Check if a class has a member function of a given signature

Asked 14 years, 3 months ago Modified 3 months ago Viewed 115k times

168

I'm asking for a template trick to detect if a class has a specific member function of a given signature.

The problem is similar to the one cited here http://www.gotw.ca/gotw/071.htm but not the same: in the item of Sutter's book he answered to the question that a class C MUST PROVIDE a member function with a particular signature, else the program won't compile. In my problem I need to do something if a class has that function, else do "something else".

A similar problem was faced by boost::serialization but I don't like the solution they adopted: a template function that invokes by default a free function (that you have to define) with a particular signature unless you define a particular member function (in their case "serialize" that takes 2 parameters of a given type) with a particular signature, else a compile error will happens. That is to implement both intrusive and non-intrusive serialization.

I don't like that solution for two reasons:

1. To be non intrusive you must override the global "serialize" function that is in boost::serialization namespace, so you have IN YOUR CLIENT CODE to open namespace boost and namespace serialization!
2. The stack to resolve that mess was 10 to 12 function invocations.

I need to define a custom behavior for classes that has not that member function, and my entities are inside different namespaces (and I don't want to override a global function defined in one namespace while I'm in another one)

Can you give me a hint to solve this puzzle?

c++ c++11 templates sfinae

Share Improve this question Follow

edited Nov 11, 2018 at 12:24
Stephen Kennedy
19.8k 22 93 106

asked Sep 17, 2008 at 20:36
ugasoft
3,700 7 27 23

2 Similar question: stackoverflow.com/questions/257288 – Johannes Schaub - litb Aug 25, 2009 at 20:02

@R.MartinhoFernandes What kind of answer are you looking for? This answer by Mike Kinghan goes quite in depth and is using C++11 stuff. – jrok May 29, 2013 at 17:04

@R.MartinhoFernandes Maybe this is the modern version you are looking for? – Daniel Frey May 29, 2013 at 19:36

17 Answers

Sorted by: Highest score (default)

161

+25

Here's a possible implementation relying on C++11 features. It correctly detects the function even if it's inherited (unlike the solution in the accepted answer, as Mike Kinghan observes in his answer).

The function this snippet tests for is called `serialize`:

```
#include <type_traits>

// Primary template with a static assertion
// for a meaningful error message
// if it ever gets instantiated.
// We could leave it undefined if we didn't care.

template<typename, typename T>
struct has_serialize {
    static_assert(
        std::integral_constant<T, false>::value,
        "Second template parameter needs to be of function type.");
};

// specialization that does the checking

template<typename C, typename Ret, typename... Args>
struct has_serialize<C, Ret(Args...)> {
private:
    template<typename T>
    static constexpr auto check(T*)
    -> typename
        std::is_same<
            decltype( std::declval<T>().serialize( std::declval<Args>
()... ) ),
            Ret    //
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        >::type;  // attempt to call it and see if the return type is
correct

    template<typename>
```

```
    static constexpr std::false_type check(...);

    typedef decltype(check<C>(0)) type;

public:
    static constexpr bool value = type::value;
};
```

Usage:

```
struct X {
     int serialize(const std::string&) { return 42; }
};

struct Y : X {};

std::cout << has_serialize<Y, int(const std::string&)>::value; //
will print 1
```

Share Improve this answer Follow

edited May 23, 2017 at 11:33
Community Bot
1 1

answered May 29, 2013 at 21:33
jrok
53.7k 9 106 140

Does this work if Y does not have a method called "serialize"? I don't see how it would return a false value if the method "serialize" didn't exist. – Talia Feb 20, 2014 at 3:47

1 @Collin in that case substitution of template parameter fails for first overload of check and it's discarded from overload set. It falls back to the second one that returns false_type. This is not a compiler error because SFINAE principle. – jrok Feb 20, 2014 at 7:57

Oh, and the arrow is from the new return value syntax in C++11, described here: cprogramming.com/c++11/... – Talia Feb 22, 2014 at 0:24

@jrok is there a way to add an extra template argument to this struct so you can also pass to it the function name you want to check, (eg. serialize) – elios264 Oct 16, 2015 at 15:19

1 @elios264 There isn't. You can use a macro to write a template for each function you want to check for. – jrok Oct 16, 2015 at 16:37

1 Any particular reason why the argument for check is of type T* rather than T or T&? – shibumi Nov 9, 2017 at 5:28

I've figured it out. Without a pointer type it's hard to pass the template argument. – shibumi Nov 9, 2017 at 5:42

1 But what if the `serialize` itself accepts a template. Is there a way to test for `serialize`

existence without typing the exact type? – Hi-Angel May 9, 2018 at 12:40

I think that it can be simplified like stackoverflow.com/a/60603142/3676427 – Isaac Pascual Mar 9, 2020 at 14:57

106

I'm not sure if I understand you correctly, but you may exploit SFINAE to detect function presence at compile-time. Example from my code (tests if class has member function size_t used_memory() const).

```
template<typename T>
struct HasUsedMemoryMethod
{
    template<typename U, size_t (U::*)() const> struct SFINAE {};
    template<typename U> static char Test(SFINAE<U,
&U::used_memory>*);
    template<typename U> static int Test(...);
    static const bool Has = sizeof(Test<T>(0)) == sizeof(char);
};

template<typename TMap>
void ReportMemUsage(const TMap& m, std::true_type)
{
        // We may call used_memory() on m here.
}
template<typename TMap>
void ReportMemUsage(const TMap&, std::false_type)
{
}
template<typename TMap>
void ReportMemUsage(const TMap& m)
{
    ReportMemUsage(m,
        std::integral_constant<bool, HasUsedMemoryMethod<TMap>::Has>
());
}
```

Share Improve this answer Follow

edited May 1, 2014 at 1:57
Oktalist
14k 3 46 63

answered Sep 17, 2008 at 21:27
yrp
4,535 2 24 10

3 The example is missing the definition of 'int_to_type'. Obviously it doesn't add to the answer, but it does mean that people can see your code in action after a quick cut & paste. – Richard Corden Sep 18, 2008 at 17:50

