SFINAE: Failure is an Option

Jeff Cohen — ACCU 02/12/2020

What is SFINAE?

- Substitution Failure Is Not An Error
- A powerful template metaprogramming tool
- An *accidental* metaprogramming tool

What is substitution?

```
struct B {
  using type = int;
};

template<typename T>
void foo(T& p1, typename T::type p2) { }

void bar() {
  B v;
  foo(v, 10);
}
```

In the call of foo, the template parameter T is first deduced to be B.

Having been deduced, int is *substituted* for T::type.

What is substitution failure?

```
struct B {
};

template<typename T>
void foo(T& p1, typename T::type p2) { }

void bar() {
  B v;
  foo(v, 10);
}
```

```
note: candidate template ignored:
substitution failure [with T = B]:
no type named 'type' in 'B'
```

If the member type is removed from the struct, substitution fails.

But what happened to “substitution failure is not an error?”

Why ignore substitution failures?

```
struct A {
};

struct B {
  using type = int;
};

void foo(A& p1, int p2) { }

template<typename T>
void foo(T& p1, typename T::type p2) { }

void bar() {
  A v1;
  foo(v1, 10);  // no substitution error!

  B v2;
  foo(v2, 10);
}
```

Struct A would fail substitution, as it lacks type. But there's an overload of foo that takes it as an argument, so that substitution error is suppressed.

Does this mean no attempt was made to use the template? After all, the other overload matched the argument types exactly.

No. A attempt is made to use all template overloads of the function. When a template fails type deduction or substitution, that error is ignored if (and only if!) a different overload is successfully picked.

In other words, without SFINAE, this code would not compile, reducing the utility of template functions.

What about Metaprogramming?

- SFINAE was never designed to be a metaprogramming tool. It was introduced to prevent spurious errors from making template functions too painful to use.

- It was an accidental discovery that it *could* be used for metaprogramming, and a powerful tool it turned out to be.

Example: enable_if

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

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



template<typename T, typename =
     typename enable_if<some_test>::type>
void foo(T&& x) {
}
```

enable_if is a struct that defines a member type if and only if Cond is true.

In this example, that member is allowed to default to void, as the specific type does not matter. It only matters whether it *has* a type.

When the test is false, the substitution failure happens in the default value for the second template argument (allowed since C++11). There's no need to give this argument a name, as it only matters whether it successfully gets a value.

Example: test for method

```
template<typename T>
struct has_XYZ {
  // Produces a substitution failure if U does
  // not possess a suitable method signature.
  template<typename U, void (U::*)()>
  struct CheckSignature;

  // Overload resolution will pick the first if the
  // method is present; otherwise, substitution
  // failure will leave the second as the only
  // choice.
  template<typename U>
  static char test(CheckSignature<U, &U::XYZ> *);

  template<typename U>
  static long test(...);

  // Convert answer (the return type) to bool.
  enum { value =
    sizeof(test<T>(nullptr)) == sizeof(char) };
};
```

has_XYZ is a struct that answers the question, does T have a member method XYZ with no parameters?

Whether it does will determined which overload of test is selected. We can determine which overload that was by testing the size of the selected overload's return type.

Note that no definitions are provided for any overload of test. These helper functions are never actually called. Likewise CheckSignature does not need a definition, as it is never instantiated, nor is any member accessed.

Example: even/odd functions

```
template<int I>
void foo(char(*)[I % 2 == 0] = nullptr) {
  // Substitution succeeds when I is even.
}

template<int I>
void foo(char(*)[I % 2 == 1] = nullptr) {
  // Substitution succeeds when I is odd.
}
```

The substitution failure here occurs when an array of zero elements is declared.

The argument to foo is otherwise being ignored. It defaults to nullptr; it doesn't even have a name.

The Gory Details: Where

```
template<typename T,
         typename U = typename T::here>
typename T::here
foo(typename T::here x) { return x; }

// C++11
template<typename T, typename U>
auto foo(T&& t, U&& u) -> decltype(t + u)
{ return t + u; }

