Checking *expression validity* in C++11/14/17

Meeting[0] - C++::London - 05/12/2016

by Vittorio Romeo (**@supahvee1234**)

```cpp
struct Cat
{
    void meow() const { cout << "meow\n"; }
};

struct Dog
{
    void bark() const { cout << "bark\n"; }
};
```

```cpp
template <typename T>
void make_noise(const T& x)
{
    // Pseudocode:

    if( <`x.meow()` is well-formed> )
    {
        x.meow();
    }
    else if( <`x.bark()` is well-formed> )
    {
        x.bark();
    }
    else
    {
        <compile-time error>
    }
}
```

```
make_noise(Cat{}); // "meow"
make_noise(Dog{}); // "bark"
make_noise(int{}); // compile-time error
```

```
template <typename T>
void make_noise(const T& x){ /* ? */ }
```

- **C++11**: `std::void_t` and `std::enable_if`.

- **C++14**: `boost::hana::is_valid` and `vrm::core::static_if`.

- **C++17**: `if constexpr( ... )`, `constexpr` lambdas, and `std::is_callable`.

C++11

`std::void_t` and `std::enable_if`

Combining `std::void_t` with `std::enable_if` allows us to detect *ill-formed* expressions in SFINAE contexts.

- "Modern Template Metaprogramming: A Compendium"

  by *Walter E. Brown* at CppCon 2014

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

(`void_t` *was standardized in C++17, but can be implemented in C++11.)*

```cpp
template <typename, typename = void>
struct has_meow : std::false_type { };

template <typename T>
struct has_meow
<
    T,
    void_t<decltype(std::declval<T>().meow())>
>
: std::true_type { };
```

Instantiating `has_meow<T>` will attempt to evaluate

```
void_t<decltype(std::declval<T>().meow())>
```

If `declval<T>().meow()` is *well-formed*,

```
void_t<decltype(std::declval<T>().meow())>
```

will evaluate to `void` , and `has_meow` 's `std::true_type` specialization will be taken.

```cpp
template <typename, typename = void>
struct has_meow : std::false_type { };

template <typename T>                                      // <<<
struct has_meow<T, void_t<decltype(std::declval<T>().meow())>> // <<<
    : std::true_type { };                                 // <<<
```

If `declval<T>().meow()` is *ill-formed*,

```
void_t<decltype(std::declval<T>().meow())>
```

will be *ill-formed* as well, *SFINAE-ing* away the `std::true_type` specialization. All that's left is the `std::false_type` specialization.

```cpp
template <typename, typename = void>   // <<<
struct has_meow : std::false_type { }; // <<<

template <typename T>
struct has_meow<T, void_t<decltype(std::declval<T>().meow())>>
    : std::true_type { };
```

How does `void_t` work?

```
auto meows = has_meow<Cat>{};
```

…is equivalent to…

```
auto meows = has_meow<Cat, void>{};
```

…because of the default `void` parameter.

The compiler takes into account *partial specializations*.

- `T` is matched to `T`

- `void` is matched to

```
void_t<decltype(std::declval<T>().meow())>
```

If `void_t<decltype(std::declval<T>().meow())>` doesn't get *SFINAE-d away*, both the types of the original templates and the specialization match.

The partial specialization is prioritized and chosen.

If the original template's second parameter wasn't defaulted to `void`, it would have been chosen over the partial specialization!

After defining the `has_bark` detector class *(which is trivial to implement, as well)*, all that's left to do is use `std::enable_if` to constrain `make_noise`.

```cpp
template <typename T>
auto make_noise(const T& x)
    → typename std::enable_if<has_meow<T>{}>::type
{
    x.meow();
}


template <typename T>
auto make_noise(const T& x)
    → typename std::enable_if<has_bark<T>{}>::type
{
    x.bark();
}
```

C++11 implementation issues:

- A *detector class* must be defined for **every** expression to check.
  - The class **cannot be defined locally**.
- `std::enable_if` must be used to constrain multiple versions of the same function.
  - It is not possible to "*branch*" **locally** at compile-time.
  - It is necessary to **repeat** the function signature.

C++14

`boost :: hana :: is_valid` and `vrm :: core :: static_if`

We can solve the aforementioned issues thanks to

♡ *generic lambdas* ♡

Generic lambdas are *"templates in disguise"* - they provide a
SFINAE-friendly context.

```cpp
auto l = [](auto x){ return x; };
```

...is somewhat equivalent to...

```cpp
struct ανσνумσυ
{
    template <typename T>
    auto operator()(T x) const { return x; }
};
```

Let's take advantage of that...

```cpp
auto has_meow =
    is_valid([](auto&& x) → decltype(x.meow())){});

static_assert(has_meow(Cat{}), "");
static_assert(!has_bark(Cat{}), "");
```

- `has_meow` can be **locally** instantiated in any scope.