2 A simple definition of int_to_type could be: 'template <int N> struct int_to_type {};'. Many implementations keep the paramter N value either in an enum or else in a static integer constant (template <int N> struct int_to_type { enum { value = N }; }; / template <int N> struct int_to_type { static const int value = N; }) – David Rodríguez - dribeas Sep 3, 2009 at 22:04

2 Simply take boost::integral_constant instead of int_to_type. – Vadim Ferderer Sep 28, 2009 at 5:15

2 @JohanLundberg It's a pointer-to-(non-static-)member-function. For example, `size_t(std::vector::*p)() = &std::vector::size;` . – Kuba hasn't forgotten Monica Jul 27, 2015 at 14:28

1 Is it possible to check pass function name as second template parameter, i.e. `HasMethod<Type,method_signature>` ? – kyb Aug 3, 2017 at 9:00

41

The accepted answer to this question of compiletime member-function introspection, although it is justly popular, has a snag which can be observed in the following program:

```
#include <type_traits>
#include <iostream>
#include <memory>

/*  Here we apply the accepted answer's technique to probe for the
    the existence of `E T::operator*() const`
*/
template<typename T, typename E>
struct has_const_reference_op
{
    template<typename U, E (U::*)() const> struct SFINAE {};
    template<typename U> static char Test(SFINAE<U, &U::operator*>*);
    template<typename U> static int Test(...);
    static const bool value = sizeof(Test<T>(0)) == sizeof(char);
};

using namespace std;

/* Here we test the `std::` smart pointer templates, including the
    deprecated `auto_ptr<T>`, to determine in each case whether
    T = (the template instantiated for `int`) provides
    `int & T::operator*() const` - which all of them in fact do.
*/
int main(void)
{
    cout << has_const_reference_op<auto_ptr<int>,int &>::value;
    cout << has_const_reference_op<unique_ptr<int>,int &>::value;
    cout << has_const_reference_op<shared_ptr<int>,int &>::value <<
```

```
endl;
    return 0;
}
```

Built with GCC 4.6.3, the program outputs `110` - informing us that `T = std::shared_ptr<int>` does *not* provide `int & T::operator*() const`.

If you are not already wise to this gotcha, then a look at of the definition of `std::shared_ptr<T>` in the header `<memory>` will shed light. In that implementation, `std::shared_ptr<T>` is derived from a base class from which it inherits `operator*() const`. So the template instantiation `SFINAE<U, &U::operator*>` that constitutes "finding" the operator for `U = std::shared_ptr<T>` will not happen, because `std::shared_ptr<T>` has no `operator*()` in its own right and template instantiation does not "do inheritance".

This snag does not affect the well-known SFINAE approach, using "The sizeof() Trick", for detecting merely whether `T` has some member function `mf` (see e.g. [this answer] and comments). But establishing that `T::mf` exists is often (usually?) not good enough: you may also need to establish that it has a desired signature. That is where the illustrated technique scores. The pointerized variant of the desired signature is inscribed in a parameter of a template type that must be satisfied by `&T::mf` for the SFINAE probe to succeed. But this template instantiating technique gives the wrong answer when `T::mf` is inherited.

A safe SFINAE technique for compiletime introspection of `T::mf` must avoid the use of `&T::mf` within a template argument to instantiate a type upon which SFINAE function template resolution depends. Instead, SFINAE template function resolution can depend only upon exactly pertinent type declarations used as argument types of the overloaded SFINAE probe function.

By way of an answer to the question that abides by this constraint I'll illustrate for compiletime detection of `E T::operator*() const`, for arbitrary `T` and `E`. The same pattern will apply *mutatis mutandis* to probe for any other member method signature.

```
#include <type_traits>

/*! The template `has_const_reference_op<T,E>` exports a
    boolean constant `value that is true iff `T` provides
    `E T::operator*() const`
*/
template< typename T, typename E>
struct has_const_reference_op
{
    /* SFINAE operator-has-correct-sig :) */
    template<typename A>
    static std::true_type test(E (A::*)() const) {
        return std::true_type();
    }
```

```
    /* SFINAE operator-exists :) */
    template <typename A>
    static decltype(test(&A::operator*))
    test(decltype(&A::operator*),void *) {
        /* Operator exists. What about sig? */
        typedef decltype(test(&A::operator*)) return_type;
        return return_type();
    }

    /* SFINAE game over :( */
    template<typename A>
    static std::false_type test(...) {
        return std::false_type();
    }

    /* This will be either `std::true_type` or `std::false_type` */
    typedef decltype(test<T>(0,0)) type;

    static const bool value = type::value; /* Which is it? */
};
```

In this solution, the overloaded SFINAE probe function `test()` is "invoked recursively". (Of course it isn't actually invoked at all; it merely has the return types of hypothetical invocations resolved by the compiler.)

We need to probe for at least one and at most two points of information:

- Does `T::operator*()` exist at all? If not, we're done.
- Given that `T::operator*()` exists, is its signature `E T::operator*() const` ?

We get the answers by evaluating the return type of a single call to `test(0,0)` . That's done by:

```
    typedef decltype(test<T>(0,0)) type;
```

This call might be resolved to the `/* SFINAE operator-exists :) */` overload of `test()` , or it might resolve to the `/* SFINAE game over :( */` overload. It can't resolve to the `/* SFINAE operator-has-correct-sig :) */` overload, because that one expects just one argument and we are passing two.

Why are we passing two? Simply to force the resolution to exclude `/* SFINAE operator-has-correct-sig :) */` . The second argument has no other signifance.

This call to `test(0,0)` will resolve to `/* SFINAE operator-exists :) */` just in case the first argument 0 satifies the first parameter type of that overload, which is `decltype(&A::operator*)` , with `A = T` . 0 will satisfy that type just in case `T::operator*` exists.