// C++20
struct S {
  template<typename T>
  explicit(!T::flag) S(const T&&) { }
};
```

Substitution occurs in the function type, i.e. the return type and function parameters. It also occurs in template parameter declarations (only class templates may have default values pre-C++11).

As of C++11, it also includes all expressions used in decltype and sizeof.

As of C++20, it also includes all expressions used in an explicit specifier.

Substitution proceeds in lexical order, left to right, and stops if and when a failure is encountered.

SFINAE does not apply to function bodies!

The Gory Details: Type Errors

- Attempting to use a member of a type, where
  - the type does not contain the specified member
  - the specified member is not a type when a type is required
  - the specified member is not a template when a template is required
  - the specified member is a type when a non-type is required
- Attempting to create a pointer to a reference.

The Gory Details: Type Errors

- Attempting to create a reference to void.
- Attempting to create a pointer-to-member of T, where T is not a class type.
- Attempting to give an invalid type to a non-type template parameter.
- Attempting to perform an invalid conversion in
  - a template argument expression
  - An expression used in a function type declaration

The Gory Details: Type Errors

- Attempting to create a function type
  - with a parameter of type void
  - which returns an array type or a function type
- Attempting to create a cv-qualified function type (until C++11).
- Attempting to instantiate a pack expansion containing multiple packs of different lengths (since C++11).

The Gory Details: Expression Errors

- Ill-formed expression used in
  - a template parameter type
  - a function type (parameter types and return type)
- Before C++11, only constant expressions used in types (such as array bounds) were required to be treated as SFINAE.

Mistakes to Avoid

```
template<typename T,
         typename U = typename T::here>
class C {
};
```

SFINAE does not apply to class template parameter defaults.

Mistakes to Avoid

```
template<typename T>
class C {
  template<typename U>
  void func(U&& u, typename T::here &&t) {
  }
};
```

SFINAE does not apply to anything dependent on a class template parameter.

Mistakes to Avoid

```
template<typename T>
void func() {
  typename T::here x;
}
```

SFINAE does not apply to the body of a function.

Mistakes to Avoid

```
template<typename T>
int func(T& t, int x = T::here) {
  return x;
}

// But this will work:

template<typename T>
int func(T& t,
         decltype(T::here) x = T::here) {
  return x;
}
```

SFINAE does not apply to parameter default values.

The Future of SFINAE

- For better or worse, it has a future—if for no other reason than there's a lot of code out there that relies upon it.
- But should you use it? SFINAE's use as a metaprogramming tool was a "lucky" discovery. It leaves much to be desired.
- But what are the alternatives? Why can't we have metaprogramming facilities that were designed as such?
- Finally, we can!

C++20 Concepts

Old Way:

```
template<typename T>
struct has_XYZ {
  template<typename U, void (U::*)()>
  struct CheckSignature;

  template<typename U>
  static char test(CheckSignature<U, &U::XYZ> *);

  template<typename U>
  static long test(...);

  enum { value =
    sizeof(test<T>(nullptr)) == sizeof(char) };
};

template<typename T, typename = typename
        enable_if<has_XYZ<T>::value>::type>
void foo(T& t) { t.XYZ(); }
```

New Way:

```
template<typename T>
concept HasXYZ = requires(T& a) {
    { a.XYZ() } -> void;
};


template<HasXYZ T>
void foo(T& t) { t.XYZ(); }
```

And Beyond

- Do concepts eliminate all use of SFINAE? Unfortunately not. But where it can replace it, it is clearly simpler, cleaner, and far easier to understand.

- if constexpr can also occasionally replace SFINAE. By allowing conditional compilation to handle type-specific differences, several overloaded template functions can be merged into one. It can do even more once compile-time reflection is added in (hopefully) C++23.

- But let's face it: SFINAE will never go away. So learn to live with it and harness its power!

Advanced SFINAE

```
template<typename T> class is_valid_helper {
  template<typename U> constexpr auto test(int)
      -> decltype(declval<T>()(declval<U>()), true_type())
    { return true_type(); }

  template<typename U> constexpr false_type test(...)
    { return false_type(); }

public:
  template<typename U> constexpr auto operator()(const U &p)
    { return test<U>(int()); }
};

template<typename T> constexpr auto is_valid(const T &t)
  { return is_valid_helper<T>(); }

auto hasXYZ =
   is_valid([](auto &&x) -> decltype(x.XYZ()) { });

template<typename T> auto func(T &obj)
    -> enable_if_t<decltype(hasXYZ(obj))::value>
  { obj.XYZ(); }

template<typename T> auto func(T &obj)
    -> enable_if_t<!decltype(hasXYZ(obj))::value>
  { /* Do something else. */ }
```

This example (C++14 compliant) is equivalent to the previous concept example: it'll accept a method whose parameters all have default values.

The key is the lambda. It's equivalent to a concept's requires clause. What's being required is what's in the decltype.

The type of this lambda is an unnamed struct with a template operator() method. Because the type of the lambda's argument is deduced, SFINAE is deferred until the lambda is "used."

(It's never actually called; this template method is never instantiated, so the fact that it returns no value is not an error.)

Advanced SFINAE

