


Notes on C++ SFINAE, Modern C++ and C++20 Concepts

```
010typename10100010010100100100100000000011111000010010
001001001001001010100001001011111100constexpr1001000101
01001001                                0001001
0class11  11110010
01001010                                0101010
110101001001010101010101011111concept11110010requires10
010010enable1010101001001001001010101101010101101010001
```

What is SFINAE? Where can you use this metaprogramming technique? Are there any better alternatives in Modern C++? And how about Concepts from C++20?

Read on to find out!

Note: I'd like to thank KJ for reviewing this article and providing me with valuable feedback from the early stage of the writing process. Also, many thanks go to GW who reviewed the beta version.

Let's start with some basic ideas behind this concept:

Very briefly: the compiler can reject code that *"would not compile"* for a given type.

From [Wiki](#):

Substitution failure is not an error (SFINAE) refers to a situation in C++ where an invalid substitution of template parameters is not in itself an error. David Vandevoorde first introduced the acronym SFINAE to describe related programming techniques.

We're talking here about something related to templates, template substitution rules and metaprogramming... which make it a possibly a scary area!

A quick example:

```
struct Bar {
    typedef double internalType;
};

template <typename T>
typename T::internalType foo(const T& t) {
    cout << "foo<T>\n";
    return 0;
}

int main() {
    foo(Bar());
    foo(0); // << error!
}
```

Run [@Compiler Explorer](#).

We have one function template that returns `T::internalType`, and we call it with `Bar` and `int` param types.

The code, of course, will not compile. The first call of `foo(Bar());` is a proper construction, but the second call generates the following error (GCC):

```
no matching function for call to 'foo(int)'  
...  
template argument deduction/substitution failed:
```

When we make a simple correction and provide a suitable function for `int` types. As simple as:

```
int foo(int i) { cout << "foo(int)\n"; return 0; }
```

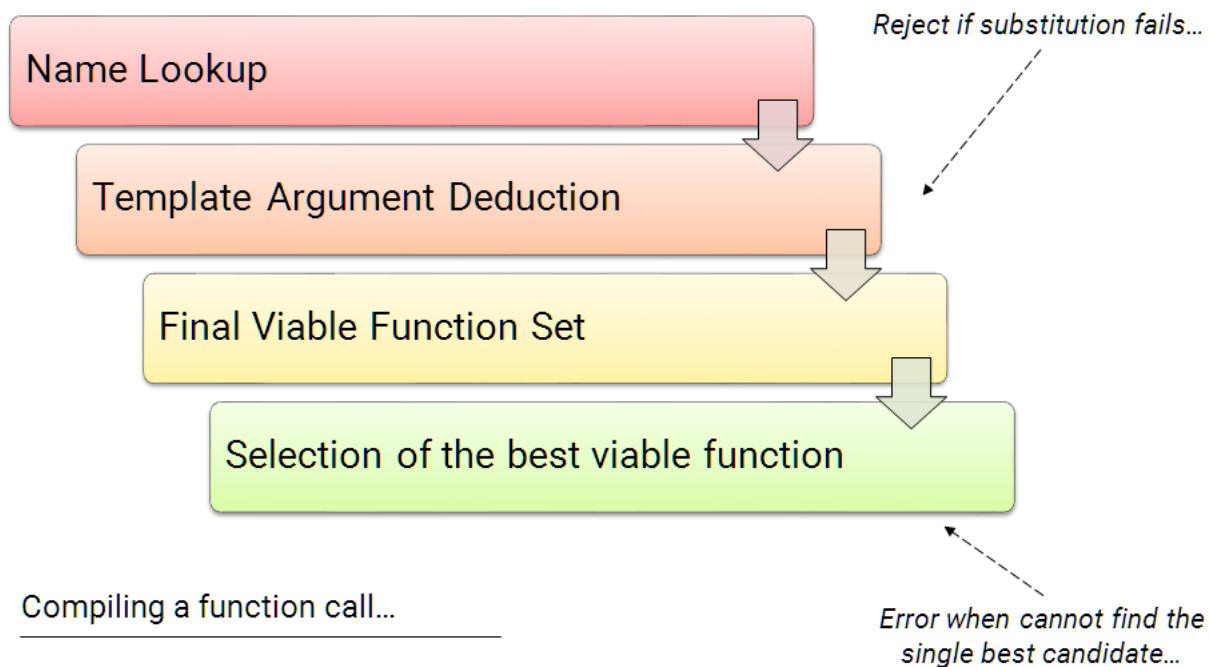
The code can be built and run. See [@Compiler Explorer](#).