Let's suppose the compiler say's Yes to that. Then it's going with `/* SFINAE operator-exists :) */` and it needs to determine the return type of the function call, which in that case is `decltype(test(&A::operator*))` - the return type of yet another call to `test()`.

This time, we're passing just one argument, `&A::operator*`, which we now know exists, or we wouldn't be here. A call to `test(&A::operator*)` might resolve either to `/* SFINAE operator-has-correct-sig :) */` or again to might resolve to `/* SFINAE game over :( */`. The call will match `/* SFINAE operator-has-correct-sig :) */` just in case `&A::operator*` satisfies the single parameter type of that overload, which is `E (A::*)() const`, with `A = T`.

The compiler will say Yes here if `T::operator*` has that desired signature, and then again has to evaluate the return type of the overload. No more "recursions" now: it is `std::true_type`.

If the compiler does not choose `/* SFINAE operator-exists :) */` for the call `test(0,0)` or does not choose `/* SFINAE operator-has-correct-sig :) */` for the call `test(&A::operator*)`, then in either case it goes with `/* SFINAE game over :( */` and the final return type is `std::false_type`.

Here is a test program that shows the template producing the expected answers in varied sample of cases (GCC 4.6.3 again).

```
// To test
struct empty{};

// To test
struct int_ref
{
    int & operator*() const {
        return *_pint;
    }
    int & foo() const {
        return *_pint;
    }
    int * _pint;
};

// To test
struct sub_int_ref : int_ref{};

// To test
template<typename E>
struct ee_ref
{
    E & operator*() {
        return *_pe;
    }
```

```
    E & foo() const {
        return *_pe;
    }
    E * _pe;
};

// To test
struct sub_ee_ref : ee_ref<char>{};

using namespace std;

#include <iostream>
#include <memory>
#include <vector>

int main(void)
{
    cout << "Expect Yes" << endl;
    cout << has_const_reference_op<auto_ptr<int>,int &>::value;
    cout << has_const_reference_op<unique_ptr<int>,int &>::value;
    cout << has_const_reference_op<shared_ptr<int>,int &>::value;
    cout << has_const_reference_op<std::vector<int>::iterator,int
&>::value;
    cout << has_const_reference_op<std::vector<int>::const_iterator,
            int const &>::value;
    cout << has_const_reference_op<int_ref,int &>::value;
    cout << has_const_reference_op<sub_int_ref,int &>::value  <<
endl;
    cout << "Expect No" << endl;
    cout << has_const_reference_op<int *,int &>::value;
    cout << has_const_reference_op<unique_ptr<int>,char &>::value;
    cout << has_const_reference_op<unique_ptr<int>,int const
&>::value;
    cout << has_const_reference_op<unique_ptr<int>,int>::value;
    cout << has_const_reference_op<unique_ptr<long>,int &>::value;
    cout << has_const_reference_op<int,int>::value;
    cout << has_const_reference_op<std::vector<int>,int &>::value;
    cout << has_const_reference_op<ee_ref<int>,int &>::value;
    cout << has_const_reference_op<sub_ee_ref,int &>::value;
    cout << has_const_reference_op<empty,int &>::value  << endl;
    return 0;
}
```

Are there new flaws in this idea? Can it be made more generic without once again falling foul of the snag it avoids?

Share Improve this answer Follow

edited May 23, 2017 at 11:33

answered May 22, 2012 at 18:23

Community Bot
1 1

Mike Kinghan
53.4k 11 147
174

This is neat, and also works with C++03 with gcc/clang __typeof__() in place of decltype(). However, I've found an issue when the method has both const and non-const overloads. In this case neither overload is detected – JustinC Oct 6, 2022 at 6:34

21

Here are some usage snippets: *The guts for all this are farther down

Check for member `x` in a given class. Could be var, func, class, union, or enum:

```
CREATE_MEMBER_CHECK(x);
bool has_x = has_member_x<class_to_check_for_x>::value;
```

Check for member function `void x()`:

```
//Func signature MUST have T as template variable here... simpler
this way :\
CREATE_MEMBER_FUNC_SIG_CHECK(x, void (T::*)(), void__x);
bool has_func_sig_void__x =
has_member_func_void__x<class_to_check_for_x>::value;
```

Check for member variable `x`:

```
CREATE_MEMBER_VAR_CHECK(x);
bool has_var_x = has_member_var_x<class_to_check_for_x>::value;
```

Check for member class `x`:

```
CREATE_MEMBER_CLASS_CHECK(x);
bool has_class_x = has_member_class_x<class_to_check_for_x>::value;
```

Check for member union `x`:

```
CREATE_MEMBER_UNION_CHECK(x);
bool has_union_x = has_member_union_x<class_to_check_for_x>::value;
```

Check for member enum `x`:

```
CREATE_MEMBER_ENUM_CHECK(x);
bool has_enum_x = has_member_enum_x<class_to_check_for_x>::value;
```

Check for any member function `x` regardless of signature:

```
CREATE_MEMBER_CHECK(x);
CREATE_MEMBER_VAR_CHECK(x);
CREATE_MEMBER_CLASS_CHECK(x);
CREATE_MEMBER_UNION_CHECK(x);
CREATE_MEMBER_ENUM_CHECK(x);
CREATE_MEMBER_FUNC_CHECK(x);
bool has_any_func_x = has_member_func_x<class_to_check_for_x>::value;
```

OR

```
CREATE_MEMBER_CHECKS(x);  //Just stamps out the same macro calls as
above.
bool has_any_func_x = has_member_func_x<class_to_check_for_x>::value;
```

