Unevaluated Operands
The SFINAE you don't expect

skypjack

November 24, 2018

Table of contents

Some notes

- Class template: the more specialized, the preferred one.

```
template<typename T, typename = void>
struct S;

template<typename T>
struct S<T, std::enable_if_t<std::is_integral_v<T>>> {
  // ...
};
```

- SFINAE: **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**error

  *If a substitution results in an invalid type or expression, type deduction fails. [...] Only invalid types and expressions in the **immediate context** of the function type and its template parameter types can result in a deduction failure.*

- Immediate context: *soft errors* vs *hard errors*

```
template<typename T>
struct S {
  std::enable_if_t<std::is_integral_v<T>>
  f(T) { /* ... */ }
};
```

Unevaluated... what?

The beauty of the standard:

> *In some contexts, unevaluated operands appear [...].*
> *An unevaluated operand is not evaluated.*

Waiting for C++20

`typeid`, `sizeof`, `noexcept`, `decltype`

decltype

Some simple requirements:

1. If `T` has member function `f` then invoke it.

Detection idiom and SFINAE

```
template<typename T, typename = void>
struct has_f: std::false_type {};

template<typename T>
struct has_f<T, std::void_t<decltype(std::declval<T>().f())>>
: std::true_type {};

template<typename T>
std::enable_if_t<has_f<T>>
invoke() { /* ... */ }

template<typename T>
std::enable_if_t<!has_f<T>>
invoke() { /* ... */ }
```

decltype

Some simple requirements:

1. If `T` has member function `f` then invoke it.
2. If `T` has member function `g` then invoke it, otherwise 1.

One more detector and ambiguous calls

```
template<typename T>
std::enable_if_t<has_g<T>>
invoke() { /* ... */ }

template<typename T>
std::enable_if_t<!has_g<T> and has_f<T>>
invoke() { /* ... */ }

template<typename T>
std::enable_if_t<!has_f<T> and !has_g<T>>
invoke() { /* ... */ }
```

decltype

Some simple requirements:

1. If `T` has member function `f` then invoke it.
2. If `T` has member function `g` then invoke it, otherwise 1.
3. If `T` has member function `h` then invoke it, otherwise 2.

It's obviously getting a mess...

```
template<typename T>
std::enable_if_t<has_h<T>>
invoke() { /* ... */ }

template<typename T>
std::enable_if_t<has_g<T> and !has_h<T>>
invoke() { /* ... */ }

template<typename T>
std::enable_if_t<!has_g<T> and !has_h<T> and has_f<T>>
invoke() { /* ... */ }

template<typename T>
std::enable_if_t<!has_f<T> and !has_g<T> and !has_h<T>>
invoke() { /* ... */ }
```

decltype

Some simple requirements:

1. If `T` has member function `f` then invoke it.
2. If `T` has member function `g` then invoke it, otherwise 1.
3. If `T` has member function `h` then invoke it, otherwise 2.

`if constexpr`?

```
template<typename T>
void invoke() {
  if constexpr(has_h<T>) {
    /* ... */
  } else if constexpr(has_g<T> and !has_h<T>) {
    /* ... */
  } else if constexpr(has_f<T> and !has_g<T> and !has_h<T>) {
    /* ... */
  } else {
    /* ... */
  }
}
```

The *choice trick*

The *choice trick* is nothing more than a way to exploit function overloading and therefore reduce the boilerplate.

Everything starts from a trivial class definition:

```
template<std::size_t N>
struct choice: choice<N-1> {};

template<>
struct choice<0> {};
```

SFINAE for everyone

```
template<typename T>
auto invoke(choice<3>)
-> decltype(std::declval<T>().h(), void())
{ /* ... */ }

template<typename T>
auto invoke(choice<2>)
-> decltype(std::declval<T>().g(), void())
{ /* ... */ }

template<typename T>
auto invoke(choice<1>)
-> decltype(std::declval<T>().f(), void())
{ /* ... */ }

template<typename T>
void invoke(choice<0>)
{ /* ... */ }

template<typename T>
void invoke() {
  invoke<T>(choice<100>{});
}
```

All at once:

- Unevaluated operands
- Substitution failure is not an error
- Function overloading
- Detection idiom (sort of)
- Choice trick

Questions?

Yet another example

To noexcept or not to noexcept

```
template<typename, typename = std::bool_constant<true>>
struct is_noexcept { /* ... */ };

template<typename T>
struct is_noexcept<T, std::bool_constant<noexcept(std::declval<T>().f())>>
{ /* ... */ };
```

Do you feel like SBO?

```
template<typename, typename = std::bool_constant<true>>
struct can_sbo { /* ... */ };

template<typename T>
struct can_sbo<T, std::bool_constant<sizeof(T) <= sizeof(void *)>>
{ /* ... */ };
```