SFINAE Functionality Is Not Arcane Esoterica

(Substitution Failure Is Not An Error FINAE)

Jonathan Wakely

Sfin-wut?

Substitution Failure Is Not An Error

- Term coined by David Vandevoorde for the *C++ Templates: The Complete Guide* book with Nicolai Josuttis
- Describes an important property of template argument deduction
- ... which can be used for clever metaprogramming tricks
- ... and which have become much more powerful in C++11

Template argument deduction

[temp.deduct]/1

> When a function template specialization is referenced, all of the template arguments shall have values. The values can be explicitly specified or, in some cases, be deduced from the use or obtained from default *template-arguments*.

When you call a function template the compiler *deduces* the value of each template parameter:

```
template<class T>
  T doNothing(T t) { return t; }

int i = doNothing(1);           // T deduced as int
double f = doNothing(1.0f);     // T deduced as float
double f = doNothing<long>(1u); // T deduced as long
```

Template argument substitution

[temp.deduct]/6

> At certain points in the template argument deduction process it is necessary to take a function type that makes use of template parameters and replace those template parameters with the corresponding template arguments. This is done at the beginning of template argument deduction when any explicitly specified template arguments are substituted into the function type, and again at the end of template argument deduction when any template arguments that were deduced or obtained from default arguments are substituted.

```
template<class T>
  T doNothing(T t) { return t; }

int i = doNothing(1);  // int doNothing(int)
```

Type deduction fails

[temp.deduct]/8

> **If a substitution results in an invalid type or expression, type deduction fails.**

```
template<class T>
  typename T::iterator
  begin(T& t) { return t.begin(); }

int i=0;
int* iter = begin(i);  // substitution failure
```

Substituting `int` for `T` result in an invalid type,`int::iterator`, so deduction fails.

Why is it not an error?

Substitution has to happen before overload resolution, so the compiler knows which function template specializations are candidates in the overload set.

```
template<class T>
  typename T::iterator
  begin(T& t) { return t.begin(); }

template<class T, size_t N>
  T*
  begin(T (&array)[N]) { return array; }

int a[2];
int* b = begin(a);
```

SFINAE

When type deduction fails because substitution produces an invalid type it simply means that function template specialization **is not a candidate in the overload set**.

Substitution failure is not an error, i.e. the program is not ill-formed, because there might be other overloads (non-templates, or function template specializations with successfully deduced and substituted arguments) that can be used.

If there are no other viable overloads *that* is still an error as normal, only substitution failures during type deduction are not "hard errors".

> I made up the term to facilitate the explanation of the associated techniques in "C++ Templates": At the time I thought the standard term "deduction failure" emphasized "failure" too much, and would therefore be confusing ("So you mean this program fails?" "No, no, failure is a good thing here. It's not really _failure_... well it is, but...").

Controlling the overload set

Anything a C++ compiler has to do for good can also be used for evil.

Usually by Boost.

By intentionally triggering a substitution failure you can control which function templates are part of the overload set.

Enter `boost::enable_if`.

boost::enable_if

The implementation is trivial:

```
template<bool Cond, class T = void>
  struct enable_if_c
  { typedef T type; };

template<class T>
  struct enable_if_c<false, T>
  { };

template<class Cond, class T = void>
  struct enable_if : enable_if_c<Cond::value, T>
  { };
```

boost::enable_if

This allows you to ensure noone calls your function template with a type that you can't or don't want to support:

```
template<class T>
  typename enable_if<is_floating_point<T>, T>::type
  frobnicate(T t)
  { … }
```

The first template argument to `enable_if` can be any compile-time boolean value, so any property of a type that you can test directly or via a type trait like `is_floating_point`.

boost::enable_if

The C++03 idiom is to use `enable_if` on the return type:

```
template<class T>
  typename enable_if<is_floating_point<T>, T>::type
  frob(T t)
  { … }
```

Or to add a dummy function parameter with a default argument:

```
template<class T>
  T
  frob(T t,
       typename enable_if<is_floating_point<T> >::type* = 0)
  { … }
```

Type traits

SFINAE also makes it possible to define custom type traits, by referring to a possibly-invalid type in a template argument deduction context.

```
template<class T>
  true_type   is_iter(typename T::iterator_category*);
template<class T>
  false_type  is_iter(...);

template<class T>
  struct is_iterator
  {
    static const bool value
      = sizeof(is_iter<T>(0)) == sizeof(true_type);
  };
```

Describe typical C++03 implementations of true_type / false_type.

In practice the `is_iter` functions would be private static members of the trait class.

C++03 Limitations

In C++03 the reasons that cause type deduction to fail are quite limited, anything except forming an invalid type or attempting invalid conversions such as `(int*)1` is a hard error not a deduction failure.

Access checking is done *after* type deduction, so your trait might use SFINAE to find that `T::iterator_category` does exist, so deduction succeeds, but then you get an access error because the type is private.

Access checking much bigger issue with Expression SFINAE, can't check if type is copy-constructible if it has a private copy constructor!

C++11 power-up

C++11 makes a number of changes that make SFINAE far more powerful and easier to use

- `std::enable_if` and dozens of standard type traits are defined in the `<type_traits>` header
- "Expression SFINAE" allows arbitrary expressions to be used in argument deduction contexts
  - The `decltype` keyword makes it much easier to query and work with types
  - Late-specified return types allow function arguments to be referred to

enable_if has different signature, no disable_if / lazy / etc.