Details and core:

```
/*
    - Multiple inheritance forces ambiguity of member names.
    - SFINAE is used to make aliases to member names.
    - Expression SFINAE is used in just one generic has_member that
can accept
      any alias we pass it.
*/

//Variadic to force ambiguity of class members.  C++11 and up.
template <typename... Args> struct ambiguate : public Args... {};

//Non-variadic version of the line above.
//template <typename A, typename B> struct ambiguate : public A,
public B {};

template<typename A, typename = void>
struct got_type : std::false_type {};

template<typename A>
struct got_type<A> : std::true_type {
    typedef A type;
};

template<typename T, T>
```

```
struct sig_check : std::true_type {};

template<typename Alias, typename AmbiguitySeed>
struct has_member {
    template<typename C> static char ((&f(decltype(&C::value))))[1];
    template<typename C> static char ((&f(...)))[2];

    //Make sure the member name is consistently spelled the same.
    static_assert(
        (sizeof(f<AmbiguitySeed>(0)) == 1)
        , "Member name specified in AmbiguitySeed is different from
member name specified in Alias, or wrong Alias/AmbiguitySeed has been
specified."
    );

    static bool const value = sizeof(f<Alias>(0)) == 2;
};
```

Macros (El Diablo!):

CREATE_MEMBER_CHECK:

```
//Check for any member with given name, whether var, func, class,
union, enum.
#define CREATE_MEMBER_CHECK(member)
\

\
template<typename T, typename = std::true_type>
\
struct Alias_##member;
\

\
template<typename T>
\
struct Alias_##member <
\
    T, std::integral_constant<bool,
got_type<decltype(&T::member)>::value>  \
> { static const decltype(&T::member) value; };
\

\
struct AmbiguitySeed_##member { char member; };
\

\
```

```
template<typename T>
\
struct has_member_##member {
\
    static const bool value
\
        = has_member<
\
            Alias_##member<ambiguate<T, AmbiguitySeed_##member>>
\
            , Alias_##member<AmbiguitySeed_##member>
\
        >::value
\
    ;
\
}
```

CREATE_MEMBER_VAR_CHECK:

```
//Check for member variable with given name.
#define CREATE_MEMBER_VAR_CHECK(var_name)
\

\
template<typename T, typename = std::true_type>
\
struct has_member_var_##var_name : std::false_type {};
\

\
template<typename T>
\
struct has_member_var_##var_name<
\
    T
\
    , std::integral_constant<
\
        bool
\
        ,
!std::is_member_function_pointer<decltype(&T::var_name)>::value   \
    >
\
> : std::true_type {}
```

CREATE_MEMBER_FUNC_SIG_CHECK:

```
//Check for member function with given name AND signature.
#define CREATE_MEMBER_FUNC_SIG_CHECK(func_name, func_sig,
templ_postfix)    \

\
template<typename T, typename = std::true_type>
\
struct has_member_func_##templ_postfix : std::false_type {};
\

\
template<typename T>
\
struct has_member_func_##templ_postfix<
\
    T, std::integral_constant<
\
        bool
\
        , sig_check<func_sig, &T::func_name>::value
\
    >
\
> : std::true_type {}
```

CREATE_MEMBER_CLASS_CHECK:

```
//Check for member class with given name.
#define CREATE_MEMBER_CLASS_CHECK(class_name)               \
                                                            \
template<typename T, typename = std::true_type>             \
struct has_member_class_##class_name : std::false_type {};  \
                                                            \
template<typename T>                                        \
struct has_member_class_##class_name<                       \
    T                                                       \
    , std::integral_constant<                               \
        bool                                                \
        , std::is_class<                                    \
            typename got_type<typename T::class_name>::type \
        >::value                                            \
    >                                                       \
> : std::true_type {}
```

CREATE_MEMBER_UNION_CHECK:

```
//Check for member union with given name.
#define CREATE_MEMBER_UNION_CHECK(union_name)                   \
                                                                \
template<typename T, typename = std::true_type>                 \
struct has_member_union_##union_name : std::false_type {};      \
                                                                \
template<typename T>                                            \
struct has_member_union_##union_name<                           \
    T                                                           \
    , std::integral_constant<                                   \
        bool                                                    \
        , std::is_union<                                        \
            typename got_type<typename T::union_name>::type     \
        >::value                                                \
    >                                                           \
> : std::true_type {}
```

CREATE_MEMBER_ENUM_CHECK:

```
//Check for member enum with given name.
#define CREATE_MEMBER_ENUM_CHECK(enum_name)                     \
                                                                \
template<typename T, typename = std::true_type>                 \
struct has_member_enum_##enum_name : std::false_type {};        \
                                                                \
template<typename T>                                            \
struct has_member_enum_##enum_name<                             \
    T                                                           \
    , std::integral_constant<                                   \
        bool                                                    \
        , std::is_enum<                                         \
            typename got_type<typename T::enum_name>::type      \
        >::value                                                \
    >                                                           \
> : std::true_type {}
```

CREATE_MEMBER_FUNC_CHECK:

```
//Check for function with given name, any signature.
#define CREATE_MEMBER_FUNC_CHECK(func)            \
template<typename T>                              \
struct has_member_func_##func {                   \
    static const bool value                       \
        = has_member_##func<T>::value             \
        && !has_member_var_##func<T>::value       \
        && !has_member_class_##func<T>::value     \
        && !has_member_union_##func<T>::value     \
```

```
        && !has_member_enum_##func<T>::value     \
    ;                                            \
}
```

CREATE_MEMBER_CHECKS:

```
//Create all the checks for one member.  Does NOT include func sig
checks.
#define CREATE_MEMBER_CHECKS(member)    \
CREATE_MEMBER_CHECK(member);            \
CREATE_MEMBER_VAR_CHECK(member);        \
CREATE_MEMBER_CLASS_CHECK(member);      \
CREATE_MEMBER_UNION_CHECK(member);      \
CREATE_MEMBER_ENUM_CHECK(member);       \
CREATE_MEMBER_FUNC_CHECK(member)
```

