DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Seminar Report

SFINAE, std::enable_if and Compile-Time Reflection

Daniel Below

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Seminar Report

SFINAE, std::enable_if and Compile-Time Reflection

Author:	Daniel Below
Supervisor:	PD Dr. rer. nat. habil. Tobias Lasser
Advisor:	Salvatore Virga
Submission Date:	July 14, 2018

Abstract

Templates in C++ provide a great improvement over C's `void*` when writing generic code. They provide a type-safe alternative, that is created at compile-time. This results in no run-time overhead, neither in speed nor in size of the binary, as a template is only instantiated for types it is used with. This goes hand in hand with the C++ mantra of "don't pay for what you don't use".

However, with great power comes great responsibility. Templates are generally overly permissive: developers need to mentally keep track of additional type constraints for any given template parameter. Recent efforts [7] strive for a way of specifying these constraints utilizing the type system and enforcing them at compile-time. Until these changes are introduced into the language, developers have to make do with what is available today. In this report we will cover the current state-of-the-art techniques on how to constrain template parameter types and discuss alternatives.

Contents

1 Type SFINAE

1.1 Overload Resolution

Currently, C++ developers can add constraints to template type parameters by conditionally compiling different functions. This can be done by utilizing a technique called SFINAE. SFINAE is an acronym for "Substitution Failure Is Not An Error". Before we can understand what this means in detail, we need to first look at what a substitution failure really is. Therefore, we need to understand overload resolution [3]. Overload resolution happens when there are more than one possible functions for a given call site. The compiler is required to check and verify all possible function calls.

1.1.1 Standard Function Call

```
1 void foo(int);
2 void foo(char);
3
4 foo(42);
```

Listing 1.1: Multiple candidate functions

Listing 1.1 shows an example where the compiler has more than one possible candidate function for the call site on line 4 after name lookup. In the next step, overload resolution will be performed on the list of candidate functions. There are a lot of rules that apply during overload resolution [3], but, as a simplified model, the preferred function will be chosen depending on

- best type conversion,
- normal function over function template,
- function template over variadic function and
- best specialized function template

In example 1.1 overload resolution would choose the function `void foo(int)`, as `void foo(char)` would require a type conversion.

1.1.2 Function Template Call

If we encounter a function template during name lookup, then we have to perform two additional steps prior to overload resolution. First, if the call site specifies the template parameters, we can skip template argument deduction and continue with template argument substitution. If no template parameter is explicitly stated, the compiler tries to deduce them from the given arguments. For each deduced, or explicitly stated, type parameter, we substitute it into the function template.

```
1 template<typename T>
2 void foo(const T&);
3
4 foo(42);
```

Listing 1.2: Template Argument Substitution

The function template in 1.2 will be transformed to `template<int> void foo(const int&)` during template argument substitution for the call site on line 4.

1.1.3 Substitution Failures

If we were to add another function argument, we could constrain the template type in a way, that would not be possible for other types, as 1.3 shows.

```
template<typename T>
void foo(const T&, typename T::ElemTy* = nullptr);
```

Listing 1.3: Template Constraint

Here, our template parameter is required to have a static data member type defined, called "ElemTy". Of course, this does not exist for type `int`. Still, the compiler performs template argument substitution and would encounter `template<int> void foo(const int&, typename int::ElemTy* = nullptr)`. Since `int::ElemTy` does not exist, this substitution fails. However, thanks to SFINAE, this is not a hard error during compilation, and the function template, where template argument substitution failed, will simply be discarded from the candidate list. The compiler will try to find another function to call, if such a function exists. If there is no available function, though, this will result in a hard error.
This means, providing a fallback function, it is possible to write specialized implementations for certain types, while types that do not fulfill these additional constraints will use the fallback.

2 std::enable_if

Using SFINAE we can construct a type that will allow us to guide overload resolution and discard candidate functions based on conditions known at compile-time. std::enable_if resides in the <type_traits> header and was added in C++11.

2.1 Sample Implementation

A sample implementation of std::enable_if could look like the example in 2.1

```
template<bool B, typename T = void>
struct enable_if {};

template<typename T>
struct enable_if<true, T> {
  using type = T;
}
```

Listing 2.1: Sample Implementation

The base template does not define any member types, but the partial specialization on true does. This means, if the condition evaluates to false, the substitution fails and the candidate will be discarded.

