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

ACCU 2017 - 28/04/2017

by Vittorio Romeo (@supahvee1234)

Bloomberg

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

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

```cpp
template <typename T>
void pet(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
    }
}
```

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

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

C++11

- `std::void_t`
- `std::enable_if`

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

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 { };
```

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

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

This pattern will be standardized as the **detection idiom**.

Marshall Clow gave a talk about it on Wednesday:

"The Detection Idiom - a simpler way to SFINAE"

C++14

- `boost::hana::is_valid`
- `vrm::core::static_if`
- Generic lambdas

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

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

How can we implement `is_valid`?

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

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

Local compile-time branching in C++14 is slightly more complicated, but **it can be done**.

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

```
template <typename T>
auto pet(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([&](auto){ x.meow(); })

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

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

- `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++17

- `if constexpr( ... )`
- `constexpr` lambdas
- `std::is_callable`
- Variadic macro *black magic*

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

`IS_VALID(T)(_0.meow())` 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.

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

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

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

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

`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!?

```
template <typename ... Ts, typename TF>
constexpr auto is_valid(TF)
{
    return is_callable<TF(Ts ... )>{};
}
```

- Deduce `TF` *(needed for lambdas)*.
- Take argument types as *template parameters*.

```
#define IS_VALID_1(type)                                \
    is_valid<type>(                                     \
        [](auto _0) constexpr                           \
            -> decltype IS_VALID_EXPANDER_END
```

- "Bind" type to `is_valid`.
- "Return" partially-formed `constexpr` lambda that expects an expression `e` in `-> decltype(e)`.

```
#define IS_VALID_EXPANDER_END( ... ) (__VA_ARGS__){})
```

- "Take" an expression and "complete" the lambda.

Example expansion: `IS_VALID_1(T)(_0.boop())` :

```
is_valid<T>([](auto _0) constexpr
    -> decltype IS_VALID_EXPANDER_END
```

```
is_valid<T>([](auto _0) constexpr
    -> decltype (_0.boop()){}
```

...

```
if constexpr(is_valid<T>([](auto _0) constexpr
    -> decltype (_0.boop()){}) { /* ... */ }
```

```
#define IS_VALID_EXPANDER_BEGIN(n)                          \
    [](VRM_PP_REPEAT_INC(n, IS_VALID_EXPANDER_MIDDLE, \
        _)) constexpr→decltype IS_VALID_EXPANDER_END

#define IS_VALID_EXPANDER_MIDDLE(idx, _) \
    VRM_PP_COMMA_IF(idx) auto _##idx

#define IS_VALID_EXPANDER_END( ... ) \
    (__VA_ARGS__){})

#define IS_VALID( ... )                                     \
    is_valid<__VA_ARGS__>(IS_VALID_EXPANDER_BEGIN( \
        VRM_PP_ARGCOUNT(__VA_ARGS__))
```

Generalized with some `vrm_pp` preprocessor metaprogramming...

It works! *(on clang)*

https://wandbox.org/permlink/sRbwCBkDm5uH9t4K

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

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

    // ...
}
```

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

    // ...
}
```

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

```
template <typename V, typename T>
auto visit(V&& visitor, T&& variant)
{
    if constexpr(IS_VALID(V, T)
                 (std::visit(_0, _1)))
    {
        return std::visit(visitor, variant);
    }
    else if constexpr(IS_VALID(V, T)
                      (boost::apply_visitor(_0, _1)))
    {
        return boost::apply_visitor(visitor, variant);
    }
}
```

subtle advertisement

Thanks!

- https://vittorioromeo.info
- vittorio.romeo@outlook.com
- https://github.com/SuperV1234
- @supahvee1234

https://github.com/SuperV1234/accu2017