Share Improve this answer Follow

answered May 31, 2013 at 23:30

Brett Rossier
3,390 3 26 36

2 This is great; it would be nice to put this in a single header file library. – Allan Oct 7, 2019 at 16:06

13

This should be sufficient, if you know the name of the member function you are expecting. (In this case, the function bla fails to instantiate if there is no member function (writing one that works anyway is tough because there is a lack of function partial specialization. You may need to use class templates) Also, the enable struct (which is similar to enable_if) could also be templated on the type of function you want it to have as a member.

```
template <typename T, int (T::*) ()> struct enable { typedef T type;
};
template <typename T> typename enable<T, &T::i>::type bla (T&);
struct A { void i(); };
struct B { int i(); };
int main()
{
  A a;
  B b;
  bla(b);
  bla(a);
}
```

Share Improve this answer Follow

edited Oct 27, 2008 at 17:18

Head Geek
37.2k 22 77 86

answered Sep 17, 2008 at 21:33

coppro
14.2k 5 57 73

5 thaks! it's similar to the solution proposed by yrp. I didn't know that template can be templated over member functions. That's a new feature I've learned today! ... and a new lesson: "never say you are expert on c++" :) – ugasoft Sep 17, 2008 at 21:43

11

With c++ 20 this becomes much simpler. Say we want to test if a class `T` has a member function `void T::resize(typename T::size_type)`. For example, `std::vector<U>` has such a member function. Then,

```
template<typename T>
concept has_resize_member_func = requires {
    typename T::size_type;
    { std::declval<T>().resize(std::declval<typename T::size_type>()) 
} -> std::same_as<void>;
};
```

and the usage is

```
static_assert(has_resize_member_func<std::string>, "");
static_assert(has_resize_member_func<int> == false, "");
```

Share Improve this answer Follow

answered Oct 11, 2020 at 19:09

Peter Rindal
333 2 8

10

Here is a simpler take on Mike Kinghan's answer. This will detect inherited methods. It will also check for the *exact* signature (unlike jrok's approach which allows argument conversions).

```
template <class C>
class HasGreetMethod
{
    template <class T>
    static std::true_type testSignature(void (T::*)(const char*) 
const);

    template <class T>
    static decltype(testSignature(&T::greet)) test(std::nullptr_t);
```

```
    template <class T>
    static std::false_type test(...);

public:
    using type = decltype(test<C>(nullptr));
    static const bool value = type::value;
};

struct A { void greet(const char* name) const; };
struct Derived : A { };
static_assert(HasGreetMethod<Derived>::value, "");
```

Runnable [example]

Share Improve this answer Follow

answered Jul 21, 2015 at 12:42

Valentin Milea
3,106 3 28 28

This is good, but it won't work if the function takes no argument – Triskeldeian May 10, 2016 at 17:27

It does work great. I did not have any problems applying this trick to member functions taking no arguments. – JohnB Nov 27, 2016 at 16:45

This works well for me with multiple and no method arguments, including with overloads, and including with inheritance, and with the use of `using` to bring overloads from the base class. It works for me on MSVC 2015 and with Clang-CL. It doesn't work with MSVC 2012 however. – steveire Mar 3, 2017 at 11:37

8

You appear to want the detector idiom. The above answers are variations on this that work with C++11 or C++14.

The `std::experimental` library has features which do essentially this. Reworking an example from above, it might be:

```
#include <experimental/type_traits>

// serialized_method_t is a detector type for T.serialize(int) const
template<typename T>
using serialized_method_t = decltype(std::declval<const T&>
().serialize(std::declval<int>()));

// has_serialize_t is std::true_type when T.serialize(int) exists,
// and false otherwise.
```

```
template<typename T>
using has_serialize_t =
std::experimental::is_detected_t<serialized_method_t, T>;
```

If you can't use std::experimental, a rudimentary version can be made like this:

```
template <typename... Ts>
using void_t = void;
template <template <class...> class Trait, class AlwaysVoid, class...
Args>
struct detector : std::false_type {};
template <template <class...> class Trait, class... Args>
struct detector<Trait, void_t<Trait<Args...>>, Args...> :
std::true_type {};

// serialized_method_t is a detector type for T.serialize(int) const
template<typename T>
using serialized_method_t = decltype(std::declval<const T&>
().serialize(std::declval<int>()));

// has_serialize_t is std::true_type when T.serialize(int) exists,
// and false otherwise.
template <typename T>
using has_serialize_t = typename detector<serialized_method_t, void,
T>::type;
```

Since has_serialize_t is really either std::true_type or std::false_type, it can be used via any of the common SFINAE idioms:

```
template<class T>
std::enable_if_t<has_serialize_t<T>::value, std::string>
SerializeToString(const T& t) {
}
```

Or by using dispatch with overload resolution:

```
template<class T>
std::string SerializeImpl(std::true_type, const T& t) {
  // call serialize here.
}

template<class T>
std::string SerializeImpl(std::false_type, const T& t) {
  // do something else here.
}
```

```
template<class T>
std::string Serialize(const T& t) {
  return SerializeImpl(has_serialize_t<T>{}, t);
}
```

Share Improve this answer Follow

edited Feb 13, 2021 at 11:33
cmannett85
21.4k 8 74 112

answered May 28, 2020 at 9:50
lar
81 1 1

For modern C++, this is the best answer of the lot. – cmannett85 Feb 13, 2021 at 11:33

This is my favorite answer, but this solution (like many others) only checks that the method can be invoked with a given type. To ensure an exact signature match, use an integral_constant template as Op. template<class T> using serialize_method_t = std::integral_constant<void(T::*)(int), &T::serialize> – jisrael18 Mar 3, 2021 at 17:08

5

You can use **[std::is_member_function_pointer](#)**

```
class A {
   public:
     void foo() {};
}

 bool test =
std::is_member_function_pointer<decltype(&A::foo)>::value;
```

