Customization Points that Suck Less

Michał Dominiak
Nokia Networks
griwes@griwes.info
@Guriwesu

Outline

1. Introduction
2. Existing and well known techniques
3. Interludes
4. Proposed technique
5. Tie-in with Concepts TS

What is a customization point?

A customization point is a well-defined way to specify the behavior of a feature for own types and such.

Is the problem even important?

Yes, as proven by the amount of threads about this topic on std-proposals *and* on the reflectors.

Is this a rant?

Somewhat.

Do you like macros?

Not particularly.

Outline

std::hash, structured bindings: template specializations

```
struct foo {
    int value;
};
```

std::hash, structured bindings: template specializations

```
struct foo {
    int value;
};

namespace std {
    template<>
    struct hash<foo> {
        std::size_t operator()(const foo & f) const {
            return std::hash(f.value);
        }
    };
}
```

std::hash, structured bindings: template specializations

```
struct foo {
    int value;
};

template<>
struct std::hash<foo> {
    std::size_t operator()(const foo & f) const {
        return std::hash(f.value);
    }
};
```

std::hash, structured bindings: template specializations

```
namespace foo {
    struct bar {
        int value;
    };
```

std::hash, structured bindings: template specializations

```
namespace foo {
    struct bar {
        int value;
    };
}

template<>
struct std::hash<foo> {
    std::size_t operator()(const foo & f) const {
        return std::hash(f.value);
    }
};
```

std::hash, structured bindings: template specializations

```
namespace foo {
    struct bar {
        int value;
    };

    template<>
    struct ::std::hash<foo> {
        std::size_t operator()(const foo & f) const {
            return std::hash(f.value);
        }
    };
}
```

std::hash, structured bindings: template specializations

- this is tiresome

std::hash, structured bindings: template specializations

- this is tiresome
- sometimes there's a lot of namespaces you're in

std::hash, structured bindings: template specializations

- this is tiresome
- sometimes there's a lot of namespaces you're in
- makes it impossible to include a header containing a specialization like that inside a namespace

std::hash, structured bindings: template specializations

- some of the customization points for structured bindings are `std::tuple_size` and `std::tuple_element`

std::hash, structured bindings: template specializations

- some of the customization points for structured bindings are `std::tuple_size` and `std::tuple_element`
- both are defined in `<tuple>`

std::hash, structured bindings: template specializations

- some of the customization points for structured bindings are `std::tuple_size` and `std::tuple_element`
- both are defined in `<tuple>`
- that header is not freestanding

std::hash, structured bindings: template specializations

- some of the customization points for structured bindings are `std::tuple_size` and `std::tuple_element`
- both are defined in `<tuple>`
- that header is not freestanding
- and also you need to escape all your namespaces in this case too

std::swap, (proposed) Boost.Yap, structured bindings: ADL

- ADL

`std::swap`, (proposed) Boost.Yap, structured bindings: ADL

- ADL - argument dependent lookup

std::swap, (proposed) Boost.Yap, structured bindings: ADL

- ADL - argument dependent lookup
- arguments to a function call can pull additional overloads into the overload set

std::swap, (proposed) Boost.Yap, structured bindings: ADL

```
template<typename T>
void swap(T & a, T & b) noexcept(/* ... */);
```

std::swap, (proposed) Boost.Yap, structured bindings: ADL

```
template<typename T>
void swap(T & a, T & b) noexcept(/* ... */);

template<typename ForwardIt1, typename ForwardIt2>
void iter_swap(ForwardIt1 a, ForwardIt2 b)
```

std::swap, (proposed) Boost.Yap, structured bindings: ADL

```
template<typename T>
void swap(T & a, T & b) noexcept(/* ... */);

template<typename ForwardIt1, typename ForwardIt2>
void iter_swap(ForwardIt1 a, ForwardIt2 b)
{
    using std::swap;
    swap(*a, *b);
}
```

std::swap, (proposed) Boost.Yap, structured bindings: ADL

- Zach Laine's talk from Tuesday: Expression Templates Everywhere with C++14 and Yap

std::swap, (proposed) Boost.Yap, structured bindings: ADL

- Zach Laine's talk from Tuesday: Expression Templates Everywhere with C++14 and Yap

```
struct expr_thing {
    // ...
};
expr_thing eval_plus(expr_thing lhs, expr_thing rhs) {
    // ...
}
```

std::swap, (proposed) Boost.Yap, structured bindings: ADL

- Zach Laine's talk from Tuesday: Expression Templates Everywhere with C++14 and Yap

```
struct expr_thing {
    // ...
};
expr_thing eval_plus(expr_thing lhs, expr_thing rhs) {
    // ...
}
```