- Terser & nicer syntax.

How can we implement `is_valid` ?

```cpp
template <typename TF>
constexpr auto is_valid(TF)
{
    return validity_checker<TF>{};
}
```

```cpp
template <typename TF>
struct validity_checker
{
    template <typename ... Ts>
    constexpr auto operator()(Ts&& ... ) const
    {
        return is_callable<TF(Ts ... )>{};
    }
};
```

Remember that `TF` is the type of:

```cpp
[](auto&& x) → decltype(x.meow()) { }
```

How can we implement `is_callable` ?

```cpp
template <typename, typename = void>
struct is_callable : std::false_type { };

template <typename TF, class ... Ts>
struct is_callable
<
    TF(Ts ... ),

    void_t<decltype(
        std::declval<TF>()(std::declval<Ts>() ... )
    )>
>
: std::true_type { };
```

`is_valid` solves the first C++11 *annoyance.*

- Detectors can be instantiated locally.

Solving the second issue *(local compile-time branching)* is slightly more complicated, but **it can be done**.

I explain how in my **CppCon 2016 talk**:
"Implementing `static` control flow in C++14".

```cpp
template <typename T>
auto make_noise(const T& x)
{
    auto has_meow = is_valid([](auto&& x)
        → decltype(x.meow()){ });

    auto has_bark = is_valid([](auto&& x)
        → decltype(x.bark()){ });

    static_if(has_meow(x))
        .then([&x](auto){ x.meow(); })

        .else_if(has_bark(x))
        .then([&x](auto){ x.bark(); })

        .else_([](auto)
            {
                struct cannot_meow_or_bark;
                cannot_meow_or_bark{};
            })();
}
```

Is it *nicer* than the C++11 version? **Debatable.**

There are some *objective advantages*, though:

- Expression validity detector instantiation is **local to the function scope**.

- There is a **single overload** of `make_noise`.

- Compile-time branching is **local to the function scope**.

- `boost :: hana:is_valid` is a production-ready C++14 implementation of the above `is_valid` function.

- You can find my `static_if` implementation in `vrm :: core :: static_if` .

C++14 implementation issues:

- `is_valid` has to be assigned to a variable in order to be used in a constant expression.

    - This happens because lambdas are not `constexpr`.

- **Verbosity.**

    - `static_if` makes the code much less readable.

    - Having to create a lambda with a `decltype( ... )` *trailing return type.*

C++17

- `if constexpr( ... )`

- `constexpr` lambdas

- `std::is_callable`

- Variadic macro *black magic*

```cpp
template <typename T>
auto make_noise(const T& x)
{
    if constexpr(IS_VALID(_0.meow())(T))
    {
        x.meow();
    }
    else if constexpr(IS_VALID(_0.bark())(T))
    {
        x.bark();
    }
    else
    {
        struct cannot_meow_or_bark;
        cannot_meow_or_bark{};
    }
}
```

`IS_VALID(_0.meow())(T)` is a variadic macro that:

- Takes an expression built with *type placeholders*.

- Takes some *types*.

- Evaluates to `true` if the expression is valid for the given types.

```cpp
// Can `T` be dereferenced?
IS_VALID(*_0)(T);

// Can `T0` and `T1` be added together?
IS_VALID(_0 + _1)(T0, T1);

// Can `T` be streamed into itself?
IS_VALID(_0 << _0)(T);

// Can a tuple be made out of `T0`, `T1` and `float`?
IS_VALID(std::make_tuple(_0, _1, _2))(T0, T1, float);
```

`IS_VALID` can be used in contexts where only a *constant expression* is accepted such as `static_assert( ... )` or `if constexpr( ... )` .

What is this magic!?

Let's begin by defining some utilities...

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


template <typename T>
constexpr type_w<T> type_c{};
```

`type_c` is a `constexpr` *variable template* that **wraps a type into a value**.

```
constexpr auto wrapped_int = type_c<int>;

using unwrapped_int =
    typename decltype(wrapped_int)::type;
```

`type_c` is useful because it can be passed to *template functions* like a regular value, **retaining the type information**.

```cpp
template <typename TF>
constexpr auto is_valid(TF)
{
    return validity_checker<TF>{};
}
```

Identical to the previous version.