Share Improve this answer Follow

answered Nov 7, 2012 at 8:58
Yochai Timmer
47.2k 23 144 184

18 Won't `&A::foo` be a compile error if there's no `foo` at all in `A` ? I read the original question as being supposed to work with any input class, not just ones that have some sort of member named `foo` . – Jeff Walden Mar 11, 2013 at 22:11

This does not work. It gives a compile error if the function is not a member of `A` . – Marc Dirven Sep 28, 2020 at 19:42

5

Came with the same kind of problem myself, and found the proposed solutions in here very interesting... but had the requirement for a solution that:

1. Detects inherited functions as well;
2. Is compatible with non C++11 ready compilers (so no decltype)

Found another thread proposing something like this, based on a BOOST discussion. Here is the generalisation of the proposed solution as two macros declaration for traits class, following the model of boost::has_* classes.

```
#include <boost/type_traits/is_class.hpp>
#include <boost/mpl/vector.hpp>

/// Has constant function
/** \param func_ret_type Function return type
    \param func_name Function name
    \param ... Variadic arguments are for the function parameters
*/
#define DECLARE_TRAITS_HAS_FUNC_C(func_ret_type, func_name, ...) \
    __DECLARE_TRAITS_HAS_FUNC(1, func_ret_type, func_name,
##__VA_ARGS__)

/// Has non-const function
/** \param func_ret_type Function return type
    \param func_name Function name
    \param ... Variadic arguments are for the function parameters
*/
#define DECLARE_TRAITS_HAS_FUNC(func_ret_type, func_name, ...) \
    __DECLARE_TRAITS_HAS_FUNC(0, func_ret_type, func_name,
##__VA_ARGS__)

// Traits content
#define __DECLARE_TRAITS_HAS_FUNC(func_const, func_ret_type,
func_name, ...)  \
    template
\
    <   typename Type,
\
        bool is_class = boost::is_class<Type>::value
\
    >
\
    class has_func_ ## func_name;
\
    template<typename Type>
\
    class has_func_ ## func_name<Type,false>
```

```
\
    {public:
\
        BOOST_STATIC_CONSTANT( bool, value = false );
\
        typedef boost::false_type type;
\
    };
\
    template<typename Type>
\
    class has_func_ ## func_name<Type,true>
\
    {   struct yes { char _foo; };
\
        struct no { yes _foo[2]; };
\
        struct Fallback
\
        {   func_ret_type func_name( __VA_ARGS__ )
\
                UTILITY_OPTIONAL(func_const,const) {}
\
        };
\
        struct Derived : public Type, public Fallback {};
\
        template <typename T, T t>  class Helper{};
\
        template <typename U>
\
        static no deduce(U*, Helper
\
            <   func_ret_type (Fallback::*)( __VA_ARGS__ )
\
                    UTILITY_OPTIONAL(func_const,const),
\
                &U::func_name
\
            >* = 0
\
        );
\
        static yes deduce(...);
\
    public:
\
        BOOST_STATIC_CONSTANT(
\
```

```
                bool,
\
                value = sizeof(yes)
\
                    == sizeof( deduce( static_cast<Derived*>(0) ) )
\
            );
\
            typedef ::boost::integral_constant<bool,value> type;
\
            BOOST_STATIC_CONSTANT(bool, is_const = func_const);
\
            typedef func_ret_type return_type;
\
            typedef ::boost::mpl::vector< __VA_ARGS__ > args_type;
\
        }

// Utility functions
#define UTILITY_OPTIONAL(condition, ...) UTILITY_INDIRECT_CALL(
__UTILITY_OPTIONAL_ ## condition , ##__VA_ARGS__ )
#define UTILITY_INDIRECT_CALL(macro, ...) macro ( __VA_ARGS__ )
#define __UTILITY_OPTIONAL_0(...)
#define __UTILITY_OPTIONAL_1(...) __VA_ARGS__
```

These macros expand to a traits class with the following prototype:

```
template<class T>
class has_func_[func_name]
{
public:
    /// Function definition result value
    /** Tells if the tested function is defined for type T or not.
    */
    static const bool value = true | false;

    /// Function definition result type
    /** Type representing the value attribute usable in

http://www.boost.org/doc/libs/1_53_0/libs/utility/enable_if.html
    */
    typedef boost::integral_constant<bool,value> type;

    /// Tested function constness indicator
    /** Indicates if the tested function is const or not.
        This value is not deduced, it is forced depending
        on the user call to one of the traits generators.
    */
```

```
    static const bool is_const = true | false;

    /// Tested function return type
    /** Indicates the return type of the tested function.
        This value is not deduced, it is forced depending
        on the user's arguments to the traits generators.
    */
    typedef func_ret_type return_type;

    /// Tested function arguments types
    /** Indicates the arguments types of the tested function.
        This value is not deduced, it is forced depending
        on the user's arguments to the traits generators.
    */
    typedef ::boost::mpl::vector< __VA_ARGS__ > args_type;
};
```

So what is the typical usage one can do out of this?

```
// We enclose the traits class into
// a namespace to avoid collisions
namespace ns_0 {
    // Next line will declare the traits class
    // to detect the member function void foo(int,int) const
    DECLARE_TRAITS_HAS_FUNC_C(void, foo, int, int);
}

// we can use BOOST to help in using the traits
#include <boost/utility/enable_if.hpp>

// Here is a function that is active for types
// declaring the good member function
template<typename T> inline
typename boost::enable_if< ns_0::has_func_foo<T> >::type
foo_bar(const T &_this_, int a=0, int b=1)
{   _this_.foo(a,b);
}

// Here is a function that is active for types
// NOT declaring the good member function
template<typename T> inline
typename boost::disable_if< ns_0::has_func_foo<T> >::type
foo_bar(const T &_this_, int a=0, int b=1)
{   default_foo(_this_,a,b);
}

// Let us declare test types
struct empty
```

```
{
};
struct direct_foo
{
    void foo(int,int);
};
struct direct_const_foo
{
    void foo(int,int) const;
};
struct inherited_const_foo :
    public direct_const_foo
{
};

// Now anywhere in your code you can seamlessly use
// the foo_bar function on any object:
void test()
{
    int a;
    foo_bar(a); // calls default_foo

    empty b;
    foo_bar(b); // calls default_foo

    direct_foo c;
    foo_bar(c); // calls default_foo (member function is not const)

    direct_const_foo d;
    foo_bar(d); // calls d.foo (member function is const)

    inherited_const_foo e;
    foo_bar(e); // calls e.foo (inherited member function)
}
```

