C++ SFINAE examples?

Asked 12 years, 11 months ago Modified 5 months ago Viewed 58k times

155

I want to get into more template meta-programming. I know that SFINAE stands for "substitution failure is not an error." But can someone show me a good use for SFINAE?

c++ templates metaprogramming sfinae

102

Share Edit Follow Flag

asked Jun 11, 2009 at 18:25

rlbond
62.5k • 53 • 169 • 222

4 This is a good question. I understand SFINAE pretty well, but I don't think I've ever had to use it (unless libraries are doing it without me knowing it). – Zifre Jun 11, 2009 at 19:05

10 Answers

Sorted by: Highest score (default)

99

I like using `SFINAE` to check boolean conditions.

```
template<int I> void div(char(*)[I % 2 == 0] = 0) {
    /* this is taken when I is even */
}

template<int I> void div(char(*)[I % 2 == 1] = 0) {
    /* this is taken when I is odd */
}
```

It can be quite useful. For example, i used it to check whether an initializer list collected using operator comma is no longer than a fixed size

```
template<int N>
struct Vector {
    template<int M>
    Vector(MyInitList<M> const& i, char(*)[M <= N] = 0) { /* ... */ }
}
```

The list is only accepted when M is smaller than N, which means that the initializer list has not too many elements.

The syntax `char(*)[C]` means: Pointer to an array with element type char and size `C`. If `C` is false (0 here), then we get the invalid type `char(*)[0]`, pointer to a zero sized array: SFINAE makes it so that the template will be ignored then.

Expressed with `boost::enable_if`, that looks like this

```
template<int N>
struct Vector {
    template<int M>
    Vector(MyInitList<M> const& i,
           typename enable_if_c<(M <= N)>::type* = 0) { /* ... */ }
}
```

In practice, i often find the ability to check conditions a useful ability.

Share Edit Follow Flag

answered Jun 12, 2009 at 23:40

Johannes Schaub - litb
481k • 123 • 868 • 1188

3 @Johannes Weirdly enough, GCC (4.8) and Clang (3.2) accept to declare arrays of size 0 (so the type is not really "invalid"), yet it behaves properly on your code. There is probably special support for this case in the case of SFINAE vs. "regular" uses of types. – akim Feb 5, 2013 at 9:07

@akim: if that is ever true (weird ?! since when ?) then maybe `M <= N ? 1 : -1` could work instead. – v.oddou Jun 13, 2014 at 10:43

2 @v.oddou Just try `int foo[0]`. I'm not surprised it's supported, as it allows the very useful "struct ending with a 0-length array" trick (gcc.gnu.org/onlinedocs/gcc/Zero-Length.html). – akim Jun 14, 2014 at 16:06

@akim: yeah its what I thought -> C99. This is not allowed in C++, here is what you get with a modern compiler : `error C2466: cannot allocate an array of constant size 0` – v.oddou Jun 16, 2014 at 1:31

3 @v.oddou No, I really meant C++, and actually C++11: both clang++ and g++ accept it, and I have pointed to a page that explains why this is useful. – akim Jun 16, 2014 at 6:45

79

Heres one example (from here):

```
template<typename T>
class IsClassT {
  private:
    typedef char One;
    typedef struct { char a[2]; } Two;
    template<typename C> static One test(int C::*);
    // Will be chosen if T is anything except a class.
    template<typename C> static Two test(...);
  public:
    enum { Yes = sizeof(IsClassT<T>::test<T>(0)) == 1 };
    enum { No = !Yes };
};
```

When `IsClassT<int>::Yes` is evaluated, 0 cannot be converted to `int int::*` because int is not a class, so it can't have a member pointer. If SFINAE didn't exist, then you would get a compiler error, something like '0 cannot be converted to member pointer for non-class type int'. Instead, it just uses the `...` form which returns Two, and thus evaluates to false, int is not a class type.