Why is that?

When we added an overloaded function for the `int` type, the compiler could find a proper match and invoke the code. But in the compilation process, the compiler also ‘looks’ at the templated function header. This function is invalid for the `int` type, so why was there not even a warning reported (like we got when there was no second function provided)? To understand this, we need to look at the process of building the overload resolution set for a function call.

When the compiler tries to compile a function call (simplified):

- Perform a name lookup (see more [@CppReference](#)).
- For function templates the template argument values are deduced from the types of the actual arguments passed into the function.
 - All occurrences of the template parameter (in the return type and parameters types) are substituted with those deduced types.
 - When this process leads to invalid type (like `int::internalType`) the particular function is removed from the overload resolution set.
(**SFINAE**)
- At the end, we have a list of viable functions that can be used for the specific call.
 - If this set is empty, then the compilation fails.
 - If more than one function is chosen, we have an ambiguity.
 - In general, the candidate function, whose parameters match the arguments most closely is the one that is called.



In our example: `typename T::internalType foo(const T& t)` was not a good match for `int` and it was rejected from overload resolution set. But at the end, `int foo(int i)` was the only option in the set, so the compiler did not report any problems.

I hope you get a basic idea what SFINAE does, but where can we use this technique? A general answer: whenever we want to select a proper function/specialization for a specific type.

Some of the examples:

- Call a function when `T` has a given method (like call `toString()` if `T` has `toString` method)
- Disallow narrowing or wrong conversions from wrapper types. For example, this is used to prevent `std::variant` from deducing the wrong types. See [Everything You Need to Know About std::variant from C++17 - type conversions](#).
- [Nice example here at SO](#) of detecting count of objects passed in initializer list to a constructor.
- Specialize a function for all kind of type traits that we have (`is_integral`, `is_array`, `is_class`, `is_pointer`, etc... [more traits here](#))
- AT [Foonathan blog](#): there is an example of how to count bits in a given input number type. SFINAE is part of the solution (along with tag dispatching)
- [Another example from foonathan blog](#) - how to use SFINAE and Tag dispatching to construct a range of objects in raw memory space.

Ok, but how can we write such SFINAE expressions? Are there any helpers?

Let's meet `std::enable_if`.

One of the primary uses of SFINAE can be found through `enable_if` expressions.

`enable_if` is a set of tools, available in the Standard Library since C++11, that internally use SFINAE. They allow to include or exclude overloads from possible function templates or class template specialization.

For example:

```
// C++11:
template <class T>
```

```

typename std::enable_if<std::is_arithmetic<T>::value, T>::type
foo(T t) {
    std::cout << "foo<arithmetic T>\n";
    return t;
}

```

This function ‘works’ for all the types, that [are arithmetic](#) (int, long, float...). If you pass other types (for instance MyClass), it will fail to instantiate. In other words, template instantiations for non-arithmetic types are rejected from overload resolution sets. This construction might be used as a template parameter, function parameter or as a function return type.

enable_if<condition, T>::type will generate T, if the condition is true, or an invalid substitution if condition is false.

enable_if can be used along with [type traits](#) to provide the best function version based on the trait criteria.

Also please note that since C++14 and C++17 we have a nicer syntax and more compact. There’s no need to use ::type or ::value for enable_if or the traits, as there are _v and _t variable templates and template aliases introduced.