```cpp
template <typename TF>
struct validity_checker
{
    template <typename ... Ts>
    constexpr auto operator()(Ts ... ts)
    {
        return std::is_callable<
            TF(typename decltype(ts)::type ... )
        >{};
    }
};
```

Expects a bunch of `type_c` values, then unwraps them into `std::is_callable` *(standardized in C++17)*.

The lambda passed as `TF` is almost identical as well:

```
is_valid([](auto _0) constexpr → decltype(_0.meow())){ }
```

...but there's a `constexpr` in there!

constexpr lambdas were standardized in C++17.

```cpp
// Make sure that `int*` can be dereferenced.
static_assert(
    is_valid([](auto _0) constexpr → decltype(*_0){})
    (type_c<int*>)
);
```

This technique is very verbose...

```cpp
template <typename T>
auto make_noise(const T& x)
{
    if constexpr(is_valid([](auto _0) constexpr
        →decltype(_0.meow())(T))
    {
        x.meow();
    }
    else if constexpr(is_valid([](auto _0) constexpr
        →decltype(_0.bark())(T))
    {
        x.bark();
    }
    else
    {
        struct cannot_meow_or_bark;
        cannot_meow_or_bark{};
    }
}
```

That's why we need a **macro**.

```
#define IS_VALID_1_EXPANDER(type0) \
    (type_c<type0>)

#define IS_VALID_1( ... ) \
    is_valid( \
        [](auto _0) constexpr \
            → decltype(__VA_ARGS__){} \
    ) \
    IS_VALID_1_EXPANDER
```

```
#define IS_VALID_2_EXPANDER(type0, type1) \
    (type_c<type0>, type_c<type1>)

#define IS_VALID_2( ... ) \
    is_valid( \
        [](auto _0, auto _1) constexpr \
            → decltype(__VA_ARGS__){} \
    ) \
    IS_VALID_2_EXPANDER
```

```
#define IS_VALID_3_EXPANDER(type0, type1, type2) \
    (type_c<type0>, type_c<type1>, type_c<type2>)

#define IS_VALID_3( ... ) \
    is_valid( \
        [](auto _0, auto _1, auto _2) constexpr \
            → decltype(__VA_ARGS__){} \
    ) \
    IS_VALID_3_EXPANDER
```

...with some `vrm_pp` *preprocessor metaprogramming* and some pre-generated code...

```
#define IS_VALID( ... ) \
    VRM_PP_CAT( \
        IS_VALID_, VRM_PP_ARGCOUNT( __VA_ARGS__ ) \
      )( __VA_ARGS__ )
```

```
IS_VALID(*_0)(int*)
```

...expands to...

```
is_valid(
    [](auto _0) constexpr → decltype(*_0){})
)(type_c<int*>)
```

...which is equivalent to...

```
std::is_callable<
    decltype(
        [](auto _0) constexpr → decltype(*_0){})
    )(typename decltype(type_c<int*>)::type)
>{}
```

This technique is very useful when combined with `if constexpr( ... )` - it's a barebones *in-place* concept definition&check.

```cpp
template <typename T0, typename T1>
auto some_generic_function(T0 a, T1 b)
{
    if constexpr(IS_VALID(foo(_0, _1))(T0, T1))
    {
        return foo(a, b);
    }
    else if constexpr(IS_VALID(_0 + _1)(T0, T1))
    {
        return a + b;
    }

    // ...
}
```

```cpp
template <typename TC, typename T>
auto unify_legacy_apis(TC& c, T x)
{
    if constexpr(IS_VALID(_0.erase(_1))(TC, T))
    {
        return c.erase(x);
    }
    else if constexpr(IS_VALID(_0.remove(_1))(TC, T))
    {
        return c.remove(x);
    }

    // ...
}
```

```cpp
template <typename T>
auto poor_man_ufcs(T& x)
{
    if constexpr(IS_VALID(_0.foo())(T))
    {
        return x.foo();
    }
    else if constexpr(IS_VALID(foo(_0))(T))
    {
        return foo(x);
    }
}
```

Small caveat: it does not yet compile.

- `clang++` hasn't implemented support for `constexpr` lambdas yet.

- `g++` has, but there's a bug I found and reported *(as #78131)*.

`IS_VALID` does work properly with `g++` trunk in other contexts where a *constant expression* is required though *(e.g. non-template context* `if constexpr( ... )` *and* `static_assert` *).*

Thanks for your time!

- https://vittorioromeo.info

- vittorio.romeo@outlook.com

- https://github.com/SuperV1234

- @supahvee1234