# Type loopholes in C++: Убербаг уровня стандарта


Квятковский Антон

Software Engineer at Dino Systems

# Основные определения:
# Инстанциация, специализация, подстановка

```
template <typename T>
struct MetaFoo
{
    T val{};
};
```

# Основные определения: Инстанциация, специализация, подстановка

```
template <typename T>
struct MetaFoo
{
    T val{};
};


void foo()
{
    MetaFoo<int> foo; //затребовали специализацию
}
```

# Основные определения: Инстанциация, специализация, подстановка

```
template <typename T> //запустился процесс инстанциииации
struct MetaFoo        //в течении которого компилятор выполнил подстановки
{
    T val{};
};


void foo()
{
    MetaFoo<int> foo;
}
```

# Основные определения: Инстанциация, специализация, подстановка

```
template <typename T>
struct MetaFoo
{
    T val{};
};
template<>                  // ← сгенерированная специализация
struct MetaFoo<int>
{
    int val{};
};
void foo()
{
    MetaFoo<int> foo;
}
```

# Метафункции в C++

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

```
template <typename T>
struct MetaFoo
{
    static constexpr bool value = expression<T>;
};
```

```
static_assert(std::is_same<int, int>::value);
```

# Функциональная парадигма в шаблонах

«Чистые функции - … не имеют … памяти (они зависят только от своих параметров и возвращают только свой результат).»

```
static_assert(std::is_same<int, int>::value); // всегда возвращает один и тот же результат
```

# Friend members

```
struct Foo {
    Foo(){}
    Foo(int){}
    friend void bar(Foo){}
};

void check()
{
    //bar(5); ← doesn't compile

    Foo foo = 5;
    bar(foo); //OK, found through ADL
}
```

# Friend members of template class

```
struct adl_tag{};

constexpr int foo(adl_tag);

template<typename T>
struct Foo {
    friend constexpr int foo(adl_tag) { return 0; }
};

constexpr void bar()
{
    constexpr auto result = foo(adl_tag{}); // ← doesn’t compile
}
```

# Friend members of template class «Инжектим» имя во внешний скоуп

```
struct adl_tag{};

constexpr int foo(adl_tag);

template<typename T>
struct Foo {
    friend constexpr int foo(adl_tag) { return 0; }
};

constexpr void bar()
{
    Foo<int> obj;
    constexpr auto result = foo(adl_tag{});
}
```

# Friend members of template class

```
struct adl_tag{};

constexpr int foo(adl_tag);

template<typename T>
struct Foo {
    friend constexpr int foo(adl_tag) { return 0; }
};

constexpr void bar()
{
    Foo<int> obj;
    constexpr auto result = foo(adl_tag{}); // ← OK, compiles
}
```

# Заготовка для инъекции

```
template<typename>
struct DefInserter {
    friend constexpr auto get(DefInserter);
};

template <typename Key, typename Value>
struct DeclInserter
{
    friend constexpr auto get(Key)
    {
        return Value{};
    }
};
```

# Инжектим имя

```
template<typename>
struct DefInserter {
    friend constexpr auto get(DefInserter);
};
template <typename Key, typename Value>
struct DeclInserter
{
    friend constexpr auto get(Key)
    { return Value{}; }
};

void foo()
{
    sizeof(DeclInserter<DefInserter<int>, double>);
    static_assert(std::is_same_v<decltype(get(DefInserter<int>{})), double > );
}
```

# Проверяем себя

```
template<typename>
struct DefInserter {
    friend constexpr auto get(DefInserter);
};
template <typename Key, typename Value>
struct DeclInserter
{
    friend constexpr auto get(Key)
    { return Value{}; }
};

void foo()
{
    sizeof(DeclInserter<DefInserter<int>, double>);
    static_assert(std::is_same_v<decltype(get(DefInserter<int>{})), double > );
}
```

# Использование

# Кратко про magic_get
# Считаем кол-во полей агрегата

```
struct Foo
{
    int a;
    double b;
    char c;
};

Foo{ 1, 1.2, 'z' };
```

```
template<typename T>
struct Caster
{
    template<typename U>
    operator U();
};
```

# Кратко про magic_get
# Считаем кол-во полей агрегата

```
template <typename T, std::size_t I0, std::size_t ... Tail>
constexpr auto get_size(std::index_sequence<I0, Tail...>) ->
  decltype (T{ Caster<T, I0>{}, Caster<T, Tail>{}... }, std::size_t{ 0 })
{
    return sizeof... (Tail) + 1;
}

template<typename T, std::size_t ... Indexes>
constexpr std::size_t get_size(std::index_sequence<Indexes...>)
{
    return get_size<T>(std::make_index_sequence<sizeof... (Indexes) - 1>{});
}
```