Our previous code can become:

```

// C++17:
template <class T>
typename std::enable_if_t<std::is_arithmetic_v<T>, T> // << shorter!
foo(T t) {
    std::cout << "foo<arithmetic T>\n";
    return t;
}

```

Please notice the use of std::enable_if_t and std::is_arithmetic_v.

See the full example:

```

#include <iostream>
#include <type_traits>

template <class T>
typename std::enable_if_t<std::is_arithmetic_v<T>, T> // << shorter!
foo(T t) {
    std::cout << "foo<arithmetic T>\n";
    return t;
}

template <class T>
typename std::enable_if_t<!std::is_arithmetic_v<T>, void>
foo(T t) {
    std::cout << "foo fallback\n";
}

int main() {
    foo(0);
    foo(std::string{});
}

```

And play [@Compiler Explorer](#).

From [@CppReference](#) - SFINAE:

Only the failures in the types and expressions in the immediate context of the function type or its template parameter types or its explicit specifier (since C++20) are SFINAE errors. If the evaluation of a substituted type/expression

causes a side-effect such as instantiation of some template specialization, generation of an implicitly-defined member function, etc, errors in those side-effects are treated as hard errors. A lambda expression is not considered part of the immediate context. (since C++20).

See a separate blog post:

C++11 has even more complicated option for SFINAE.

[n2634: Solving the SFINAE problem for expressions](#)

Basically, this document clears the specification, and it lets you use expressions inside decltype and sizeof.

For example:

```
template <class T> auto f(T t1, T t2) → decltype(t1 + t2);
```

In the above case, the expression of t1+t2 needs to be checked. It will work for two int's (the return type of the + operator is still int), but not for int and std::vector.

Expression checking adds more complexity into the compiler. In the section about overload resolution, I mentioned only about doing a simple substitution for a template parameter. But now, the compiler needs to look at expressions and perform full semantic checking.

BTW: VS2013 and VS2015 support this feature only partially ([msdn blog post about updates in VS 2015 update 1](#)), some expressions might work, some (probably more complicated) might not. Clang (since 2.9) and GCC (since 4.4) fully handle "Expression SFINAE".

SFINAE and enable_if are compelling features, but also it's hard to get it right. Simple examples might work, but in real-life scenarios, you might get into all sorts of problems:

- Template errors: do you like reading template errors generated by the compiler? Especially when you use STL types?
- Readability
- Nested templates usually won't work in enable_if statements

Here is a discussion at StackOverflow: [Why should I avoid std::enable_if in function signatures.](#)

Can we do something better?

We have at least three things:

- tag dispatching
- compile-time if
- and... Concepts!

Let's review them briefly.

This is a much more readable version of selecting which version of a function is called. First, we define a core function, and then we call version A or B depending on some compile-time condition.

```
template <typename T>
int get_int_value_impl(T t, std::true_type) {
    return static_cast<int>(t+0.5f);
```

```

}

template <typename T>
int get_int_value_impl(T t, std::false_type) {
    return static_cast<int>(t);
}

template <typename T>
int get_int_value(T t) {
    return get_int_value_impl(t, std::is_floating_point<T>{});
}

```

When you call `get_int_value` the compiler will then check the value of `std::is_floating_point` and then call the matching `_impl` function.

Compile Time if - Since C++17

Since C++17 we have a new tool, build in the language, that allows you to check the condition at compile time - without the need to write complex templated code!

In a short form we can present it:

```

template <typename T>
int get_int_value(T t) {
    if constexpr (std::is_floating_point<T>) {
        return static_cast<int>(t+0.5f);
    }
    else {
        return static_cast<int>(t);
    }
}

```

You can read more in the following blog post: [Simplify code with 'if constexpr' in C++17](#).

With each C++ Standard revision, we get much better techniques and tools to write templates. In C++20 we'll get a long-awaited feature, that will revolutionise the way we write templates!