Share Improve this answer Follow

edited May 23, 2017 at 12:03

Community Bot
1 1

answered Mar 28, 2013 at 15:16

S. Paris
159 4 5

5

To accomplish this we'll need to use:

1. Function template overloading with differing return types according to whether the method is available
2. In keeping with the meta-conditionals in the `type_traits` header, we'll want to return a `true_type` or `false_type` from our overloads

3. Declare the `true_type` overload expecting an `int` and the `false_type` overload expecting Variadic Parameters to exploit: "The lowest priority of the ellipsis conversion in overload resolution"

4. In defining the template specification for the `true_type` function we will use `declval` and `decltype` allowing us to detect the function independent of return type differences or overloads between methods

You can see a live example of this here. But I'll also explain it below:

I want to check for the existence of a function named `test` which takes a type convertible from `int`, then I'd need to declare these two functions:

```
template <typename T, typename S = decltype(declval<T>
().test(declval<int>))> static true_type hasTest(int);
template <typename T> static false_type hasTest(...);
```

- `decltype(hasTest<a>(0))::value` is `true` (Note there is no need to create special functionality to deal with the `void a::test()` overload, the `void a::test(int)` is accepted)
- `decltype(hasTest<b>(0))::value` is `true` (Because `int` is convertable to `double` `int b::test(double)` is accepted, independent of return type)
- `decltype(hasTest<c>(0))::value` is `false` (`c` does not have a method named `test` that accepts a type convertible from `int` therefor this is not accepted)

This solution has 2 drawbacks:

1. Requires a per method declaration of a pair of functions
2. Creates namespace pollution particularly if we want to test for similar names, for example what would we name a function that wanted to test for a `test()` method?

So it's important that these functions be declared in a details namespace, or ideally if they are only to be used with a class, they should be declared privately by that class. To that end I've written a macro to help you abstract this information:

```
#define FOO(FUNCTION, DEFINE) template <typename T, typename S =
decltype(declval<T>().FUNCTION)> static true_type __ ## DEFINE(int);
\
                              template <typename T> static false_type
__ ## DEFINE(...); \
                              template <typename T> using DEFINE =
decltype(__ ## DEFINE<T>(0));
```

You could use this like:

```
namespace details {
    FOO(test(declval<int>()), test_int)
    FOO(test(), test_void)
}
```

Subsequently calling `details::test_int<a>::value` or `details::test_void<a>::value` would yield `true` or `false` for the purposes of inline code or meta-programming.

Share Improve this answer Follow

edited Dec 11, 2018 at 16:25

answered May 9, 2016 at 13:27

Jonathan Mee
37.3k 21 124 283

4

To be non-intrusive, you can also put `serialize` in the namespace of the class being serialised, or of the archive class, thanks to [Koenig lookup](). See [Namespaces for Free Function Overrides]() for more details. :-)

Opening up any given namespace to implement a free function is Simply Wrong. (e.g., you're not supposed to open up namespace `std` to implement `swap` for your own types, but should use Koenig lookup instead.)

Share Improve this answer Follow

answered Sep 17, 2008 at 20:44

C. K. Young
217k 44 383 428

2

Okay. Second try. It's okay if you don't like this one either, I'm looking for more ideas.

Herb Sutter's article talks about traits. So you can have a traits class whose default instantiation has the fallback behaviour, and for each class where your member function exists, then the traits class is specialised to invoke the member function. I believe Herb's article mentions a technique to do this so that it doesn't involve lots of copying and pasting.

Like I said, though, perhaps you don't want the extra work involved with "tagging" classes that do implement that member. In which case, I'm looking at a third solution....

Share Improve this answer Follow

answered Sep 17, 2008 at 21:16

C. K. Young
217k 44 383 428

eh... I've analyzed this solution... I think it's a little bit too expensive for users of my framework. (ok, I admit, I'm developing a streaming framework and I'm choosing between extending iostream or rewriting something easier) – ugasoft Sep 17, 2008 at 21:34

My third solution would be to use SFINAE. Since yrp's answer already mentions it, I won't go into it (because I'm still researching on it: I know the idea, but the devil is in the details), unless his solution doesn't work for you in the end. :-) – C. K. Young Sep 17, 2008 at 22:00

2

I had a similar need and came across o this SO. There are many interesting/powerful solutions proposed here, though it is a bit long for just a specific need : detect if a class has member function with a precise signature. So I did some reading/testing and came up with my version that could be of interest. It detect :

- static member function
- non-static member function
- non-static member function const

