An introduction to C++'s SFINAE concept: compile-time introspection of a class member

Trivia:

As a C++ enthusiast, I usually follow the annual C++ conference cppconf or at least try to keep myself up-to-date with the major events that happen there. One way to catch up, if you can't afford a plane ticket or the ticket, is to follow the youtube channel dedicated to this conference. This year, I was impressed by **Louis Dionne** talk entitled "C++ Metaprogramming: A Paradigm Shift". One feature called **is_valid** that can be found in Louis's Boost.Hana library particulary caught my attention. This genious **is_valid** function heavily rely on an even more "magic" C++ programming technique coined with the term **SFINAE** discovered at the end of the previous century. If this acronym doesn't speak to you, don't be scared, we are going to dive straight in the subject.

Note: for the sake of your sanity and the fact that *errare humanum est*, this article might not be 100% accurate!

Introspection in C++?

Before explaining what is **SFINAE**, let's explore one of its main usage: **introspection**. As you might be aware, C++ doesn't excel when it comes to examine the type or properties of an object at runtime. The best ability provided by default would be RTTI. Not only **RTTI** isn't always available, but it also gives you barely more than the current type of the manipulated object. Dynamic languages or those having **reflection** on the other hand are really convenient in some situations like **serialization**.

For instance, in Python, using reflection, one can do the following:

```
class A(object):
    # Simply overrides the 'object.__str__' method.
    def __str__(self):
        return "I am a A"

class B(object):
    # A custom method for my custom objects that I want to serialize.
```

```
    def serialize(self):
        return "I am a B"

class C(object):
    def __init__(self):
        # Oups! 'serialize' is not a method.
        self.serialize = 0

    def __str__(self):
        return "I am a C"

def serialize(obj):
    # Let's check if obj has an attribute called 'serialize'.
    if hasattr(obj, "serialize"):
        # Let's check if this 'serialize' attribute is a method.
        if hasattr(obj.serialize, "__call__"):
            return obj.serialize()

    # Else we call the __str__ method.
    return str(obj)

a = A()
b = B()
c = C()

print(serialize(a)) # output: I am a A.
print(serialize(b)) # output: I am a B.
print(serialize(c)) # output: I am a C.
```

As you can see, during serialization, it comes pretty handy to be able to check if an object has an attribute and to query the type of this attribute. In our case, it permits us to use the **serialize** method if available and fall back to the more generic method **str** otherwise. Powerful, isn't it? Well, we can do it **in plain C++**!

Here is the **C++14** solution mentionned in **Boost.Hana** documentation, using **is_valid**:

```
#include <boost/hana.hpp>
#include <iostream>
#include <string>

using namespace std;
```

```
namespace hana = boost::hana;

// Check if a type has a serialize method.
auto hasSerialize = hana::is_valid([](auto&& x) -> decltype(x.serialize()) { });

// Serialize any kind of objects.
template <typename T>
std::string serialize(T const& obj) {
    return hana::if_(hasSerialize(obj), // Serialize is selected if available!
                     [](auto& x) { return x.serialize(); },
                     [](auto& x) { return to_string(x); }
    )(obj);
}

// Type A with only a to_string overload.
struct A {};

std::string to_string(const A&)
{
    return "I am a A!";
}

// Type B with a serialize method.
struct B
{
    std::string serialize() const
    {
        return "I am a B!";
    }
};

// Type C with a "wrong" serialize member (not a method) and a to_string overload.
struct C
{
    std::string serialize;
};

std::string to_string(const C&)
{
    return "I am a C!";
```

```
}

int main() {
    A a;
    B b;
    C c;

    std::cout << serialize(a) << std::endl;
    std::cout << serialize(b) << std::endl;
    std::cout << serialize(c) << std::endl;
}
```

As you can see, it only requires a bit more of boilerplate than Python, but not as much as you would expect from a language as complexe as C++. How does it work? Well if you are too lazy to read the rest, here is the simplest answer I can give you: unlike dynamically typed languages, your compiler has access a lot of static type information once fired. It makes sense that we can constraint your compiler to do a bit of work on these types! The next question that comes to your mind is "How to?". Well, right below we are going to explore the various options we have to enslave our favorite compiler for fun and profit! And we will eventually recreate our own **is_valid**.

