In the example above, the first call to `fun` binds to overload (1) as the first argument binds to `x`, deducing `T` to be `X` and `T::type` to be **`int`**. The second call to `fun` binds to overload (2), as each of its parameters is an exact match with the types of its respective arguments being passed in. The third call to `fun` is not an exact match to overload (2), so overload (1) is considered, but because the deduced type **`int`**`::type` is ill-formed, SFINAE causes (1) to be removed from the overload set, leaving (2) as the only function remaining in the overload set still suitable for invocation.

Attempting to use SFINAE in a similar manner on an exception specification does not work because the exception specification itself is not involved in overload resolution and is not evaluated until the function has already been selected, at which point SFINAE no longer applies and the ill-formed value causes a hard error:

```
template <typename T>
void func(T x) noexcept(T::value);  // (1) function template
void func(double x);                // (2) ordinary function

struct Y { static const bool value = true; };  // compile-time constant value

void test_func()  // Demonstrate how SFINAE fails to work with exception specs.
{
    func(Y());  // OK, calls (1), evaluating T::value as true
    func(0.0);  // OK, calls (2) by means of an exact match
    func(0);    // Error, value is not a member of int
}
```

The first two calls to `func` are similar to those for `fun` (above): The first call, not being an exact match for the ordinary function overload, makes the function template a better match, while the second call is an exact match to the ordinary function, so it is preferred. For the third call to `func`, the templated overload (1) is the best match during overload resolution; because the (conditional) **`noexcept`** specifier is not considered by SFINAE, the function template is not removed from consideration, resulting in an ill-formed program that attempts to reference **`int`**`::value`.