# Кратко про magic_get
# Считаем кол-во полей агрегата

```
template <typename T, std::size_t I0, std::size_t ... Tail>
constexpr auto get_size (std::index_sequence<I0, Tail...>) ->
  decltype (T{ Caster<T, I0>{}, Caster<T, Tail>{}... }, std::size_t{ 0 })
{
    return sizeof... (Tail) + 1;
}

template<typename T, std::size_t ... Indexes>
constexpr std::size_t get_size (std::index_sequence<Indexes...>)
{
    return get_size<T>(std::make_index_sequence<sizeof... (Indexes) - 1>{});
}
```

# Кратко про magic_get
# Считаем кол-во полей агрегата

```
template <typename T, std::size_t I0, std::size_t ... Tail>
constexpr auto get_size(std::index_sequence<I0, Tail...>) ->
  decltype (T{ Caster<T, I0>{}, Caster<T, Tail>{}... }, std::size_t{ 0 })
{
    return sizeof... (Tail) + 1;
}

template<typename T, std::size_t ... Indexes>
constexpr std::size_t get_size(std::index_sequence<Indexes...>)
{
    return get_size<T>(std::make_index_sequence<sizeof... (Indexes) - 1>{});
}
```

# Конвертилка

```
template<typename T, std::size_t Index>
struct Caster
{
    template<
            typename U
            , int = sizeof(DeclInserter<DefInserter<T, Index>, U>)
            >
    operator U();
};
```

# Добавляем пару ключ-значение

```
template<typename T, std::size_t Index>
struct Caster
{
    template<
            typename U
            , int = sizeof (DeclInserter<DefInserter<T, Index>, U>)
            >
    operator U();
};
```

```
template<typename>
struct DefInserter {
    friend constexpr auto get(DefInserter);
};

template <typename Key, typename Value>
struct DeclInserter
{
    friend constexpr auto get(Key)
    {
        return Value{};
    }
};
```


T – тип агрегата
Index – порядковый номер поля
U – тип поля

# Получаем список полей

```
template<typename T, std::size_t ... Indexes>
constexpr auto get_list(std::index_sequence<Indexes...>)
{
    return std::tuple<decltype(get(DefInserter<Foo, Indexes>{}))... > {};
}

template <typename T>
struct GetListOfFields
{
    using type = decltype(get_list<T>(std::make_index_sequence<GetFieldCount<T>::value>{}));
};
```

# Получаем список полей

```
template<typename T, std::size_t ... Indexes>
constexpr auto get_list(std::index_sequence<Indexes...>)
{
    return std::tuple<decltype (get(DefInserter<Foo, Indexes>{}))... > {};
}

template <typename T>
struct GetListOfFields
{
    using type = decltype(get_list<T>(std::make_index_sequence<GetFieldCount<T>::value>{}));
};
```

# Constexpr счётчик

```
template <std::size_t>
struct decl
{
    friend constexpr int flag(decl);
};

template <typename Key>
struct def
{
    friend constexpr int flag(Key) { return 0;
}
};
```

# Constexpr счётчик

```
template < std::size_t param, int = sizeof(def<decl<param>>) >
constexpr int impl_next(...)
{
    return param;
}

template <std::size_t param, int = flag(decl<param>{}), auto = unique() >
constexpr int impl_next(int)
{
    return impl_next<param + 1>(int{});
}
```

# Constexpr счётчик

```
template < std::size_t param, int = sizeof(def<decl<param>>) >
constexpr int impl_next(...)
{
    return param;
}

template <std::size_t param, int = flag(decl<param>{}), auto = unique() >
constexpr int impl_next(int)
{
    return impl_next<param + 1>(int{});
}
```

```
int = flag(decl<param>{})  ⇔  “работай, если такая функция есть”
```

# Constexpr счётчик

```
template <auto = unique()>
constexpr int next()
{
    return impl_next<0>(int{});
}

int foo()
{
    static_assert(next() == 1);
    static_assert(next() == 2);
    static_assert(next() == 3);
    static_assert(next() == 4);
}
```

# Немного магии (а куда без неё)

```
template<auto fn = []() {} >
constexpr auto unique() {
    return fn;
}
```

C++ 20 allows lambda as non-type template parameter

```
template <auto = unique()>
constexpr int next()
{
    return impl_next<0>(int{});
}
```

# Применение счётчика CallbackConnector

https://github.com/Toxa-man/CallbackConnector

```
void c_api_func_with_callback(void(*cb)(int)) { cb(5); }

struct Foo
{
    void some_member(int) { std::cout << "some_member " << std::endl; }
};

void foo()
{
    Foo obj;
    using namespace cbc;
    auto func_ptr = obtain_connector(&obj, &Foo::some_member);
    c_api_func_with_callback(func_ptr);
}
```