The old-fashioned C++98-way:

Whether your compiler is a dinosaur, your boss refuses to pay for the latest Visual Studio license or you simply love archeology, this chapter will interest you. It's also interesting for the people stuck between C++11 and C++14. The solution in C++98 relies on 3 key concepts: **overload resolution**, **SFINAE** and the static behavior of **sizeof**.

Overload resolution:

A simple function call like "**f(obj);**"" in **C++** activates a mechanism to figure out which **f** function shoud be called according to the argument **obj**. If a **set** of **f** functions could accept **obj** as an argument, the compiler must choose the most appropriate function, or in other words **resolve** the best **overload**! Here is a good cppreference page explaining the full process: Overload resolution. The rule of thumb in this case is *the compiler picks the candidate function whose parameters match the arguments most closely is the one that is called*. Nothing is better than a good example:

```
void f(std::string s); // int can't be convert into a string.
void f(double d); // int can be implicitly convert into a double, so this version could be
void f(int i); // ... this version using the type int directly is even more close!
```

```
f(1); // Call f(int i);
```

In C++ you also have some sink-hole functions that accept everything. First, function templates accept any kind of parameter (let's say T). But the true black-hole of your compiler, the devil variable vacuum, the oblivion of the forgotten types are the variadic functions. Yes, exactly like the horrible C **printf**.

```
std::string f(...); // Variadic functions are so "untyped" that...
template <typename T> std::string f(const T& t); // ...this templated function got the prec

f(1); // Call the templated function version of f.
```

The fact that function templates are less generic than variadic functions is the first point you must remember!

Note: A **templated function** can actually be more precise than a **normal function**. However, in case of a draw, the **normal function** will have the precedence.

SFINAE:

I am already teasing you with the power for already few paragraphs and here finally comes the explanation of this not so complex acronym. **SFINAE** stands for **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror. In rough terms, a **substitution** is the mechanism that tries to replace the template parameters with the provided types or values. In some cases, if the **substitution** leads to an invalid code, the compiler shouldn't throw a massive amount of errors but simply continue to try the other available **overloads**. The **SFINAE** concept simply guaranties such a "sane" behavior for a "sane" compiler. For instance:

```
/*
 The compiler will try this overload since it's less generic than the variadic.
 T will be replace by int which gives us void f(const int& t, int::iterator* b = nullptr);
 int doesn't have an iterator sub-type, but the compiler doesn't throw a bunch of errors.
 It simply tries the next overload.
*/
template <typename T> void f(const T& t, typename T::iterator* it = nullptr) { }

// The sink-hole.
void f(...) { }

f(1); // Calls void f(...) { }
```

All the expressions won't lead to a **SFINAE**. A broad rule would be to say that all the **substitutions** out of the function/methods **body** are "safes". For a better list, please take a look at this wiki page. For instance, a wrong substitution within a function **body** will lead to a horrible C++ template error:

```
// The compiler will be really unhappy when it will later discover the call to hahahaICrash
template <typename T> void f(T t) { t.hahahaICrash(); }
void f(...) { } // The sink-hole wasn't even considered.

f(1);
```

The operator sizeof:

The **sizeof operator** is really a nice tool! It permits us to returns the size in bytes of a type or an expression at compilation time. **sizeof** is really interesting as it accurately evaluates an expression as precisely as if it were compiled. One can for instance do:

```
typedef char type_test[42];
type_test& f();

// In the following lines f won't even be truly called but we can still access to the size
// Thanks to the "fake evaluation" of the sizeof operator.
char arrayTest[sizeof(f())];
std::cout << sizeof(f()) << std::endl; // Output 42.
```