2.1.1 std::enable_if's Actual Type

The actual type of a std::enable_if expression can be thought of like this:

- typename std::enable_if<true, T>::type results in type T
- typename std::enable_if<false, T>::type is ill-formed, because ::type does not exist in this case.

C++14 has introduced a shorthand for this expression.

```
template<bool B, typename T = void>
using enable_if_t = typename enable_if<B, T>::type;
```

Listing 2.2: Shorthand for `typename std::enable_if<B, T>::type`

2.2 Using std::enable_if

There are three different places in a function where one can put `std::enable_if`. The expression can be

1. the return type,
2. an additional (unnamed) function argument with a default value or
3. an additional (unnamed) template argument with a default value

We will illustrate each with an example. Assume we want to write a function that converts an object of type `From` to type `To`. The `<type_traits>` header provides a meta-function called `std::is_convertible<From, To>`, which we will use as condition to `std::enable_if`. Note the `_v` suffix, which has been added in C++17, as a shorthand for `::value`. Listing 2.3 shows how to use `std::enable_if` as our new return type.

```
template<typename To, typename From>
std::enable_if_t<std::is_convertible_v<From, To>, To>
convert(const From&);
```

Listing 2.3: std::enable_if Return Type

The resulting type of the `std::enable_if` expression now depends on the value of `std::is_convertible<From, To>::value`. If type `From` is convertible to type `To`, this will be `true`, and `::type` will be defined on `std::enable_if`. This would instantiate the template as `template<To, From> To convert(const From&)`. If, however, the type is not convertible, then `::type` is not defined and the function will be removed from the candidate set.
The second place one can add `std::enable_if` is as an additional (unnamed, because we don't intend to use it) function argument with a default value. 2.4 shows how this can be accomplished.

```
template<typename To, typename From>
To convert(const From&,
  std::enable_if_t<std::is_convertible_v<From, To>>* = nullptr);
```

Listing 2.4: std::enable_if Function Argument

The resulting type again depends on the value of `std::is_convertible<From, To>::value`, but, if instantiated correctly, the signature will be
`template<To, From> To convert(const From&, void* = nullptr)`.

The last place to add `std::enable_if` to is as default template type parameter. 2.5 illustrates this.

```
template<typename To, typename From,
  typename = std::enable_if_t<std::is_convertible_v<From, To>>
>
To convert(const From&);
```

Listing 2.5: std::enable_if Template Argument

If convertible, this instantiates the template as
`template<typename To, typename From, typename = void> To convert(const From&)`.
However, because default template arguments are *not* part of a function template's signature, this can lead to compile errors, where the reason is not obvious. 2.6 shows code that fails to compile because of this exact error.

```
template<typename T,
  typename = std::enable_if_t<std::is_integral_v<T>>
>
void print(const T&);

template<typename T,
  typename = std::enable_if_t<std::is_floating_point<T>>
>
void print(const T&);
```

Listing 2.6: Same Signature Error

Since default template arguments are not part of the function template's signature, both function templates in 2.6 have the exact same signature. This leads to a compile-time error.

3 The <type_traits> Header

The <type_traits> header [4] has been added to the language with C++11. It defines a lot of different meta-functions that can be used to query information from the type system. In the previous chapter we have already seen one of these meta-functions: `std::is_convertible<From, To>`.
It is also possible to modify types, but first let's have a look at the building blocks of all meta-functions. The <type_traits> header defines a struct called `std::integral_constant`.

3.1 std::integral_constant

`std::integral\_constant` is a simple struct with a one data member.

```cpp
template<typename T, T v>
struct std::integral_constant {
  static constexpr T value = v;
};
```

Listing 3.1: std::integral_constant

With this struct, it is now possible to define `std::true_type` and `std::false_type`. These are full specializations of `std::integral_constant`:

- `using std::true_type = std::integral_constant<bool, true>;`
- `using std::false_type = std::integral_constant<bool, false>;`

We now have two types that represent the boolean values of `true` and `false`. These should not be confused with the primitive type `bool`, because both `std::true_type` and `std::false_type` only hold `true` and `false` respectively, whereas a `bool` could be either. Also, because they are full specializations of `std::integral_constant`, both `std::true_type` and `std::false_type` have a `::value` data member.

3.2 Implementing a Type Trait

Using these structs we can now implement every type trait. As an example, listing 3.2 shows how to implement a type trait that checks whether `T` is a reference type.