- no way to make the type have different evaluation functions for different contexts

std::swap, (proposed) Boost.Yap, structured bindings: ADL

- structured bindings use ADL to select the overload for get

std::swap, (proposed) Boost.Yap, structured bindings: ADL

- structured bindings use ADL to select the overload for get
- this differs from normal calls by being *only* ADL, without normal overload resolution

ADL is not my favorite

- my own library with an `optional` implementation, with `make_optional`

ADL is not my favorite

- my own library with an `optional` implementation, with `make_optional`
- inside the same library - unqualified call to `make_optional` in generic code

ADL is not my favorite

- my own library with an `optional` implementation, with `make_optional`
- inside the same library - unqualified call to `make_optional` in generic code
- worked great until I tried GCC 7.1

ADL is not my favorite

- my own library with an `optional` implementation, with `make_optional`
- inside the same library - unqualified call to `make_optional` in generic code
- worked great until I tried GCC 7.1
- it turned out I called it on `std::vector`

N4381

Document number:	N4381=yy-nnnn
Date:	2015-03-11
Project:	Programming Language C++, Library Working Group
Reply-to:	Eric Niebler <eniebler@boost.org>,

Suggested Design for Customization Points

`http://wg21.link/n4381`

N4381

```
namespace __detail {
    template<class T, size_t N>
    constexpr T* begin(T (&a)[N]) noexcept {
        return a;
    }
    struct __begin_fn {
        template<class R>
        constexpr auto operator()(R && rng) const ->
            decltype(begin(forward<R>(rng))) {
            return begin(forward<R>(rng));
        }
    };
}
```

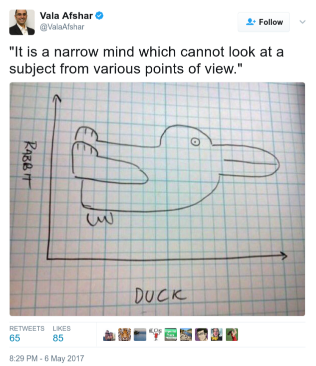
N4381

```
// To avoid ODR violations:
template<class T>
struct __static_const {
    static constexpr T value{};
};
template<class T>
constexpr T __static_const<T>::value;
// std::begin is a global function object!
namespace {
    constexpr auto const & begin =
        __static_const<__detail::__begin_fn>::value;
}
```

Outline

1. Introduction
2. Existing and well known techniques
3. Interludes
4. Proposed technique
5. Tie-in with Concepts TS

The Haskell Interlude: Typeclasses



https://twitter.com/ValaAfshar/status/860924553844969473/photo/1

The Haskell Interlude: Typeclasses

The Haskell Interlude: Typeclasses

```haskell
one = 1
```

The Haskell Interlude: Typeclasses

```
one = 1

const x _ = x
const_12 = const 12
const_12 "abc" -- this "returns" 12
```

The Haskell Interlude: Typeclasses

The Haskell Interlude: Typeclasses

```
one :: Int
one = 1
```

The Haskell Interlude: Typeclasses

```haskell
one :: Int
one = 1

const :: a -> b -> a
const x _ = x
const_12 :: a -> Int
const_12 = const 12
```

The Haskell Interlude: Typeclasses

The Haskell Interlude: Typeclasses

```haskell
one :: Num a => a
one = 1
```

The Haskell Interlude: Typeclasses

```haskell
one :: Num a => a
one = 1

const :: a -> b -> a
const x _ = x
const_12 :: Num a => b -> a
const_12 = const 12
```

The Haskell Interlude: Typeclasses

```
class Num a where
    (+), (-), (*) :: a -> a -> a
    negate :: a -> a
    abs :: a -> a

    fromInteger :: Integer -> a

    x - y = x + negate y
    negate x = 0 - x
```

The Haskell Interlude: Typeclasses

```
data Complex a = Complex a a
foo = Complex 1.2 3.4
```

