template<typename T>
struct remove_reference<T&&> {
    using type = T;
};
```

Shorthand Helpers

```
template<typename T>
constexpr auto name_v = name<T>::value;

template<typename T>
using name_t = typename name<T>::type;
```

Conditionals

```
(predicate) ? if_true : if_false;
```

Conditionals

```cpp
template<bool p, typename IfTrue, typename IfFalse>
struct conditional {
    using type = IfFalse;
};

template<typename IfTrue, typename IfFalse>
struct conditional<true, IfTrue, IfFalse> {
    using type = IfTrue;
};
```

Enable If

```cpp
template<bool p, typename T = void>
struct enable_if;

template<typename T>
struct enable_if<true, T> {
    using type = T;
};
```

Enable If

```
template<typename T,
         typename = enable_if_t<!is_reference_v<T>>>
struct optional {
    // ...
};
```

void_t

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

void_t

```cpp
template<typename T, typename = void>
struct s {
    static auto f(const T&) { return 0; }
};

template<typename T>
struct s<T, void_t<decltype(std::declval<const T>().f())>> {
    static auto f(const T& ob) { return ob.f(); }
};
```

Supported Operations

- `is_move_constructible`, `is_trivially_move_constructible`, `is_nothrow_move_constructible`
- `is_move_assignable`, `is_trivially_move_assignable`, `is_nothrow_move_assignable`
- `is_destructible`, `is_trivially_destructible`, `is_nothrow_destructible`
- `has_virtual_destructor`
- ...

Supported Operations

```
template<typename T>
std::conditional_t<
    !std::is_nothrow_move_constructible_v<T> &&
    std::is_copy_constructible_v<T>,
    const T&,
    T&&>
move_if_noexcept(T& ob) noexcept {
    return ob;
}
```

Supported Operations

```
template<typename T>
void vector<T>::push_back(const T& value) {
    if (m_size == m_capacity) {
        std::size_t new_capacity = m_capacity * 2;
        auto buffer = std::make_unique<T[]>(new_capacity);

        std::size_t ix = 0;
        for (T& element : m_data) {
            buffer[ix++] = move_if_noexcept(element);
        }

        m_data = std::move(buffer);
        m_capacity = new_capacity;
    }
```

Extras

```
template<bool p, typename V>
struct case_;

template<typename... Cases>
struct cond;
```

cond **and** case

```
template<typename V, typename... Cases>
struct cond<case_<true, V>, Cases...> {
    using type = V;
};

template<typename V, typename... Cases>
struct cond<case_<false, V>, Cases...> {
    using type = typename cond<Cases...>::type;
};
```

cond and case

```cpp
template<typename A, typename B>
struct align_dimensions {
private:
    struct base_case { using type = std::pair<A, B>; };

public:
    using type = typename utils::cond<
        utils::case_<(wired::ndim<A> < wired::ndim<B>),
                     align_dimensions<array<A>, B>>,
        utils::case_<(wired::ndim<A> > wired::ndim<B>),
                     align_dimensions<A, array<B>>>,
        utils::case_<true, base_case>>::type;
};
```

Debugging

```
template<typename>
struct print_type;

template<auto>
struct print_value;
```

Debugging

```
print_type<std::remove_reference_t<int&>>::p;
// error: invalid use of incomplete type
// struct print_type<int>
```

Debugging

```
print_value<fib<35>>::p;
// error: invalid use of incomplete type
// struct print_value<14930352>
```

Thank You

Questions?