with a precise signature. Since I don't need to capture *any* signature (that'd require a more complicated solution), this one suites to me. It basically used **enable_if_t**.

```
struct Foo{ static int sum(int, const double&){return 0;} };
struct Bar{ int calc(int, const double&) {return 1;} };
struct BarConst{ int calc(int, const double&) const {return 1;} };

// Note : second typename can be void or anything, as long as it is
consistent with the result of enable_if_t
template<typename T, typename = T> struct has_static_sum :
std::false_type {};
template<typename T>
struct has_static_sum<typename T,

std::enable_if_t<std::is_same<decltype(T::sum), int(int, const
double&)>::value,T>
                      > : std::true_type {};

template<typename T, typename = T> struct has_calc : std::false_type
{};
template<typename T>
struct has_calc <typename T,
                  std::enable_if_t<std::is_same<decltype(&T::calc),
int(T::*)(int, const double&)>::value,T>
                > : std::true_type {};

template<typename T, typename = T> struct has_calc_const :
```

```
std::false_type {};
template<typename T>
struct has_calc_const <T,

std::enable_if_t<std::is_same<decltype(&T::calc), int(T::*)(int,
const double&) const>::value,T>
                      > : std::true_type {};

int main ()
{
    constexpr bool has_sum_val = has_static_sum<Foo>::value;
    constexpr bool not_has_sum_val = !has_static_sum<Bar>::value;

    constexpr bool has_calc_val = has_calc<Bar>::value;
    constexpr bool not_has_calc_val = !has_calc<Foo>::value;

    constexpr bool has_calc_const_val =
has_calc_const<BarConst>::value;
    constexpr bool not_has_calc_const_val =
!has_calc_const<Bar>::value;

    std::cout<< "           has_sum_val " << has_sum_val
<< std::endl
             << "       not_has_sum_val " << not_has_sum_val
<< std::endl
             << "          has_calc_val " << has_calc_val
<< std::endl
             << "      not_has_calc_val " << not_has_calc_val
<< std::endl
             << "    has_calc_const_val " << has_calc_const_val
<< std::endl
             << "not_has_calc_const_val " << not_has_calc_const_val
<< std::endl;
}
```

Output :

```
           has_sum_val 1
       not_has_sum_val 1
          has_calc_val 1
      not_has_calc_val 1
    has_calc_const_val 1
not_has_calc_const_val 1
```

ctNGUYEN
241 2 6

2

If you are using facebook folly, there are out of box macro to help you:

```
#include <folly/Traits.h>
namespace {
  FOLLY_CREATE_HAS_MEMBER_FN_TRAITS(has_test_traits, test);
} // unnamed-namespace

void some_func() {
  cout << "Does class Foo have a member int test() const? "
    << boolalpha << has_test_traits<Foo, int() const>::value;
}
```

Though the implementation details is the same with the previous answer, use a library is simpler.

Share Improve this answer Follow edited Sep 23, 2022 at 3:02 answered Apr 15, 2019 at 3:32

prehistoricpenguin
6,025 3 23 41

1

Without C++11 support (`decltype`) this might work:

SSCCE

```
#include <iostream>
using namespace std;

struct A { void foo(void); };
struct Aa: public A { };
struct B { };

struct retA { int foo(void); };
struct argA { void foo(double); };
struct constA { void foo(void) const; };
struct varA { int foo; };

template<typename T>
struct FooFinder {
    typedef char true_type[1];
    typedef char false_type[2];

    template<int>
```

```
    struct TypeSink;

    template<class U>
    static true_type &match(U);

    template<class U>
    static true_type &test(TypeSink<sizeof( matchType<void (U::*)
(void)>( &U::foo ) )> *);

    template<class U>
    static false_type &test(...);

    enum { value = (sizeof(test<T>(0, 0)) == sizeof(true_type)) };
};

int main() {
    cout << FooFinder<A>::value << endl;
    cout << FooFinder<Aa>::value << endl;
    cout << FooFinder<B>::value << endl;

    cout << FooFinder<retA>::value << endl;
    cout << FooFinder<argA>::value << endl;
    cout << FooFinder<constA>::value << endl;
    cout << FooFinder<varA>::value << endl;
}
```

How it hopefully works

`A`, `Aa` and `B` are the clases in question, `Aa` being the special one that inherits the member we're looking for.

In the `FooFinder` the `true_type` and `false_type` are the replacements for the correspondent C++11 classes. Also for the understanding of template meta programming, they reveal the very basis of the SFINAE-sizeof-trick.

The `TypeSink` is a template struct that is used later to sink the integral result of the `sizeof` operator into a template instantiation to form a type.

The `match` function is another SFINAE kind of template that is left without a generic counterpart. It can hence only be instantiated if the type of its argument matches the type it was specialized for.

Both the `test` functions together with the enum declaration finally form the central SFINAE pattern. There is a generic one using an ellipsis that returns the `false_type` and a counterpart with more specific arguments to take precedence.

To be able to instantiate the `test` function with a template argument of `T`, the `match` function must be instantiated, as its return type is required to instantiate the `TypeSink` argument. The caveat is that `&U::foo`, being wrapped in a function argument, is *not* referred to from within a template argument specialization, so inherited member lookup still takes place.

Share Improve this answer Follow

answered Jul 9, 2017 at 17:21

Kamajii
1,826 9 19

1

Building on jrok's answer, I have avoided using nested template classes and/or functions.

```
#include <type_traits>

#define CHECK_NESTED_FUNC(fName) \
    template <typename, typename, typename = std::void_t<>> \
    struct _has_##fName \
    : public std::false_type {}; \
    \
    template <typename Class, typename Ret, typename... Args> \
    struct _has_##fName<Class, Ret(Args...), \
        std::void_t<decltype(std::declval<Class>
().fName(std::declval<Args>()...))>> \
    : public std::is_same<decltype(std::declval<Class>
().fName(std::declval<Args>()...)), Ret> \
    {}; \
    \
    template <typename Class, typename Signature> \
    using has_##fName = _has_##fName<Class, Signature>;

#define HAS_NESTED_FUNC(Class, Func, Signature) has_##Func<Class,
Signature>::value
```

We can use the above macros as below:

```
class Foo
{
public:
    void Bar(int, const char *) {}
};

CHECK_NESTED_FUNC(Bar);  // generate required metafunctions

int main()
{
    using namespace std;
```

```
    cout << boolalpha
         << HAS_NESTED_FUNC(Foo, Bar, void(int, const char *))  //
prints true
         << endl;
    return 0;
}
```

Suggestions are welcome.

Share Improve this answer Follow

edited Jul 2, 2020 at 20:07

answered Jun 28, 2020 at 18:50

debashish.ghosh
113 8