# Применение счётчика CallbackConnector

```
struct Foo
{
    void some_member(int) { std::cout << "some_member " << std::endl; }
    void another_member_with_same_signature(int) { std::cout << "another_member"; }
};

void foo()
{
    Foo obj;
    using namespace cbc;
    auto func_ptr = obtain_connector(&obj, &Foo::some_member);
    c_api_func_with_callback(func_ptr);

    auto new_func_ptr = obtain_connector<1>(&obj, &Foo::another_member_with_same_signature);
    c_api_func_with_callback(new_func_ptr);
}
```

# Вычисляем тип аргумента конструктора

```
struct Param{};

struct Foo
{
    Foo() = default;
    Foo(Param);
};


template <typename T>
struct get_type
{
    using type = ???;
};

static_assert(std::is_same_v<get_type<Foo>::type, Param>);
```

# Вычисляем тип аргумента конструктора

```
template<typename>
struct DefInserter {
    friend constexpr auto get(DefInserter);
};

template <typename Key, typename Value>
struct DeclInserter
{
    friend constexpr auto get(Key)
    {
        return Value{};
    }
};
```

# Вычисляем тип аргумента конструктора

```
template <typename T>
struct ubic
{
    template <typename Arg
            , typename = std::enable_if_t<!std::is_same_v<Arg, std::remove_cv_t<T>>>
            , int = sizeof(DeclInserter<DefInserter<T>, Arg>)
             >
    operator Arg();
};

template <typename T,
           int = sizeof(T{ ubic<T>{} })
          >
struct get_type
{
    using type = decltype(get(DefInserter<T>{}));
};
```

# Вычисляем тип аргумента конструктора

```
template <typename T>
struct ubic
{
    template <typename Arg
            , typename = std::enable_if_t<!std::is_same_v<Arg, std::remove_cv_t<T>>>
            , int = sizeof(DeclInserter<DefInserter<T>, Arg>)
             >
    operator Arg();
};

template <typename T,
          int = sizeof(T{ ubic<T>{} })
         >
struct get_type
{
    using type = decltype(get(DefInserter<T>{}));
};
```

# Вычисляем тип аргумента конструктора

```
template <typename T>
struct ubic
{
    template <typename Arg
            , typename = std::enable_if_t<!std::is_same_v<Arg, std::remove_cv_t<T>>>
            , int = sizeof(DeclInserter<DefInserter<T>, Arg>)
             >
    operator Arg();
};

template <typename T,
          int = sizeof(T{ ubic<T>{} })
         >
struct get_type
{
    using type = decltype(get(DefInserter<T>{}));
};
```

# Компилируем…

MSVC:

```
static_assert(std::is_same_v<get_type<Foo>::type, Param>);
```

```
1    <No assembly generated>
```

# Компилируем…

MSVC:

```
static_assert(std::is_same_v<get_type<Foo>::type, Param>);
```

```
1    <No assembly generated>
```

Clang:

```
static_assert(std::is_same_v<get_type<Foo>::type, Param>);
```

```
1    <No assembly generated>
```

# Компилируем…

MSVC:

```
static_assert(std::is_same_v<get_type<Foo>::type, Param>);
```

```
1    <No assembly generated>
```

Clang:

```
static_assert(std::is_same_v<get_type<Foo>::type, Param>);
```

```
1    <No assembly generated>
```

GCC:

```
static_assert(std::is_same_v<get_type<Foo>::type, Param>);
```

```
<source>:13:17: error: redefinition of 'auto get(DefInserter<Foo>)'
   13 |     friend auto get(Key) {
      |                 ^~~
<source>:13:17: note: 'auto get(DefInserter<Foo>)' previously declared here
<source>:44:20: error: static assertion failed
   44 | static_assert(std::is_same_v<get_type<Foo>::type, Param>);
      |               ~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

# Текущий статус type loopholes

Richard Smith:

“For a function that is inline or has a deduced return type, the name lookup performed for each reference to the function that requires a definition to exist shall find a declaration of the function that is introduced in the same scope as the definition. For this purpose:
-- a function declaration with a qualfiied name is considered to be introduced in the scope of the declaration named by its nested-name-specifier,
-- a non-defining friend function declaration with an unqualified name is considered to be introduced in the scope in which the function is introduced or redeclared,
-- a function declaration at block scope is considered to be introduced in the innermost enclosing namespace scope, and
-- any other function declaration is considered to be introduced in the scope in which it appears. [Note: Friend function definitions are introduced in the enclosing class scope.]”

# Текущий статус type loopholes

Richard Smith:

“The specific case of this for CWG2118 would be: for a function F defined in a friend declaration in some class C, the class C shall be an associated class for all potentially-evaluated calls to F.

It's not yet clear whether this is too restrictive (there are cases where this forces a friend function to be defined outside of a class where it is currently defined inside the class, so it would be a breaking change for code that might be considered reasonable), and we've certainly not made any decision. But every time the core working group has considered the issue, they have concluded that they want to make such code ill-formed somehow.”