The Haskell Interlude: Typeclasses

```haskell
data Complex a = Complex a a
foo = Complex 1.2 3.4

instance Num a => Num (Complex a) where
    Complex x y + Complex u v = Complex (x + u) (y + v)
    Complex x y - Complex u v = Complex (x - u) (y - v)
    Complex x y * Complex u v = Complex (x * u - y * v) (x * v + y * u)

    fromInteger x = Complex (fromInteger x) 0
    negate (Complex x y) = Complex (negate x) (negate y)
```

The Haskell Interlude: Typeclasses

```haskell
data Complex a = Complex a a
foo = Complex 1.2 3.4

instance Num a => Num (Complex a) where
    Complex x y + Complex u v = Complex (x + u) (y + v)
    Complex x y - Complex u v = Complex (x - u) (y - v)
    Complex x y * Complex u v = Complex (x * u - y * v) (x * v + y * u)

    fromInteger x = Complex (fromInteger x) 0
    negate (Complex x y) = Complex (negate x) (negate y)

bar = foo - Complex 1 2
```

The Haskell Interlude: Typeclasses

```haskell
data Complex a = Complex a a
foo = Complex 1.2 3.4

instance Num a => Num (Complex a) where
    Complex x y + Complex u v = Complex (x + u) (y + v)
    Complex x y - Complex u v = Complex (x - u) (y - v)
    Complex x y * Complex u v = Complex (x * u - y * v) (x * v + y * u)

    fromInteger x = Complex (fromInteger x) 0
    negate (Complex x y) = Complex (negate x) (negate y)

bar = foo - Complex 1 2

baz = bar + 1
```

The Haskell Interlude: Typeclasses

```
data Complex a = Complex a a
foo = Complex 1.2 3.4

instance Num a => Num (Complex a) where
    Complex x y + Complex u v = Complex (x + u) (y + v)
 -- Complex x y - Complex u v = Complex (x - u) (y - v)
    Complex x y * Complex u v = Complex (x * u - y * v) (x * v + y * u)

    fromInteger x = Complex (fromInteger x) 0
    negate (Complex x y) = Complex (negate x) (negate y)

bar = foo - Complex 1 2

baz = bar + 1
```

The C++0x Interlude: Concept Maps

The C++0x Interlude: Concept Maps

```
auto concept LessThanComparable<typename T> {
    bool operator<(T, T);
}
```

The C++0x Interlude: Concept Maps

```
auto concept LessThanComparable<typename T> {
    bool operator<(T, T);
}

template<typename T>
    requires LessThanComparable<T>
const T & min(const T & x, const T & y) {
    return (y < x) ? y : x;
}
```

The C++0x Interlude: Concept Maps

```
auto concept Numeric<typename T> {
    T operator+(const T &, const T &);
    T operator-(const T &, const T &);
    T operator*(const T &, const T &);

    T negate(const T &);
    T abs(const T &);
};
```

The C++0x Interlude: Concept Maps

```
template<typename T>
struct complex {
    T real, comp;
};
```

The C++0x Interlude: Concept Maps

```
template<typename T>
struct complex {
    T real, comp;
};

template<typename T>
concept_map Numeric<complex<T>> {
    complex<T> operator+(const complex<T> & lhs, const complex<T> & rhs) {
        return { lhs.real + rhs.real, lhs.comp + rhs.comp };
    }

    // ...
};
```

Outline

Basic idea

- We can specialize a member of the current namespace without leaving it.

Basic idea

- We can specialize a member of the current namespace without leaving it.
- The class template that the library facility will look into is not in the current namespace.

Basic idea

- We can specialize a member of the current namespace without leaving it.
- The class template that the library facility will look into is not in the current namespace.
- A type we are defining *is* in the current namespace.

Basic idea

```
namespace foo {
    template<typename T>
    struct tuple_size {
        static constexpr const std::size_t value = T::tuple_size_instance::value;
    };
}
namespace bar {
    struct baz {
        struct tuple_size_instance {
            static constexpr const std::size_t value = 17;
        };
    };
}
```

Basic idea

```
namespace foo {
    struct tuple_size_tc {};
    template<typename T>
    struct tuple_size {
        static constexpr const std::size_t value = T::template instance<tuple_size_tc>::value;
    };
}
namespace bar {
    struct baz {
        template<typename T>
        struct instance;
    };
    template<>
    struct baz::instance<foo::tuple_size_tc> {
        static constexpr const std::size_t value = 17;
    };
}
```