Share Edit Follow Flag

edited Sep 13, 2012 at 20:58

John Kugelman
328k • 66 • 501 • 553

answered Jun 11, 2009 at 18:54

Greg Rogers
34.5k • 17 • 65 • 94

8 @rlbond, i answered your question in the comments to this question here: stackoverflow.com/questions/822059/... . In short: If both test functions are candidates and viable, then "..." has the worst conversion cost, and hence will never be taken, in favor of the other function. "..." is the ellipsis, var-arg thing: int printf(char const*, ...); – Johannes Schaub - litb Jun 12, 2009 at 23:25

The link changed to blog.olivierlanglois.net/index.php/2007/09/01/... – tstenner Aug 25, 2009 at 17:32

29 The weirder thing here IMO is not the `...`, but rather the `int C::*`, which I'd never seen and had to go look up. Found the answer for what that is and what it might be used for here: stackoverflow.com/questions/670734/... – HostileFork says dont trust SE Jul 12, 2012 at 6:01

1 can someone explain what C::* is? I read all the comments and links, but I am still wondering, int C::* means that it is a member pointer of int type. what if a class has no member of int type? What am I missing? and how does test<T>(0) play into this? I must be missing something – user2584960 Nov 2, 2018 at 2:15

Can you explain why you use template in this : template<typename C> static Two test(...); ? – user6547518 Jul 8, 2021 at 13:51

It's surprise me to use that because at the end of this page en.cppreference.com/w/cpp/language/sfinae, there is an example quite similar without it. – user6547518 Jul 8, 2021 at 14:10

16

In C++11 SFINAE tests have become much prettier. Here are a few examples of common uses:

Pick a function overload depending on traits

```
template<typename T>
std::enable_if_t<std::is_integral<T>::value> f(T t){
    //integral version
}
template<typename T>
std::enable_if_t<std::is_floating_point<T>::value> f(T t){
    //floating point version
}
```

Using a so called type sink idiom you can do pretty arbitrary tests on a type like checking if it has a member and if that member is of a certain type

```
//this goes in some header so you can use it everywhere
template<typename T>
struct TypeSink{
    using Type = void;
};
template<typename T>
using TypeSinkT = typename TypeSink<T>::Type;

//use case
template<typename T, typename=void>
struct HasBarOfTypeInt : std::false_type{};
template<typename T>
struct HasBarOfTypeInt<T, TypeSinkT<decltype(std::declval<T&>().*(&T::bar))>> :
    std::is_same<typename std::decay<decltype(std::declval<T&>().*
(&T::bar))>::type,int>{};


struct S{
   int bar;
};
struct K{

};

template<typename T, typename = TypeSinkT<decltype(&T::bar)>>
void print(T){
    std::cout << "has bar" << std::endl;
}
void print(...){
    std::cout << "no bar" << std::endl;
}

int main(){
    print(S{});
    print(K{});
    std::cout << "bar is int: " << HasBarOfTypeInt<S>::value << std::endl;
}
```

Here is a live example: http://ideone.com/dHhyHE I also recently wrote a whole section on SFINAE and tag dispatch in my blog (shameless plug but relevant) http://metaporky.blogspot.de/2014/08/part-7-static-dispatch-function.html

Note as of C++14 there is a std::void_t which is essentially the same as my TypeSink here.

Share Edit Follow Flag

edited Mar 2, 2015 at 21:16

answered Aug 11, 2014 at 11:46

odinthenerd
5,192 • 1 • 27 • 58

Your first block of code redefines the same template. – T.C. Sep 21, 2014 at 5:43

Since there is no type for which is_integral and is_floating_point are both true it should be an either or because SFINAE will remove at least one. – odinthenerd Oct 7, 2014 at 17:53

You are redefining the same template with different default template arguments. Have you tried compiling it? – T.C. Oct 7, 2014 at 17:56

2 I'm new to template metaprogramming so I wanted to understand this example. Is there a reason you use `TypeSinkT<decltype(std::declval<T&>().*(&T::bar))>` at one place and then `TypeSinkT<decltype(&T::bar)>` at another? Also is the `&` necessary in `std::declval<T&>` ? – Kevin Doyon Dec 22, 2015 at 21:44

2 About your `TypeSink`, C++17 have `std::void_t` :) – YSC Dec 12, 2018 at 14:46

|

10

Boost's enable_if library offers a nice clean interface for using SFINAE. One of my favorite usage examples is in the Boost.Iterator library. SFINAE is used to enable iterator type conversions.

