

SFINAE

- ▶ Call expression $f(\text{args})$ is resolved by the compiler as:
 - ▶ The identifier f is being looked for (in this order)
 - In the scope of the function containing the call
 - In the scope of the corresponding class (if inside a member function)
 - In the scope of its base classes (except dependent base class names), recursively
 - If found in more than one base class, the call is invalid, except where resolved by *dominance*
 - Identifiers may be lifted to derived classes with “using base::ident;” declaration
 - In the global/namespace scope
 - Additional declarations may be visible via “using ns::ident;” or “using namespace ns;”
 - If not found as type/variable/constant, *argument-dependent-lookup* is invoked
 - Argument-dependent-lookup considers all namespaces related to types of call arguments
 - ▶ If f is determined to be a function, function *overload resolution* is invoked to select one of the declarations
 - Overload resolution may fail due to ambiguity

► Overload resolution

- Applies to call expressions like $f(\text{args})$ where f is determined to be a function
 - Applied in three independent cases:
 - Global/namespace scope: All function declarations found by ADL
 - Member function declarations in the same class (including those lifted by “using”)
 - Operator invocation like “ $a1+a2$ ”: Global/member cases mixed together
 - All function and function template declarations with the given name are considered
- Phases:
 - For templates, template parameters matching the call are *deducted*
 - Deterministic mechanism defined by language, produces at most one result
 - Deduction may fail
 - Deducted template parameters may cause fail elsewhere in the function header
 - Both kinds of failures lead to exclusion of the declaration from the candidate set (*SFINAE*)
 - All remaining candidate functions (non-templates and successfully instantiated templates) are checked
 - Compatibility wrt. number and types of arguments in the call is verified
 - Return type is NOT considered (i.e. checked wrt. the context of the call)
 - If more than one candidate satisfies the compatibility rules, priority is determined
 - “*More specialized*” templates have priority
 - Declarations resulting in “cheaper implicit conversions” of arguments have priority
 - Both sets of priority rules create only partial orderings – they may fail because of ambiguity

▶ Substitution Failure Is Not An Error

▶ Example:

```
namespace std {  
    template< typename IT>  
        typename iterator_traits<IT>::difference_type distance( IT a, IT b);  
};  
  
template< typename T>  
float distance( const std::string & a, const std::string & b);  
  
std::string x = "Berlin", y = "Paris";  
std::cout << distance(x,y);
```

- *std::distance* is visible for the call *distance(std::string, std::string)*
 - The return type *iterator_traits<std::string>::difference_type* is not defined
- ▶ If function template parameters derived from a function call cause an error when substituted into another *function parameter type* or the *return type*, this function template is excluded from the set of function declarations considered
- ▶ A similar definition applies to template specializations

▶ Substitution Failure Is Not An Error

- ▶ The SFINAE rule is used (and misused) for dirty tricks
- ▶ `std::enable_if<V,T>`
 - `std::enable_if<true,T>::type` `=== T`
 - `std::enable_if<true>::type` `=== void`
 - `std::enable_if<false,T>` does not define the member “type”, which invokes SFINAE
- ▶ `std::enable_if_t<V,T>` `=== typename std::enable_if<V,T>::type`

```
template< typename IT>
```

```
enable_if_t<
```

```
    is_same_v< typename iterator_traits< IT>::iterator_category, random_access_iterator_tag>,  
    typename iterator_traits< IT>::difference_type>
```

```
    distance( IT a, IT b)
```

```
{ return b - a; }
```

```
template< typename IT>
```

```
enable_if_t<
```

```
    ! is_same_v< typename iterator_traits< IT>::iterator_category, random_access_iterator_tag>,  
    typename iterator_traits< IT>::difference_type>
```

```
    distance( IT a, IT b)
```

```
{ for (ptrdiff_t n = 0; a != b; ++n, ++a); return n; }
```

Function template priority

▶ *More specialized* function templates

- ▶ Defines priority of two template functions like

```
template< args1> void f( formals1);    // f1
```

```
template< args2> void f( formals2);    // f2
```

- ▶ Informally: f1 is more specialized than f2 if...
 - ... for each combination of types/constants assigned to args1...
 - Instead of checking infinite number of types/constants, the compiler introduces a fictitious unique type/constant for each template parameter
 - ... the resulting sequence of (types of) formals1 ...
 - Determined by simply substituting the fictitious types/constants into types of formals1
 - Substitution failures are ignored
 - ... may be successfully used as actual arguments to f2
 - the function template argument deduction for f2 is successful
 - Substitution failures are ignored, conversion failures are relevant
- ▶ This relation is not even a partial ordering
 - The winner must be more specialized than all other candidates and all other candidates must not be more specialized than the winner
 - Often there is no winner
- ▶ Partial specializations of class templates are selected using similar rules
 - There are no conversions – simpler and more predictable behavior

- ▶ Enabling a function depending on policy

- Auxiliary template to test convertibility

```
template< typename policy_to, typename policy_from>
```

```
struct is_convertible_policy : std::false_type {};
```

- Partial specialization applicable when the target policy declares “convert_from” policy type

```
template< typename policy_to>
```

```
struct is_convertible_policy< policy_to, typename policy_to::convert_from>
```

```
: std::true_type {};
```

- Convenience interface (templated constant)

```
template< typename policy_to, typename policy_from>
```

```
static constexpr bool is_convertible_policy_v =
```

```
is_convertible_policy< policy_to, policy_from>::value;
```

- Conditionally enabled conversion constructor

- `std::enable_if` placed in an additional anonymous template argument with a default value

```
template< typename policy>
```

```
class generic_iterator {
```

```
template< typename policy2,
```

```
typename = std::enable_if< is_convertible_policy_v< policy, policy2>>>
```

```
generic_iterator(const generic_iterator< policy2> & b) { /*...*/ }
```

```
};
```