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

template<typename T>
struct is_reference<T&> : std::true_type {};

template<typename T>
struct is_reference<T&&> : std::true_type {};
```

Listing 3.2: Type Trait Implementation

The base template catches all types that do not fit one of the specializations and derives from `std::false_type`. The two specializations catch both lvalue- and rvalue-references and derive from `std::true_type`.

3.2.1 Predefined Type Traits

Most pre-defined type traits are self-explanatory. For example, the trait `std::is_integral<T>` is `true` when `T` is an integral value, and false otherwise. `std::is_member_object_pointer<T>` is `true` when `T` is a pointer to a non-static member object, and false otherwise. Other traits can be used to instantiate different types, such as `std::enable_if` and `std::conditional`. The latter is interesting, because it allows the compiler to pick a different type depending on a compile-time condition. 3.3 shows how to use this.

```
template<typename T>
std::conditional<std::is_integral_v<T>, int, float>
do_something(const T&);
```

Listing 3.3: std::conditional

In 3.3, the return type of the function is `int`, if `std::is_integral_v<T>` evaluates to `true`, and `float` otherwise.

4 Expression SFINAE

Expression SFINAE, like type SFINAE, let's developers write code that may be invalid for certain type substitutions. However, unlike SFINAE, this does not only work for types, but for whole expressions. There are three prerequisites we have to understand first:

1. the comma operator,
2. trailing return type syntax and
3. `std::devlcal<T>()` and `decltype`

4.1 The Comma Operator

A comma operator expression has the form `E1, E2` [2]. In this expression, `E1` will be evaluated and its result discarded (and its side-effects completed) before evaluation of `E2` begins. The type, value and value category of the result of the comma operator expression are exactly those of `E2`.

Listing 4.1 illustrates how the comma operator works.

```
// equivalent to i = 2
const auto i = (5, 2);

// works for different types and more commas, too
const auto j = (printf("Hello, World!\n"),
       other_call(),
       one_more_call(),
       2);
```

Listing 4.1: Comma Operator

In the example, parenthesis are necessary, because the assignment operator has higher precedence.

4.2 Trailing Return Type

Sometimes we don't know what the return type will be when writing generic code. The trailing return type syntax can solve this problem.

```
template<typename T, typename U>
auto add(const T& t, const U& u) -> decltype(t + u) { ... }
```

Listing 4.2: Trailing Return Type

In 4.2 we use the trailing return type syntax in combination with `decltype` to retrieve the type of the addition of an object of type `T` with an object of type `U`. In this case, the `decltype` expression is simple, because we can use the named function arguments. However, there can be cases, where we don't have this information. For those cases, we must rely on `std::declval<T>()`.

4.3 std::declval<T>()

`std::declval<T>()` is used to "create" an rvalue-reference of type `T`. The type does not need to have a constructor available, and we don't need to specify one. We are basically telling the compiler "assume you had an object of this type".

```
template<typename T, typename U>
auto sum(const std::vector<T>&, const std::vector<U>&)
  -> decltype(std::declval<T>() + std::declval<U>()) { ... }
```

Listing 4.3: Using std::declval<T>()

In the example of listing 4.3 we don't have an instance of type `T` or `U`. Here, we can use `std::declval<T>()` to "create" the instances to perform the addition on.

4.4 Expression SFINAE

Putting the three things together, we can use expression SFINAE to have a function only available, if the type `T` has a `serialize()` member function (see listing 4.4).

```
template<typename T>
auto do_something(const T& t) -> decltype(t.serialize(), void()) { ... }
```

Listing 4.4: Checking serialize() member function

Here, `t.serialize()` will be evaluated, but `decltype` will be applied to `void()`. This can only be successful, if such a member function exists. If it doesn't, expression SFINAE will discard this function from the candidate set. Since the comma operator can be chained, and only the right-most expression will be used, it is possible to check the availabilty of more than one member function. This way, e.g. as library maintainers, we can verify that passed types provide certain functionalities.

4.4.1 Converting to Type Trait

The previous example of checking whether a type defines a specific member function can be extracted and converted into a type trait. This way, we can apply type SFINAE without having to rely on `decltype` and the trailing return type syntax. 4.5 shows how this type trait might be implemented.