Basic idea

```
namespace foo {
    struct tuple_size_tc {};
    template<typename T>
    struct tuple_size {
        static constexpr const std::size_t value = T::template instance<tuple_size_tc>::value;
    };
}
namespace bar {
    struct baz {
        template<typename T>
        struct instance;
    };
    template<>
    struct baz::instance<foo::tuple_size_tc> {
        static constexpr const std::size_t value = 17;
    };
}
```

Basic idea

```
namespace bar {
    INSTANCE(foo::tuple_size_tc, baz) {
        static constexpr const std::size_t value = 17;
    };
}
```

Default instances

```
struct tuple_size_tc {
    template<typename Typeclass, typename T>
    struct instance;
};
template<typename T>
struct tuple_size {
    static constexpr const std::size_t value
        = T::template instance<tuple_size_tc, T>::value;
};
template<typename T>
struct tuple_size_tc::instance<tuple_size_tc, T> {
    static constexpr const std::size_t value = 0;
};
```

Default instances

```
template<typename Typeclass, typename T, typename = void>
struct typeclass_trait {
    using type = typename Typeclass::template instance<Typeclass, T>;
};
template<typename Typeclass, typename T>
struct typeclass_trait<Typeclass, T,
        void_t<typename T::template instance<Typeclass, T>>> {
    using type = typename T::template instance<Typeclass, T>;
};
template<typename Typeclass, typename T>
using tc_instance = typename typeclass_trait<Typeclass, T>::type;
```

Default instances

```
struct tuple_size_tc {
    template<typename Typeclass, typename T>
    struct instance;
};
template<typename T>
struct tuple_size {
    static constexpr const std::size_t value = tc_instance<tuple_size_tc, T>::value;
};
```

Default instances

```
struct tuple_size_tc {
    template<typename Typeclass, typename T>
    struct instance;
};
template<typename T>
struct tuple_size {
    static constexpr const std::size_t value = tc_instance<tuple_size_tc, T>::value;
};

template<typename T>
struct tuple_size_tc::instance<tuple_size_tc, T> {
    static constexpr const std::size_t value = 0;
};
template<typename T>
struct tuple_size_tc::instance<tuple_size_tc, std::vector<T>> {
    static constexpr const std::size_t value = 17;
};
```

Default instances

```
DEFAULT_INSTANCE(tuple_size_tc, T) {
    static constexpr const std::size_t value = 0;
};
```

Default instances

```
struct swappable
{
    TYPECLASS_INSTANCE(typename T);
};
```

Default instances

```
struct swappable
{
    TYPECLASS_INSTANCE(typename T);
};

template<typename T>
auto swap(T & lhs, T & rhs) -> decltype(tc_instance<swappable, T>::swap(lhs
    noexcept(noexcept(tc_instance<swappable, T>::swap(lhs, rhs)))
{
    tc_instance<swappable, T>::swap(lhs, rhs);
};
```

Default instances

```
struct swappable
{
    TYPECLASS_INSTANCE(typename T);
};

template<typename T>
auto swap(T & lhs, T & rhs) SFINAE_FUNCTION(
    tc_instance<swappable, T>::swap(lhs, rhs)
)
```

Default instances

```
DEFAULT_INSTANCE(swappable, T)
{
    // simplified *a lot* for the purpose of the talk
    template<typename T>
    static void swap(T & lhs, T & rhs)
    {
        auto tmp = std::move(lhs);
        lhs = std::move(rhs);
        rhs = std::move(tmp);
    }
};
```

Enter templates: everything is terrible

```
template<typename T>
struct counter_definition {
    void set_value(T);
    T get_value() const;
    T increment();
};
```

Enter templates: everything is terrible

```
template<typename T>
struct counter {
    template<typename Typeclass, typename T>
    struct instance;

    template<typename Counter>
    static void set_value(Counter c, T t) SFINAE_FUNCTION(
        tc_instance<counter_tc, T>::set_value(c, t);
    )
};
```

Enter templates: everything is terrible

```
template<typename T>
DEFAULT_INSTANCE(counter<T>, U)
{
    static void set_value(U & u, T val) { u.set_value(std::move(val)); }
    static T get_value(const U & u) { return u.get_value(); }
    static T increment(U & u) { return ++u; }
};
```

Enter templates: everything is terrible

```
template<typename T>
struct atomic_counter
{
    TYPECLASS_INSTANCE(typename U);

    void set_value(T val) { _value = std::move(val); }
    T get_value() const { return _value; }
    T add(T v) { return _value += std::move(v); }
private:
    std::atomic<T> _value;
};
```