C++11 power-up (continued)

- Access checking is part of deduction, so referring to private members causes deduction to fail.
- Function templates are allowed to have default template arguments
- Alias templates greatly simplify syntax

`std::enable_if`

In contrast to the Boost version, `std::enable_if` takes a `bool` as its first argument, not a "metafunction" with a `value` member

- Equivalent to `boost::enable_if_c` rather than `boost::enable_if`.

Standard `std::true_type` and `std::false_type` types too.

- They have the same size, so test their `static const bool value` member, not size

SFINAE for Expressions

Instead of creating a type trait that tests the property you care about you and passing it to `enable_if` you can just write code:

```
template<class T>
  auto
  begin(T& t) -> decltype(t.begin())
  { return t.begin(); }
```

`decltype` is an unevaluated context, like `sizeof`, so you can write arbitrary expressions using the function arguments.

Invalid expressions result in substitution failure.

Default template arguments

C++11 allows function templates to have default template arguments, which can be very useful when there is no return type to add `enable_if` to (i.e. for constructors), when you don't want to or can't add a default function argument, or just to remove clutter from the function signature:

```
class Adaptor {
  template<class T, class Iter = typename T::iterator>
    Adaptor(const T& t)
    { ... }
    ...
};
```

Example - C++03 Style

```
public:
  // requires T is derived from WidgetBase
  template<class T>
    WidgetDecorator(const Ptr<T>& p,
                    typename boost::enable_if<
                      boost::is_base_of<WidgetBase, T>
                    >::type* = 0)
    : ptr(p) { }
```

Example - C++03 Style

```
private:
  template<class T,
           bool = boost::is_base_of<WidgetBase, T>::value>
    struct IsDerivedWidget
    { typedef void type; };

  template<class T>
    struct IsDerivedWidget<T, false>
    { };

public:
  // requires T is derived from WidgetBase
  template<class T>
    WidgetDecorator(const Ptr<T>& p,
                    typename IsDerivedWidget<T>::type* = 0)
    : ptr(p) { }
```

Example - Warming Up

```
private:
  template<class T>
    struct IsDerivedWidget
    : std::enable_if<std::is_base_of<WidgetBase, T>::value>
    { };

public:
  // requires T is derived from WidgetBase
  template<class T,
           class Requires = typename IsDerivedWidget<T>::type>
    WidgetDecorator(const Ptr<T>& p)
    : ptr(p) { }
```

Example - C++11 Hotness

```
private:
  template<class T>
    using IsDerivedWidget = typename
      std::enable_if<std::is_base_of<WidgetBase, T>::value>
      ::type;

public:
  // requires T is derived from WidgetBase
  template<class T,
           class Requires = IsDerivedWidget<T>>
    WidgetDecorator(const Ptr<T>& p)
    : ptr(p) { }
```

Example - SFINASH

```
private:
  template<class T, bool B = T::value>
    struct If { typedef void enabled; };

  template<class T>
    struct If<T, false> { };

  template<class T>
    struct IsDerivedWidget : std::is_base_of<WidgetBase, T>
    { };

public:
  template<class T,
           class Requires = typename If<IsDerivedWidget<T>>::enabled>
    WidgetDecorator(const Ptr<T>& p)
    : ptr(p) { }
```

Gotchas

- SFINAE only works for function templates, not member functions of class templates
- `enable_if` conditions must be disjoint
- Substitution failures must be in the "immediate context"

Immediate context

```
template<class It>
  typename std::iterator_traits<It>::reference
  dereference(It it)
  { return *it; }

int dereference(int i) { return i; }

auto x = dereference(1);
```

Is this code valid?

Immediate context

```
template<class It>
  typename std::iterator_traits<It>::reference
  dereference(It it)
  { return *it; }

int dereference(int i) { return i; }

auto x = dereference(1);
```

It depends

It's not guaranteed to work, and not portable

Immediate context

The problem is in the definition of `iterator_traits`:

```
namespace std {
  template<class Iter>
    struct iterator_traits {
      typedef Iter::difference_type   difference_type;
      typedef Iter::value_type        value_type;
      typedef Iter::reference         reference;
      typedef Iter::pointer           pointer;
      typedef Iter::iterator_category iterator_category;
    };
  template<class T>
    struct iterator_traits<T*>
    { … };
}
```

Immediate context

```
template<class It>
  typename std::iterator_traits<It>::reference
  dereference(It it)
  { return *it; }

auto x = dereference(1);
```

To do argument substitution the compiler must first instantiate `std::iterator_traits<int>`, then check if it has a nested `reference` member.

The instantiation is invalid, because it refers to `Iter::reference` in a non-SFINAE context.

Immediate context

My mental model is to imagine the compiler adding an explicit instantiation of the types that will be instantiated by argument deduction:

```
template struct iterator_traits<int>;

template<class It>
  typename std::iterator_traits<It>::reference
  dereference(It it)
  { return *it; }

auto x = dereference(1);
```

When such explicit instantiations would compile any references to invalid types or expressions in the function template declaration are in the *immediate context* and SFINAE applies.

Slides finishing is not an error

The XHTML source for these slides is available from http://gitorious.org/wakelyaccu/sfinae