But wait! If we can manipulate some compile-time integers, couldn't we do some compile-time comparison? The answer is: absolutely yes, my dear reader! Here we are:

```
typedef char yes; // Size: 1 byte.
typedef yes no[2]; // Size: 2 bytes.

// Two functions using our type with different size.
yes& f1();
no& f2();

std::cout << (sizeof(f1()) == sizeof(f2())) << std::endl; // Output 0.
std::cout << (sizeof(f1()) == sizeof(f1())) << std::endl; // Output 1.
```

Combining everything:

Now we have all the tools to create a solution to check the existence of a method within a type at compile time. You might even have already figured it out most of it by yourself. So let's create it:

```
template <class T> struct hasSerialize
{
    // For the compile time comparison.
    typedef char yes[1];
    typedef yes no[2];

    // This helper struct permits us to check that serialize is truly a method.
    // The second argument must be of the type of the first.
    // For instance reallyHas<int, 10> would be substituted by reallyHas<int, int 10> and w
    // reallyHas<int, &C::serialize> would be substituted by reallyHas<int, int &C::seriali
    // Note: It only works with integral constants and pointers (so function pointers work)
    // In our case we check that &C::serialize has the same signature as the first argument
    // reallyHas<std::string (C::*)(), &C::serialize> should be substituted by
    // reallyHas<std::string (C::*)(), std::string (C::*)() &C::serialize> and work!
    template <typename U, U u> struct reallyHas;

    // Two overloads for yes: one for the signature of a normal method, one is for the sign
    // We accept a pointer to our helper struct, in order to avoid to instantiate a real in
    // std::string (C::*)() is function pointer declaration.
    template <typename C> static yes& test(reallyHas<std::string (C::*)(), &C::serialize>*
    template <typename C> static yes& test(reallyHas<std::string (C::*)() const, &C::serial

    // The famous C++ sink-hole.
    // Note that sink-hole must be templated too as we are testing test<T>(0).
    // If the method serialize isn't available, we will end up in this method.
    template <typename> static no& test(...) { /* dark matter */ }

    // The constant used as a return value for the test.
    // The test is actually done here, thanks to the sizeof compile-time evaluation.
    static const bool value = sizeof(test<T>(0)) == sizeof(yes);
};

// Using the struct A, B, C defined in the previous hasSerialize example.
std::cout << hasSerialize<A>::value << std::endl;
std::cout << hasSerialize<B>::value << std::endl;
std::cout << hasSerialize<C>::value << std::endl;
```

The **reallyHas** struct is kinda tricky but necessary to ensure that serialize is a method and not a simple member of the type. You can do a lot of test on a type using variants of this solution (test a member, a sub-type...) and I suggest you to google a bit more about **SFINAE**

tricks. Note: if you truly want a pure compile-time constant and avoid some errors on old compilers, you can replace the last **value** evaluation by: "**enum { value = sizeof(test(o)) == sizeof(yes) };**".

You might also wonder why it doesn't work with **inheritence**. **Inheritence** in C++ and **dynamic polymorphism** is a concept available at runtime, or in other words, a data that the compiler won't have and can't guess! However, compile time type inspection is much more efficient (o impact at runtime) and almost as powerful as if it were at runtime. For instance:

```
// Using the previous A struct and hasSerialize helper.

struct D : A
{
    std::string serialize() const
    {
        return "I am a D!";
    }
};

template <class T> bool testHasSerialize(const T& /*t*/) { return hasSerialize<T>::value; }

D d;
A& a = d; // Here we lost the type of d at compile time.
std::cout << testHasSerialize(d) << std::endl; // Output 1.
std::cout << testHasSerialize(a) << std::endl; // Output 0.
```

Last but no least, our test cover the main cases but not the tricky ones like a Functor:

```
struct E
{
    struct Functor
    {
        std::string operator()()
        {
            return "I am a E!";
        }
    };

    Functor serialize;
};

E e;
```

```
std::cout << e.serialize() << std::endl; // Succefully call the functor.
std::cout << testHasSerialize(e) << std::endl; // Output 0.
```

The trade-off for a full coverage would be the readability. As you will see, C++11 shines in that domain!