Enter templates: everything is terrible

```
template<typename T>
INSTANCE(counter<T>, atomic_counter<T>)
{
    static T increment(atomic_counter<T> & c) { return c.add(1); }
};
```

Enter templates: everything is terrible

```
template<typename T>
INSTANCE(counter<T>, atomic_counter<T>)
{
    static T increment(atomic_counter<T> & c) { return c.add(1); }
};
```

error: cannot specialize (with 'template<>') a member of an unspecialized template

Enter templates: everything is terrible

```
template<typename T>
struct counter {
    template<typename Typeclass, typename U>
    struct instance;
};
template<typename T>
struct atomic_counter {
    template<typename Typeclass, typename U>
    struct instance;
};
template<typename T> template<>
struct atomic_counter<T>::instance<counter<T>, atomic_counter<T>>
    : counter<T>::template instance<counter<T>, atomic_counter<T>> {};
```

Enter templates: everything is terrible

```
INSTANCE_TEMPLATE_HELPER;

template<typename T>
struct atomic_counter
{
    TYPECLASS_INSTANCE_TEMPLATE((typename U), (atomic_counter), counter);
    void set_value(T val) { _value = std::move(val); }
    T get_value() const { return _value; }
    T add(T v) { return _value += std::move(v); }
private:
    std::atomic<T> _value;
};
```

Enter templates: everything is terrible

```
INSTANCE_TEMPLATE((typename T), (atomic_counter), counter,
    atomic_counter<T>)
{
    static T increment(atomic_counter<T> & c) { return c.add(1); }
};
```

Enter templates: everything is terrible

```
#define INSTANCE_TEMPLATE_HELPER                                          \
    template<typename Typeclass, typename Class>                         \
    struct typeclass_instance_helper
```

Enter templates: everything is terrible

```
#define INSTANCE_TEMPLATE_HELPER                                          \
    template<typename Typeclass, typename Class>                          \
    struct typeclass_instance_helper

#define TYPECLASS_INSTANCE_TEMPLATE(template_decl, template_args, class)  \
    template<typename _typeclass, ONLY template_decl>                     \
    struct instance : typeclass_instance_helper<_typeclass, ONLY template_args> \
    {                                                                     \
    }
```

Enter templates: everything is terrible

```
#define INSTANCE_TEMPLATE_HELPER                                                    \
    template<typename Typeclass, typename Class>                                    \
    struct typeclass_instance_helper

#define TYPECLASS_INSTANCE_TEMPLATE(template_decl, template_args, class)            \
    template<typename _typeclass, ONLY template_decl>                               \
    struct instance : typeclass_instance_helper<_typeclass, ONLY template_args>     \
    {                                                                               \
    }

#define INSTANCE_TEMPLATE(template_decl, template_args, typeclass, class)           \
    template<ONLY template_decl>                                                    \
    struct typeclass_instance_helper<typeclass<ONLY template_args>, class>          \
        : public typeclass<ONLY template_args>                                      \
            ::template instance<typeclass<ONLY template_args>, class>               \
```

Type erasure

```
struct hashable_definition {
    using hash = std::size_t () const;
};
```

Type erasure

```
struct hashable_definition {
    using hash = std::size_t () const;
};

DEFINE_TYPECLASS(hashable, hash);
```

Type erasure

```
struct hashable_definition {
    using hash = std::size_t () const;
};

DEFINE_TYPECLASS(hashable, hash);

template<typename T>
auto hash(const T & t) SFINAE_FUNCTION(tc_instance<hashable, T>::hash(t));
```

Type erasure

```
hashable::erased erased = 123;
```

Type erasure

```
hashable::erased erased = 123;

erased.hash();
```

Type erasure

```
hashable::erased erased = 123;

erased.hash();

hash(erased);
```

Type erasure

```
hashable::erased erased = 123;

erased.hash();

hash(erased);

int foo = 456;
```

Type erasure

```
hashable::erased erased = 123;
erased.hash();
hash(erased);
int foo = 456;
hashable::erased_ref ref = foo;
```

Type erasure

```
hashable::erased erased = 123;
erased.hash();
hash(erased);
int foo = 456;
hashable::erased_ref ref = foo;
ref.hash();
```