With Concepts, you'll be able to add constraints on the template parameters and get better compiler warnings.

One basic example:

```

// define a concept:
template <class T>
concept SignedIntegral = std::is_integral_v<T> && std::is_signed_v<T>;

// use:
template <SignedIntegral T>
void signedIntsOnly(T val) { }

```

In the code above we first create a concept that describes types that are signed and integral. Please notice that we can use existing type traits. Later, we use it to define a function template that supports only types that match the concept. Here we don't use `typename T`, but we can refer to the name of a concept.

Let's now try to wrap our knowledge with an example.

To conclude my notes, it would be nice to go through some working example and see how SFINAE is utilized:

See the code [@Wandbox](#)

The test class:

```
template <typename T>
class HasToString {
private:
    typedef char YesType[1];
    typedef char NoType[2];

    template <typename C> static YesType& test(decltype(&C::ToString));
    template <typename C> static NoType& test(...);

public:
    enum { value = sizeof(test<T>(0)) == sizeof(YesType) };
};
```

The above template class will be used to test if some given type `T` has `ToString()` method or not. What we have here... and where is the SFINAE concept used? Can you see it?

When we want to perform the test, we need to write:

```
HasToString<T>::value
```

What happens if we pass `int` there? It will be similar to our first example from the beginning of the article. The compiler will try to perform template substitution, and it will fail on:

```
template <typename C> static YesType& test( decltype(&C::ToString) ) ;
```

Obviously, there is no `int::ToString` method so that the first overloaded method will be excluded from the resolution set. But then, the second method will pass (`NoType& test(...)`), because it can be called on all the other types. So here we get SFINAE! One method was removed and only the second was valid for this type.

In the end the final enum value, computed as:

```
enum { value = sizeof(test<T>(0)) == sizeof(YesType) };
```

returns `NoType` and since `sizeof(NoType)` is different than `sizeof(YesType)` the final value will be `0`.

What will happen if we provide and test the following class?

```
class ClassWithToString {
public:
    string ToString() { return "ClassWithToString object"; }
};
```

Now, the template substitution will generate two candidates: both test methods are valid, but the first one is better, and that will be *'used'*. We'll get the `YesType` and finally the `HasToString<ClassWithToString>::value` returns `1` as the result.

How to use such checker class?

Ideally it would be handy to write some if statement:

```
if (HasToString<decltype(obj)>::value)
    return obj.ToString();
else
    return "undefined";
```

We can write this code with `if constexpr`, but for the purpose of this example, let's focus on the C++11/14 solution.

To do that, we can use `enable_if` and create two functions: one that will accept classes with `ToString` and one that accepts all other cases.

```
template<typename T>
typename enable_if<HasToString<T>::value, string>::type
```

```

CallToString(T * t) {
    return t→ToString();
}

string CallToString(...) {
    return "undefined...";
}

```

Again, there is SFINAE in the code above. `enable_if` will fail to instantiate when you pass a type that generates `HasToString<T>::value = false`.

The above technique is quite complicated and also limited. For example, it doesn't restrict the return type of the function.

Let's see how Modern C++ - can help.

In one comment under the initial version of the article, [STL \(Stephan T. Lavavej\)](#) mentioned that the solution I presented in the article was from old Cpp style. What is this new and modern style then?

We can see at several things:

- `decltype`
- `declval`
- `constexpr`
- `std::void_t`
- detection idiom

Let's have a look:

`decltype` is a powerful tool that returns type of a given expression. We already use it for:

```

template <typename C>
static YesType& test( decltype(&C::ToString) ) ;

```

It returns the type of `C::ToString` member method (if such a method exists in the context of that class).

`declval` is a utility that lets you call a method on a `T` without creating a real object. In our case, we might use it to check the return type of a method:

```
decltype(declval<T>().toString())
```

`constexpr` suggests the compiler to evaluate expressions at compile time (if possible). Without that our checker methods might be evaluated only at run time. The new style suggests adding `constexpr` for most methods.