Time to use our genius idea:

Now you would think that it will be super easy to use our **hasSerialize** to create a **serialize** function! Okay let's try it:

```
template <class T> std::string serialize(const T& obj)
{
    if (hasSerialize<T>::value) {
        return obj.serialize(); // error: no member named 'serialize' in 'A'.
    } else {
        return to_string(obj);
    }
}

A a;
serialize(a);
```

It might be hard to accept, but the error raised by your compiler is absolutely normal! If you consider the code that you will obtain after substitution and compile-time evaluation:

```
std::string serialize(const A& obj)
{
    if (0) { // Dead branching, but the compiler will still consider it!
        return obj.serialize(); // error: no member named 'serialize' in 'A'.
    } else {
        return to_string(obj);
    }
}
```

Your compiler is really a good guy and won't drop any dead-branch, and **obj** must therefore have both a **serialize method** and a **to_string overload** in this case. The solution consists in spliting the serialize function into two different functions: one where we solely use **obj.serialize()** and one where we use **to_string** according to **obj's type**. We come back to an earlier problem that we already solved, how to split according to a type? **SFINAE**, for sure! At that point we could re-work our **hasSerialize** function into a **serialize** function and make it return a **std::string** instead of compile time **boolean**. But we won't do it that way! It's cleaner to separate the **hasSerialize** test from its usage **serialize**.

We need to find a clever **SFINAE** solution on the signature of "**template <class T> std::string serialize(const T& obj)**". I bring you the last piece of the puzzle called **enable_if**.

```
template<bool B, class T = void> // Default template version.
struct enable_if {}; // This struct doesn't define "type" and the substitution will fail if

template<class T> // A specialisation used if the expression is true.
struct enable_if<true, T> { typedef T type; }; // This struct do have a "type" and won't fa

// Usage:
enable_if<true, int>::type t1; // Compiler happy. t's type is int.
enable_if<hasSerialize<B>::value, int>::type t2; // Compiler happy. t's type is int.

enable_if<false, int>::type t3; // Compiler unhappy. no type named 'type' in 'enable_if<fal
enable_if<hasSerialize<A>::value, int>::type t4; // no type named 'type' in 'enable_if<fals
```

As you can see, we can trigger a substitution failure according to a compile time expression with **enable_if**. Now we can use this failure on the "**template <class T> std::string serialize(const T& obj)**" signature to dispatch to the right version. Finally, we have the true solution of our problem:

```
template <class T> typename enable_if<hasSerialize<T>::value, std::string>::type serialize(
{
    return obj.serialize();
}

template <class T> typename enable_if<!hasSerialize<T>::value, std::string>::type serialize
{
    return to_string(obj);
}

A a;
B b;
C c;

// The following lines work like a charm!
std::cout << serialize(a) << std::endl;
std::cout << serialize(b) << std::endl;
std::cout << serialize(c) << std::endl;
```

Two details worth being noted! Firstly we use **enable_if** on the return type, in order to keep the paramater deduction, otherwise we would have to specify the type explicitely "**serialize<A>(a)**". Second, even the version using **to_string** must use the **enable_if**, otherwise **serialize(b)** would have two potential overloads available and raise an ambiguity. If you want to check the full code of this C++98 version, here is a gist. Life is much easier in C++11, so let's see the beauty of this new standard!

Note: it's also important to know that this code creates a **SFINAE** on an expression ("**&C::serialize**"). Whilst this feature wasn't required by the **C++98** standard, it was already in use depending on your compiler. It trully became a safe choice in **C++11**.

When C++11 came to our help:

After the great century leap year in 2000, people were fairly optimistic about the coming years. Some even decided to design a new standard for the next generation of **C++** coders like me! Not only this standard would ease **TMP** headaches (**T**emplate **M**eta **P**rogramming side-effects), but it would be available in the first decade, hence its code-name **C++0x**. Well, the standard sadly came the next decade (2011 ==> **C++11**), but it brought a lot of features interesting for the purpose of this article. Let's review them!

decltype, declval, auto & co:

Do you remember that the **sizeof operator** does a "fake evaluation" of the expression that you pass to it, and return gives you the size of the type of the expression? Well **C++11** adds a new operator called **decltype**. decltype gives you the type of the of the expression it will evaluate. As I am kind, I won't let you google an example and give it to you directly:

```
B b;
decltype(b.serialize()) test = "test"; // Evaluate b.serialize(), which is typed as std::st
// Equivalent to std::string test = "test";
```

declval is an utility that gives you a "fake reference" to an object of a type that couldn't be easily construct. **declval** is really handy for our **SFINAE** constructions. **cppreference** example is really straightforward, so here is a copy:

```
struct Default {
    int foo() const {return 1;}
};

struct NonDefault {
    NonDefault(const NonDefault&) {}
    int foo() const {return 1;}
```

```
};

int main()
{
    decltype(Default().foo()) n1 = 1; // int n1
//  decltype(NonDefault().foo()) n2 = n1; // error: no default constructor
    decltype(std::declval<NonDefault>().foo()) n2 = n1; // int n2
    std::cout << "n2 = " << n2 << '\n';
}
```

The **auto specifier** *specifies that the type of the variable that is being declared will be automatically deduced*. auto is equivalent of **var** in C#. **auto** in **C++11** has also a less famous but nonetheless usage for function declaration. Here is a good example:

```
bool f();
auto test = f(); // Famous usage, auto deduced that test is a boolean, hurray!



//                              vvv t wasn't declare at that point, it will be after as a pa
template <typename T> decltype(t.serialize()) g(const T& t) {   } // Compilation error

// Less famous usage:
//                    vvv auto delayed the return type specification!
//                    vvv                vvv the return type is specified here and use t!
template <typename T> auto g(const T& t) -> decltype(t.serialize()) {   } // No compilation
```

As you can see, **auto** permits to use the trailing return type syntax and use **decltype** coupled with an expression involving one of the function argument. Does it means that we can use it to test the existence of **serialize** with a SFINAE? Yes Dr. Watson! **decltype** will shine really soon, you will have to wait for the **C++14** for this tricky **auto** usage (but since it's a C++11 feature, it ends up here).

constexpr:

C++11 also came with a new way to do compile-time computations! The new keyword **constexpr** is a hint for your compiler, meaning that this expression is constant and could be evaluate directly at compile time. In C++11, constexpr has a lot of rules and only a small subset of VIEs (Very Important Expression) expressions can be used (no loops...)! We still have enough for creating a compile-time factorial function:

```
constexpr int factorial(int n)
{
    return n ⩽ 1? 1 : (n * factorial(n - 1));
}

int i = factorial(5); // Call to a constexpr function.
// Will be replace by a good compiler by:
// int i = 120;
```

constexpr increased the usage of **std::true_type** & **std::false_type** from the STL. As their name suggest, these types encapsulate a constexpr boolean "true" and a constrexpr boolean "false". Their most important property is that a class or a struct can inherit from them. For instance:

```
struct testStruct : std::true_type { }; // Inherit from the true type.

constexpr bool testVar = testStruct(); // Generate a compile-time testStruct.
bool test = testStruct::value; // Equivalent to: test = true;
test = testVar; // true_type has a constexpr converter operator, equivalent to: test = true
```

Blending time:

First solution:

In cooking, a good recipe requires to mix all the best ingredients in the right proportions. If you don't want to have a spaghetti code dating from 1998 for dinner, let's revisit our C++98 **hasSerialize** and **serialize** functions with "fresh" ingredients from 2011. Let's start by removing the rotting **reallyHas** trick with a tasty **decltype** and bake a bit of **constexpr** instead of **sizeof**. After 15min in the oven (or fighting with a new headache), you will obtain:

```
template <class T> struct hasSerialize
{
    // We test if the type has serialize using decltype and declval.
    template <typename C> static constexpr decltype(std::declval<C>().serialize(), bool())
    {
        // We can return values, thanks to constexpr instead of playing with sizeof.
        return true;
    }

    template <typename C> static constexpr bool test(...)
    {
        return false;
```

```
    }

    // int is used to give the precedence!
    static constexpr bool value = test<T>(int());
};
```

You might be a bit puzzled by my usage of **decltype**. The C++ comma operator "**,**" can create a chain of multiple expressions. In **decltype**, all the expressions will be evaluated, but only the last expression will be considered for the type. The **serialize** doesn't need any changes, minus the fact that the **enable_if** function is now provided in the **STL**. For your tests, here is a gist.