Type erasure

```
hashable::erased erased = 123;

erased.hash();

hash(erased);

int foo = 456;

hashable::erased_ref ref = foo;

ref.hash();

foo = 789;
hash(erased);
```

Type erasure

```
#define TYPECLASS_PREPARE_MIXINS(x, typeclass_name, memfn_name)        \
    template<typename ReturnType, typename... Args>                    \
    struct CONCAT3(typeclass_name, _typeclass_base_provide_, memfn_name) \
        : protected ::virtual_dtor<class typeclass_base_provide_>      \
    {                                                                  \
        virtual ReturnType memfn_name(Args...) = 0;                    \
    };
```

Type erasure

```
#define TYPECLASS_BASE_BASE_CLASS(x, typeclass_info, memfn_name)                 \
    TYPECLASS_BASE_BASE_CLASS_IMPL(FIRST typeclass_info, SECOND typeclass_info, memfn_name)
#define TYPECLASS_BASE_BASE_CLASS_IMPL(typeclass_name, template_args, memfn_name)     \
public                                                                                \
    virtual ::explode<typename CONCAT(typeclass_name, _definition) template_args::memfn_name,   \
        CONCAT3(typeclass_name, _typeclass_base_provide_, memfn_name)>,
```

Type erasure

```
#define TYPECLASS_BASE_BASE_CLASS(x, typeclass_info, memfn_name)                \
    TYPECLASS_BASE_BASE_CLASS_IMPL(FIRST typeclass_info, SECOND typeclass_info, memfn_name)
#define TYPECLASS_BASE_BASE_CLASS_IMPL(typeclass_name, template_args, memfn_name)   \
public                                                                          \
    virtual ::explode<typename CONCAT(typeclass_name, _definition) template_args::memfn_name,  \
        CONCAT3(typeclass_name, _typeclass_base_provide_, memfn_name)>,

#define DEFINE_TYPECLASS_SEQ(template_decl, template_args, typeclass_name, member_list)  \
    BOOST_PP_SEQ_FOR_EACH(TYPECLASS_PREPARE_MIXINS, typeclass_name, member_list)        \
    ...                                                                                  \
struct _base : BOOST_PP_SEQ_FOR_EACH(TYPECLASS_BASE_BASE_CLASS,                          \
                  (typeclass_name)(template_args),                                       \
                  member_list) ::virtual_dtor<_base>                                     \
{                                                                                        \
    virtual ~_base() = default;                                                          \
    virtual std::unique_ptr<_base> clone() const = 0;                                    \
};                                                                                       \
```

Reflection with `idreflexpr` would be great

```
#define TYPECLASS_PREPARE_MIXINS(x, typeclass_name, memfn_name)      \
    template<typename ReturnType, typename... Args>                  \
    struct CONCAT3(typeclass_name, _typeclass_base_provide_, memfn_name) \
        : protected ::virtual_dtor<class typeclass_base_provide_>    \
    {                                                                \
        virtual ReturnType memfn_name(Args...) = 0;                  \
    };
```

Reflection with `idreflexpr` would be great

```
DEFINE_TYPECLASS(hashable); // reflexpr magic to enumerate the members
```

Reflection with idreflexpr would be great

```
DEFINE_TYPECLASS(hashable); // reflexpr magic to enumerate the members

template<typename MetaObject, typename ReturnType, typename... Args>
struct typeclass_base_provide_member
{
    virtual ~typeclass_base_provide_member() {}

    virtual ReturnType idreflexpr(MetaObject::name)(Args...) = 0;
};
```

Reflection with idreflexpr would be great

```
DEFINE_TYPECLASS(hashable); // reflexpr magic to enumerate the members

template<typename MetaObject, typename ReturnType, typename... Args>
struct typeclass_base_provide_member
{
    virtual ~typeclass_base_provide_member() {}

    virtual ReturnType idreflexpr(MetaObject::name)(Args...) = 0;
};
```

Go watch Jackie Kay's talk on reflection.