[Akrzemi1: "constexpr" function is not "const"](#)

Full video for the lecture:

[CppCon 2014: Walter E. Brown "Modern Template Metaprogramming: A Compendium, Part II" - YouTube](#)

Starting at around 29 minute, and especially around 39 minutes.

This is an amazing meta-programming pattern! I don't want to spoil anything, so just watch the video, and you should understand the idea! :)

Walter E. Brown proposes a whole utility class that can be used for checking interfaces and other properties of a given class. Of course, most of it is based on

void_t technique.

If I am correct and assuming you have void_t in your compiler/library, this is a new version of the code:

```
// default template:
template< class , class = void >
struct has_toString : false_type { };

// specialized as has_member< T , void > or sfinae
template< class T>
struct has_toString<T , void_t<decltype(&T::toString)>> : std::is_same<std::string, decltype(declval<T>().toString())> { };
```

See the code [@Wandbox](#)

Pretty nice... right? :)

It uses explicit detection idiom based on void_t. Basically, when there is no T::toString() in the class, SFINAE happens, and we end up with the general, default template (and thus with false_type). But when there is such a method in the class, the specialized version of the template is chosen. This could be the end if we don't care about the return type of the method. But in this version, we check this by inheriting from std::is_same. The code checks if the return type of the method is std::string. Then we can end up with true_type or false_type.

We can do even better in C++20. With this feature, we can declare a new concept that specifies the interface of a class:

For example:

```
template <typename T>
concept HasToString = requires(T v)
{
    {v.toString()} → std::convertible_to<std::string>;
};
```

And that's all! all written with a nice and easy to read syntax.

We can try this with some test code:

```
#include <iostream>
#include <string>
#include <type_traits>

template <typename T>
concept HasToString = requires(const T v)
{
    {v.toString()} → std::convertible_to<std::string>;
};

struct Number {
    int _num { 0 };
    std::string toString() const { return std::to_string(_num); };
};

void PrintType(HasToString auto& t) {
    std::cout << t.toString() << '\n';
}

int main() {
    Number x { 42 };
    PrintType(x);
}
```

And if your type doesn't support toString then you'll might get the following compiler error (GCC 10):

```
int x = 42;
PrintType(x);
```

And the error (a bit simplified):

error: use of function 'void PrintType(auto:11&) [with auto:11 = int]' with unsatisfied constraints

```
|      PrintType(x);
|                      ^
```

note: declared here

```
| void PrintType(HasToString auto& t) {
|      ^~~~~~
```

In instantiation of 'void PrintType(auto:11&) [with auto:11 = int]':

required for the satisfaction of 'HasToString<auto:11>' [with auto:11 = int]

in requirements with 'const int v'

note: the required expression 'v.toString()' is invalid

```
8 |      {v.toString()} → std::convertible_to<std::string>;
|      ~~~~~^~
```

We moved to an entirely new world, from some complex SFINAE code, some improvements in C++14 and C++17 to a clear syntax in C++20.

In this post, we covered theory and examples of SFINAE - a template programming technique that allows you to reject code from the overload resolution sets. In raw form, this can be a bit complicated, but thanks to modern C++ we have many tools that can help: for example `enable_if`, `std::declval` and a few others. What's more, if you're lucky to work with the latest C++ standard, you can leverage `if constexpr` from C++17 and also Concepts from C++20.

The latter - concepts - can revolutionise our template code and make it easy to read and work with!

- Where do you use SFINAE and `enable_if`?
- If you have an example of SFINAE, please let me know and share your experience!

First thing: if you have more time, please read [An introduction to C++'s SFINAE concept: compile-time introspection of a class member](#) by Jean Guegant. This is an excellent article that discusses SFINAE more deeply than I've ever found in other places. Highly recommended resource.

Thanks for comments: [@reddit/cpp thread](#)