Second solution:

Another C++11 solution described in **Boost.Hanna** documentation and using **std::true_type** and **std::false_type**, would be this one:

```
// Primary template, inherit from std::false_type.
// ::value will return false.
// Note: the second unused template parameter is set to default as std::string!!!
template <typename T, typename = std::string>
struct hasSerialize
        : std::false_type
{

};

// Partial template specialisation, inherit from std::true_type.
// ::value will return true.
template <typename T>
struct hasSerialize<T, decltype(std::declval<T>().serialize())>
        : std::true_type
{

};
```

This solution is, in my own opinion, more sneaky! It relies on a not-so-famous-property of default template parameters. But if your soul is already (stack-)corrupted, you may be aware that the **default parameters** are propagated in the **specialisations**. So when we use **hasSerialize<OurType>::value**, the default parameter comes into play and we are actually looking for **hasSerialize<OurType, std::string>::value** both on the **primary template** and the **specialisation**. In the meantime, the **substitution** and the evaluation of **decltype** are

processed and our **specialisation** has the signature **hasSerialize<OurType, std::string>** if **OurType** has a **serialize** method that returns a **std::string**, otherwise the substitution fails. The **specialisation** has therefore the precedence in the good cases. One will be able to use the std::void_t C++17 helper in these cases. Anyway, here is a gist you can play with!

I told you that this second solution hides a lot of complexity, and we still have a lot of C++11 features unexploited like **nullptr**, **lambda**, **r-values**. No worries, we are going to use some of them in **C++14**!

The supremacy of C++14:

According to the Gregorian calendar in the upper-right corner of my XFCE environment, we are in 2015! I can turn on the **C++14** compilation flag on my favorite compiler safely, isn't it? Well, I can with **clang** (is **MSVC** using a maya calendar?). Once again, let's explore the new features, and use them to build something wonderful! We will even recreate an **is_valid**, like I promised at the beggining of this article.

auto & lambdas:

Return type inference:

Some cool features in **C++14** come from the relaxed usage of the **auto** keyword (the one used for type inference).

Now, **auto** can be used on the return type of a function or a method. For instance:

```
auto myFunction() // Automagically figures out that myFunction returns ints.
{
    return int();
}
```

It works as long as the type is easily "guessable" by the compiler. We are coding in C++ after all, not OCaml!

A feature for functional lovers:

C++11 introduced lambdas. A lambda has the following syntax:

```
[capture-list](params) -> non-mandatory-return-type { ...body... }
```

A useful example in our case would be:

```
auto l1 = [](B& b) { return b.serialize(); }; // Return type figured-out by the return stat
auto l3 = [](B& b) -> std::string { return b.serialize(); }; // Fixed return type.
```

```
auto l2 = [](B& b) -> decltype(b.serialize()) { return b.serialize(); }; // Return type dep

std::cout << l1(b) << std::endl; // Output: I am a B!
std::cout << l2(b) << std::endl; // Output: I am a B!
std::cout << l3(b) << std::endl; // Output: I am a B!
```