Share Edit Follow Flag

edited Jun 11, 2009 at 19:48

answered Jun 11, 2009 at 19:39

David Joyner
21.1k • 4 • 26 • 33

4

C++17 will probably provide a generic means to query for features. See N4502 for details, but as a self-contained example consider the following.

This part is the constant part, put it in a header.

```
// See http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4502.pdf.
template <typename...>
using void_t = void;

// Primary template handles all types not supporting the operation.
template <typename, template <typename> class, typename = void_t<>>
struct detect : std::false_type {};

// Specialization recognizes/validates only types supporting the archetype.
template <typename T, template <typename> class Op>
struct detect<T, Op, void_t<Op<T>>> : std::true_type {};
```

The following example, taken from N4502, shows the usage:

```
// Archetypal expression for assignment operation.
template <typename T>
using assign_t = decltype(std::declval<T&>() = std::declval<T const &>())

// Trait corresponding to that archetype.
template <typename T>
using is_assignable = detect<T, assign_t>;
```

Compared to the other implementations, this one is fairly simple: a reduced set of tools (`void_t` and `detect`) suffices. Besides, it was reported (see N4502) that it is measurably more efficient (compile-time and compiler memory consumption) than previous approaches.

Here is a live example, which includes portability tweaks for GCC pre 5.1.

Share Edit Follow Flag

answered Jun 16, 2015 at 16:49
akim
7,622 ● 2 ● 41 ● 56

Here's another (late) SFINAE example, based on Greg Rogers's answer:

```
template<typename T>
class IsClassT {
    template<typename C> static bool test(int C::*) {return true;}
    template<typename C> static bool test(...) {return false;}
public:
    static bool value;
};

template<typename T>
bool IsClassT<T>::value=IsClassT<T>::test<T>(0);
```

In this way, you can check the `value`'s value to see whether `T` is a class or not:

```
int main(void) {
    std::cout << IsClassT<std::string>::value << std::endl; // true
    std::cout << IsClassT<int>::value << std::endl;         // false
    return 0;
}
```

Share Edit Follow Flag

edited May 23, 2017 at 12:34
Community Bot
1 ● 1

answered Feb 28, 2015 at 15:42
whoan
7,731 ● 4 ● 37 ● 47

What does this syntax `int C::*` in your answer means? How can `C::*` be a parameter name? – Kirill Kobelev Jan 29, 2016 at 13:14

1 It's a pointer to member. Some reference: isocpp.org/wiki/faq/pointers-to-members – whoan Jan 29, 2016 at 15:00

@KirillKobelev `int C::*` is the type of a pointer to an `int` member variable of `C`. – YSC Dec 12, 2018 at 14:49

Here is one good article of SFINAE: An introduction to C++'s SFINAE concept: compile-time introspection of a class member.

Summary it as following:

```
/*
 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(...) { }
```

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

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 fail on access.