```
template<typename T, typename = std::string>
struct has_serialize : std::false_type {};

template<typename T>
struct has_serialize<T, decltype(std::declval<T>().serialize())>
       : std::true_type {};

template<typename T>
inline constexpr bool has_serialize_v = has_serialize<T>::value;
```

Listing 4.5: has_serialize type trait

Here, the result of the `decltype` expression will be used for the unnamed type parameter of the base template. If the expression fails, so if `T` does not have a `serialize()` member function, then we fall back to the base template and derive from `std::false_type`. If the member function exists we take the specialization and derive from `std::true_type`. To be consistent with the C++17 shorthand, we also add a `has_serialize_v` variable.

5 Alternatives

There are a couple alternatives that are usually preferred over direct use of SFINAE [**cppref-sfinae**]. We will shortly discuss

1. `static_assert`,
2. `if constexpr` and
3. tag dispatch

Another alternative will be added with concepts [7], however, as they have not yet been released, we will not discuss them here.

5.1 static_assert

If our only goal is to restrict template arguments to types that have a certain property, we can simply add a `static_assert` to the beginning of the function. There is really no need to use SFINAE here, if we do not plan on adding an implementation for other types. Listing 5.1 shows how `static_assert` can be used to constrain template arguments.

```
template<typename T>
void do_something(const T& t) {
  static_assert(std::is_integral_v<T>,
    "T␣is␣not␣an␣integral␣value!");

  // actual implementation
}
```

Listing 5.1: static_assert

Here we statically assert that any `T` passed into this function is an integral type. If we pass in a different `T`, it will trigger a compilation error at the location of the `static_assert`. This is a small drawback compared to direct use of SFINAE, where error messages point to the call site. Here, the error message will point to the `static_assert`.

5.2 if constexpr

Another valid alternative is `if constexpr`. This construct was added in C++17 and evaluates the condition during compile-time. If we intend to differentiate between a few types only, where the return type is equal for all implementations, we can easily use this feature, as shown in listing 5.2.

```
template<typename T>
void do_something(const T& t) {
  if constexpr (std::is_integral_v<T>) {
    // T is an integral value
  } else {
    // T is not an integral value
  }
}
```

Listing 5.2: if constexpr

It is noteworthy that this does not work for the absence of specific member functions, as `if constexpr` requires both branches to compile without error.

5.3 Tag Dispatch

The last alternative we will look at is called tag dispatch. The idea with this is to have empty structs that we use as function arguments in order to guide overload resolution. For example, we can use `std::true_type` and `std::false_type`. Listing 5.3 shows how to use tag dispatch to pick different implementations depending on whether `T` is an integral type.

```
template<typename T>
void do_something(const T& t) {
  do_something_impl(t, std::is_integral<T>{});
}

template<typename T>
void do_something_impl(const T& t, std::true_type) { ... }

template<typename T>
void do_something_impl(const T& t, std::false_type) { ... }
```

Listing 5.3: Tag Dispatch

The two hidden implementations (`do_something_impl`) both take an additional argument. We do not give them a name, because we do not intend to use them. In the `do_something` function, we instantiate the resulting type of the `std::is_integral<T>` expression, which derives from `std::true_type` if `T` is an integral type, or from `std::false_type` if `T` is something else. This way, overload resolution will pick the correct implementation, based on the function's argument list.

Tag dispatch is used in the standard library e.g. for `std::distance`, to dispatch to different implementations depending on the iterator category passed as arguments. For vector iterators, for example, we can provide an implementation that runs in constant time, whereas map iterators require linear time. Using tag dispatch we can have the same, public-facing API for all iterator categories without missing out on the specialized implementation for some of them.

Bibliography

[1] E. Bendersky. *SFINAE and enable_if*. URL: https://eli.thegreenplace.net/2014/sfinae-and-enable_if.

[2] cppreference.com. *Other Operators*. URL: https://en.cppreference.com/w/cpp/language/operator_other.

[3] cppreference.com. *Overload Resolution*. URL: https://en.cppreference.com/w/cpp/language/overload_resolution.

[4] cppreference.com. *Standard Library Header <type_traits>*. URL: https://en.cppreference.com/w/cpp/header/type_traits.

[5] J. Guegant. *Compile time type introspection using SFINAE*. URL: https://www.youtube.com/watch?v=YgbTBqS-bHg.

[6] A. O'Dwyer. *A Soupcon of SFINAE*. URL: https://www.youtube.com/watch?v=ybaE9qlhHvw.

[7] A. Sutton. *C++ Extensions for Concepts*. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4549.pdf.