C++14 brings a small change to the **lambdas** but with a big impact! **Lambdas** accept **auto parameters**: the parameter type is deduced according the argument. **Lambdas** are implemented as an object having an newly created **unnamed type**, also called **closure type**. If a **lambda** has some **auto parameters**, its "Functor operator" **operator()** will be simply templated. Let's take a look:

```
// ***** Simple lambda unamed type *****
auto l4 = [](int a, int b) { return a + b; };
std::cout << l4(4, 5) << std::endl; // Output 9.

// Equivalent to:
struct l4UnamedType
{
    int operator()(int a, int b) const
    {
        return a + b;
    }
};

l4UnamedType l4Equivalent = l4UnamedType();
std::cout << l4Equivalent(4, 5) << std::endl; // Output 9 too.



// ***** auto parameters lambda unnamed type *****

// b's type is automagically deduced!
auto l5 = [](auto& t) -> decltype(t.serialize()) { return t.serialize(); };

std::cout << l5(b) << std::endl; // Output: I am a B!
std::cout << l5(a) << std::endl; // Error: no member named 'serialize' in 'A'.

// Equivalent to:
struct l5UnamedType
{
```

```
    template <typename T> auto operator()(T& t) const -> decltype(t.serialize()) // /!\ Thi
    {
        return t.serialize();
    }
};

l5UnamedType l5Equivalent = l5UnamedType();

std::cout << l5Equivalent(b) << std::endl; // Output: I am a B!
std::cout << l5Equivalent(a) << std::endl; // Error: no member named 'serialize' in 'A'.
```

More than the **lambda** itself, we are interested by the generated **unnamed type**: its lambda **operator()** can be used as a SFINAE! And as you can see, writing a **lambda** is less cumbersome than writing the equivalent type. It should remind you the beggining of my initial solution:

```
// Check if a type has a serialize method.
auto hasSerialize = hana::is_valid([](auto&& x) -> decltype(x.serialize()) { });
```

And the good new is that we have everything to recreate **is_valid**, right now!

The making-of a valid is_valid:

Now that we have a really stylish manner to generate a **unnamed types** with potential **SFINAE** properties using **lambdas**, we need to figure out how to use them! As you can see, **hana::is_valid** is a function that takes our lambda as a parameter and return a type. We will call the type returned by **is_valid** the **container**. The **container** will be in charge to keep the lambda's **unnamed type** for a later usage. Let's start by writing the **is_valid** function and its the **containter**:

```
template <typename UnnamedType> struct container
{
    // Remembers UnnamedType.
};

template <typename UnnamedType> constexpr auto is_valid(const UnnamedType& t)
{
    // We used auto for the return type: it will be deduced here.
    return container<UnnamedType>();
}
```

```
auto test = is_valid([](const auto& t) -> decltype(t.serialize()) {})
// Now 'test' remembers the type of the lambda and the signature of its operator()!
```

The next step consists at extending **container** with the operator **operator()** such as we can call it with an argument. This argument type will be tested against the **UnnamedType**! In order to do a test on the argument type, we can use once again a SFINAE on a reacreated 'UnnamedType' object! It gives us this solution:

```
template <typename UnnamedType> struct container
{
// Let's put the test in private.
private:
    // We use std::declval to 'recreate' an object of 'UnnamedType'.
    // We use std::declval to also 'recreate' an object of type 'Param'.
    // We can use both of these recreated objects to test the validity!
    template <typename Param> constexpr auto testValidity(int /* unused */)
    -> decltype(std::declval<UnnamedType>()(std::declval<Param>()), std::true_type())
    {
        // If substitution didn't fail, we can return a true_type.
        return std::true_type();
    }

    template <typename Param> constexpr std::false_type testValidity(...)
    {
        // Our sink-hole returns a false_type.
        return std::false_type();
    }

public:
    // A public operator() that accept the argument we wish to test onto the UnnamedType.
    // Notice that the return type is automatic!
    template <typename Param> constexpr auto operator()(const Param& p)
    {
        // The argument is forwarded to one of the two overloads.
        // The SFINAE on the 'true_type' will come into play to dispatch.
        // Once again, we use the int for the precedence.
        return testValidity<Param>(int());
    }
};

template <typename UnnamedType> constexpr auto is_valid(const UnnamedType& t)
```

```
{
    // We used auto for the return type: it will be deduced here.
    return container<UnnamedType>();
}

// Check if a type has a serialize method.
auto hasSerialize = is_valid([](auto&& x) -> decltype(x.serialize()) { });
```

If you are a bit lost at that point, I suggest you take your time and re-read all the previous example. You have all the weapons you need, now fight **C++**!