Outline

1. Introduction
2. Existing and well known techniques
3. Interludes
4. Proposed technique
5. Tie-in with Concepts TS

Crash course in Concepts TS

```
template<typename T>
bool concept True = true;
```

Crash course in Concepts TS

```
template<typename T>
bool concept True = true;

template<typename T>
bool concept HasFoo = requires(T t) {
    t.foo();
};
```

Crash course in Concepts TS

```
template<typename T>
bool concept True = true;

template<typename T>
bool concept HasFoo = requires(T t) {
    t.foo();
};

template<typename T>
bool concept HasFooVoidReturn = requires(T t) {
    { t.foo() } -> void;
};
```

Generating Concepts for typeclasses

Generating Concepts for typeclasses

```
template<typename T, typename Return, typename... Args>
bool concept HasFunctionFoo = requires(T t, Args... args) {
    { t.foo(args...) } -> Return;
};
```

Generating Concepts for typeclasses

```
template<typename T, typename FuncType>
    requires explode<FuncType, HasFunctionFoo, T>::...???
```

Generating Concepts for typeclasses

```
template<typename T, typename FuncType>
bool concept HasFunctionFoo = false;

template<typename T, typename Return, typename... Args>
bool concept HasFunctionFoo<T, Return (Args...)>
    = requires(T t, Args... args) {
        { t.foo(args...) } -> Return;
    };
```

Generating Concepts for typeclasses

```
template<typename T, typename FuncType>
bool concept HasFunctionFoo = false;

template<typename T, typename Return, typename... Args>
bool concept HasFunctionFoo<T, Return (Args...)>
    = requires(T t, Args... args) {
        { t.foo(args...) } -> Return;
    };
```

error: specialization of variable concept 'template⟨class T, class FuncType⟩ concept const bool HasFoo⟨T, FuncType⟩'

Generating Concepts for typeclasses

```
template<typename T, typename Return, typename... Args>
bool concept HasFoo(Return (*)(Args...)) {
    return requires(T t, Args... args) {
        { t.foo(args...) } -> Return;
    };
}
```

Generating Concepts for typeclasses

```
template<typename T, typename Return, typename... Args>
bool concept HasFoo(Return (*)(Args...)) {
    return requires(T t, Args... args) {
        { t.foo(args...) } -> Return;
    };
}
```

error: concept 'concept bool HasFoo(Return(*)(Args ...))' declared with function parameters

Generating Concepts for typeclasses

```
template<typename T, typename Return, typename... Args>
bool concept HasFunctionFoo = requires(T t, Args... args) {
    { t.foo(args...) } -> Return;
};

template<typename... Ts>
using has_foo_wrapped = std::bool_constant<HasFunctionFoo<Ts...>>;
```

Generating Concepts for typeclasses

```
template<typename T, typename Return, typename... Args>
bool concept HasFunctionFoo = requires(T t, Args... args) {
    { t.foo(args...) } -> Return;
};

template<typename... Ts>
using has_foo_wrapped = std::bool_constant<HasFunctionFoo<Ts...>>;

template<typename T>
bool concept Fooable = std::conjunction_v<has_foo_wrapped, ...>;
```

Virtual concepts

https://github.com/andyprowl/virtual-concepts

```
void print(const std::vector<Shape *> & v) { // a template
    for (const auto & s : v) {
        std::cout << s->get_area() << " ";
    }
}
```

Virtual concepts

https://github.com/andyprowl/virtual-concepts

```
void print(const std::vector<Shape *> & v) { // a template
    for (const auto & s : v) {
        std::cout << s->get_area() << " ";
    }
}

void print(const std::vector<virtual Shape *> & v) { // not a template
    for (const auto & s : v) {
        std::cout << s->get_area() << " ";
    }
}
```

Future directions for type erasure wrappers

- actually make a use of abominable function types

Future directions for type erasure wrappers

- actually make a use of abominable function types
- actually generate TS concepts for this

Future directions for type erasure wrappers

- actually make a use of abominable function types
- actually generate TS concepts for this (though I'm not 100% convinced this'll actually improve anything)

Future directions for type erasure wrappers

- actually make a use of abominable function types
- actually generate TS concepts for this (though I'm not 100% convinced this'll actually improve anything)
- allow overloading of the erased functions

Future directions for type erasure wrappers

- actually make a use of abominable function types
- actually generate TS concepts for this (though I'm not 100% convinced this'll actually improve anything)
- allow overloading of the erased functions
- use Louis Dionne's Dyno as a backend

Summary

- Types need to be first class, though I don't have much hope.

Summary

- Types need to be first class, though I don't have much hope.
- The Concepts TS needs concept maps in one of its revisions.

Summary

- Types need to be first class, though I don't have much hope.
- The Concepts TS needs concept maps in one of its revisions.
- The Concepts TS needs type erasure support.

Questions and Answers

Questions and Answers

Thank you!