```
template<typename T> class is_valid_helper {
  template<typename U> constexpr auto test(int)
      -> decltype(declval<T>()(declval<U>()), true_type())
    { return true_type(); }

  template<typename U> constexpr false_type test(...)
    { return false_type(); }

public:
  template<typename U> constexpr auto operator()(const U &p)
    { return test<U>(int()); }
};

template<typename T> constexpr auto is_valid(const T &t)
  { return is_valid_helper<T>(); }

auto hasXYZ =
   is_valid([](auto &&x) -> decltype(x.XYZ()) { });

template<typename T> auto func(T &obj)
    -> enable_if_t<decltype(hasXYZ(obj))::value>
  { obj.XYZ(); }

template<typename T> auto func(T &obj)
    -> enable_if_t<!decltype(hasXYZ(obj))::value>
  { /* Do something else. */ }
```

The function is_valid computes a type that answers the question of whether its lambda argument, when that lambda is itself provided with a type as an argument (the deduced type of x), is valid.

This computed type, is_valid_helper<T>, then answers the question by passing its type argument, from its operator() method, to the lambda.

Remember that these functions are neither compiled to object code nor executed; they're not even interpreted inside the compiler. The computation takes place solely in the type system.

Advanced SFINAE

```
template<typename T> class is_valid_helper {
  template<typename U> constexpr auto test(int)
      -> decltype(declval<T>()(declval<U>()), true_type())
    { return true_type(); }

  template<typename U> constexpr false_type test(...)
    { return false_type(); }

public:
  template<typename U> constexpr auto operator()(const U &p)
    { return test<U>(int()); }
};

template<typename T> constexpr auto is_valid(const T &t)
  { return is_valid_helper<T>(); }

auto hasXYZ =
   is_valid([](auto &&x) -> decltype(x.XYZ()) { });

template<typename T> auto func(T &obj)
    -> enable_if_t<decltype(hasXYZ(obj))::value>
  { obj.XYZ(); }

template<typename T> auto func(T &obj)
    -> enable_if_t<!decltype(hasXYZ(obj))::value>
  { /* Do something else. */ }
```

Here's where SFINAE does its magic. If substitution succeeds, the first definition is selected; a method whose parameter list is ... is selected only if nothing else qualifies.

The return type of the selected method indicates which one was selected. std::true_type and std::false_type are used here.

std::declval is used to turn deduced types into something suitable for accessing a member. The short version is that the argument to is_valid is being passed to the lambda in the context of decltype, which is not an evaluation context. Hence, the template operator() is not instantiated, suppressing any pointless errors that might cause.

Bottom line: if decltype(x.XYZ()) is ill-formed, test(...) wins.

Advanced SFINAE

```
template<typename T> class is_valid_helper {
  template<typename U> constexpr auto test(int)
      -> decltype(declval<T>()(declval<U>()), true_type())
    { return true_type(); }

  template<typename U> constexpr false_type test(...)
    { return false_type(); }

public:
  template<typename U> constexpr auto operator()(const U &p)
    { return test<U>(int()); }
};

template<typename T> constexpr auto is_valid(const T &t)
  { return is_valid_helper<T>(); }

auto hasXYZ =
   is_valid([](auto &&x) -> decltype(x.XYZ()) { });

template<typename T> auto func(T &obj)
    -> enable_if_t<decltype(hasXYZ(obj))::value>
  { obj.XYZ(); }

template<typename T> auto func(T &obj)
    -> enable_if_t<!decltype(hasXYZ(obj))::value>
  { /* Do something else. */ }
```

hasXYZ(obj) has either the type std::true_type or std::false_type. This can be used with std::enable_if to select an implementation that can handle the presence or absence of a member.

Questions