Our **hasSerialize** now takes an argument, we therefore need some changes for our serialize function. We can simply post-pone the return type using **auto** and use the argument in a **decltype** as we learn. Which gives us:

```
// Notice how I simply swapped the return type on the right?
template <class T> auto serialize(T& obj)
-> typename std::enable_if<decltype(hasSerialize(obj))::value, std::string>::type
{
    return obj.serialize();
}

template <class T> auto serialize(T& obj)
-> typename std::enable_if<!decltype(hasSerialize(obj))::value, std::string>::type
{
    return to_string(obj);
}
```

FINALLY!!! We do have a working **is_valid** and we could use it for serialization! If I were as vicious as my SFINAE tricks, I would let you copy each code pieces to recreate a fully working solution. But today, Halloween's spirit is with me and here is gist. Hey, hey! Don't close this article so fast! If you are true a warrior, you can read the last part!

For the fun:

There are few things I didn't tell you, on purpose. This article would otherwise be twice longer, I fear. I highly suggest you to google a bit more about what I am going to speak about.

- Firstly, if you wish to have a solution that works with the **Boost.Hana** static **if_**, you need to change the return type of our **testValidity** methods by Hana's equivalents, like the following:

```
template <typename Param> constexpr auto test_validity(int /* unused */)
→ decltype(std::declval<UnnamedType>()(std::declval<Param>()), boost::hana::true_c)
{
    // If substitution didn't fail, we can return a true_type.
    return boost::hana::true_c;
}

template <typename Param> constexpr decltype(boost::hana::false_c) test_validity(...)
{
    // Our sink-hole returns a false_type.
    return boost::hana::false_c;
}
```

The static **if_** implementation is really interesting, but at least as hard as our **is_valid** problem solved in this article. I might dedicate another article about it, one day!

- Did you noticed that we only check one argument at a time? Couldn't we do something like:

```
auto test = is_valid([](auto&& a, auto&& b) → decltype(a.serialize(), b.serialize()) {
A a;
B b;

std::cout << test(a, b) << std::endl;
```

Actually we can, using some parameter packs. Here is the solution:

```
template <typename UnnamedType> struct container
{
// Let's put the test in private.
private:
    // We use std::declval to 'recreate' an object of 'UnnamedType'.
    // We use std::declval to also 'recreate' an object of type 'Param'.
    // We can use both of these recreated objects to test the validity!
    template <typename... Params> constexpr auto test_validity(int /* unused */)
    → decltype(std::declval<UnnamedType>()(std::declval<Params>()...), std::true_type()
    {
        // If substitution didn't fail, we can return a true_type.
        return std::true_type();
    }

    template <typename... Params> constexpr std::false_type test_validity(...)
```

```
    {
        // Our sink-hole returns a false_type.
        return std::false_type();
    }

public:
    // A public operator() that accept the argument we wish to test onto the UnnamedType
    // Notice that the return type is automatic!
    template <typename... Params> constexpr auto operator()(Params&& ...)
    {
        // The argument is forwarded to one of the two overloads.
        // The SFINAE on the 'true_type' will come into play to dispatch.
        return test_validity<Params...>(int());
    }
};

template <typename UnnamedType> constexpr auto is_valid(UnnamedType&& t)
{
    // We used auto for the return type: it will be deduced here.
    return container<UnnamedType>();
}
```

- This code is working even if my types are incomplete, for instance a forward declaration, or a normal declaration but with a missing definition. What can I do? Well, you can insert a check on the size of your type either in the **SFINAE** construction or before calling it: "**static_assert(sizeof(T), "type is incomplete.");**".

- Finally, why are using the notation "**&&**" for the **lambdas** parameters? Well, these are called **forwarding references**. It's a really complex topic, and if you are interested, here is good article about it. You need to use "**auto&&**" due to the way **declval** is working in our **is_valid** implementation!

Notes:

This is my first serious article about **C++** on the web and I hope you enjoyed it! I would be glad if you have any suggestions or questions and that you wish to share with me in the commentaries.

Anyway, thanks to Naav and Superboum for rereading this article and theirs suggestions. Few suggestions were also provided by the reddit community or in the commentaries of this

post, thanks a lot guys!

■