Unevaluated Operands The SFINAE you don't expect

skypjac

Things to keep in mind

Templates: wh

Operands

SFINAE for Fun and

The way o

decitype and the

Questions?

Still in time?

# Unevaluated Operands The SFINAE you don't expect

skypjack



November 24, 2018

Released under CC BY-SA 4.0

Profits

The way of enable\_if

Questions?

### Table of contents

- Things to keep in mind Templates: what else? Unevaluated Operands
- SFINAE for Fun and Profits The way of enable\_if decltype and the choice trick
- **3** Questions?
- 4 Still in time?

Things to keep in mind Templates: what

Unevaluated Operands

SFINAE for Fun and Profits

The way of enable\_if decitype and the choice trick

 $\mathsf{Questions}?$ 

Still in time

#### 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: Substitution Failure Is Not An Eerror

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) { /* ... */ }
};
```

Still in time

#### 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

SFINAE fo

The way of enable\_if decitype and the

Questions?

## 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() { /* ... */ }
```

SFINAE for Fun and

The way of enable\_if decltype and th

Questions?

C4:11 :... 4:...

# 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() { /* ... */ }
```

SFINAE for Fun and Profits

The way of enable\_if decltype and the

Questions?

# 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_g < T > and !has_g < T > and has_f < T >>
invoke() { /* ... */ }
```

· SFINAE for Fun and

The way of enable\_if decitype and the

Questions?

C+ill in +ima

## 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 if constexpr(has_f<T> and !has_g<T> and !has_h<T>) {
    /* ... */
} else {
    /* ... */
}
```

Things to keep in mind Templates: what else? Unevaluated

SFINAE for

enable\_if

decltype and the

Questions?

## 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> {};
```

Things to keep in mind Templates: what else?

SFINAE for

The way of enable\_if decltype and the

Questions?

```
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

#### Things to

keep in mind

Templates: who

Unevaluat Operands

#### SFINAE for

Profits

The way o

decitype and t

#### Questions?

Still in time?

## Questions?

Italian C++
Community

Things to keep in mind Templates: wha else?

SFINAE for Fun and

The way of enable\_if decltype and the

Questions?

Still in time?

# Yet another example

#### To no except or not to no except

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

template<trypename 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 *)>>
{ /* ... */ };
```