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

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

`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.

```
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';
}
```

Share Edit Follow Flag

answered Dec 25, 2015 at 7:13
zangw
37.1k ● 17 ● 141 ● 169

The following code uses SFINAE to let compiler select an overload based on whether a type has certain method or not:

```
#include <iostream>

template<typename T>
void do_something(const T& value, decltype(value.get_int()) = 0) {
    std::cout << "Int: " <<  value.get_int() << std::endl;
}

template<typename T>
void do_something(const T& value, decltype(value.get_float()) = 0) {
    std::cout << "Float: " << value.get_float() << std::endl;
}


struct FloatItem {
    float get_float() const {
        return 1.0f;
    }
};

struct IntItem {
    int get_int() const {
        return -1;
    }
};

struct UniversalItem : public IntItem, public FloatItem {};

int main() {
    do_something(FloatItem{});
    do_something(IntItem{});
    // the following fails because template substitution
    // leads to ambiguity
    // do_something(UniversalItem{});
    return 0;
}
```

Output:

```
Float: 1
Int: -1
```

Share Edit Follow Flag

answered Jul 18, 2020 at 7:10
cowboy
41 ● 2

Examples provided by other answers seems to me more complicated than needed.

Here is the slightly easier to understand example from cppreference :

```
#include <iostream>

// this overload is always in the set of overloads
// ellipsis parameter has the lowest ranking for overload resolution
void test(...)
{
    std::cout << "Catch-all overload called\n";
}

// this overload is added to the set of overloads if
// C is a reference-to-class type and F is a pointer to member function of C
template <class C, class F>
auto test(C c, F f) -> decltype((void)(c.*f)(), void())
{
    std::cout << "Reference overload called\n";
}

// this overload is added to the set of overloads if
// C is a pointer-to-class type and F is a pointer to member function of C
template <class C, class F>
auto test(C c, F f) -> decltype((void)((c->*f)()), void())
{
    std::cout << "Pointer overload called\n";
}

struct X { void f() {} };

int main(){
  X x;
  test( x, &X::f);
  test(&x, &X::f);
  test(42, 1337);
}
```

Output:

```
Reference overload called
Pointer overload called
Catch-all overload called
```

As you can see, in the third call of test, substitution fails without errors.

Share Edit Follow Flag

edited Nov 28, 2021 at 15:23

answered Jul 8, 2021 at 13:58
user6547518

Here, I am using template function overloading (not directly SFINAE) to determine whether a pointer is a function or member class pointer: (Is possible to fix the iostream cout/cerr member function pointers being printed as 1 or true?)

https://godbolt.org/z/c2NmzR

```
#include<iostream>

template<typename Return, typename... Args>
constexpr bool is_function_pointer(Return(*pointer)(Args...)) {
    return true;
}

template<typename Return, typename ClassType, typename... Args>
constexpr bool is_function_pointer(Return(ClassType::*pointer)(Args...)) {
    return true;
}

template<typename... Args>
constexpr bool is_function_pointer(Args...) {
    return false;
}

struct test_debugger { void var() {} };
void fun_void_void(){};
void fun_void_double(double d){};
double fun_double_double(double d){return d;}

int main(void) {
    int* var;

    std::cout << std::boolalpha;
    std::cout << "0. " << is_function_pointer(var) << std::endl;
    std::cout << "1. " << is_function_pointer(fun_void_void) << std::endl;
    std::cout << "2. " << is_function_pointer(fun_void_double) << std::endl;
    std::cout << "3. " << is_function_pointer(fun_double_double) << std::endl;
```

```
    std::cout << "4. " << is_function_pointer(&test_debugger::var) << std::endl;
    return 0;
}
```

Prints

```
0. false
1. true
2. true
3. true
4. true
```

As the code is, it **could** (depending on the compiler "good" will) generate a run time call to a function which will return true or false. If you would like to force the `is_function_pointer(var)` to evaluate at compile type (no function calls performed at run time), you can use the `constexpr` variable trick:

```
constexpr bool ispointer = is_function_pointer(var);
std::cout << "ispointer " << ispointer << std::endl;
```

By the C++ standard, all `constexpr` variables are guaranteed to be evaluated at compile time (Computing length of a C string at compile time. Is this really a constexpr?).

Share Edit Follow Flag

edited Feb 2, 2020 at 19:01

answered Jan 31, 2020 at 1:59

user
7,238 • 8 • 68 • 122