

# Skatkov Alex

Implementation of `std::common_type`

# On topic

- Metafunctions using specializations
- Metafunctions using SFINAE
- Implementation of `std::common_type`

# Template specialization

```
template<class T>
class container {
    T content{};
};

int main() {
    container<int> cont1;
    container<char> cont2;
    container<std::vector<int>> cont3;
}
```

# Template specialization

```
template<class T>
class container {
    T content{};
};

int main() {
    container<int> cont1; //OK
    container<char> cont2; //OK
    container<std::vector<int>> cont3; //OK
}
```

# Template specialization

```
struct Special {  
    Special(int) {}  
    Special() = delete;  
};
```

# Template specialization

```
struct Special {  
    Special(int) {}  
    Special() = delete;  
};
```

```
template<class T>  
class container {  
    T content{};  
};
```

```
int main() {  
    container<Special> cont;  
}
```

# Template specialization

```
struct Special {  
    Special(int) {}  
    Special() = delete;  
};
```

```
template<class T>  
class container {  
    T content{}; //error: use of deleted function 'container<Special>::container()'  
};
```

```
int main() {  
    container<Special> cont;  
}
```

# Template specialization

```
struct Special { /* ... */};
```

```
template<class T>
```

```
class container { /* ... */};
```

```
template<>
```

```
struct container<Special> {
```

```
    Special content{0};
```

```
};
```

```
int main() {
```

```
    container<Special> cont;
```

```
}
```



# Template specialization

```
struct Special { /* ... */};
```

```
template<class T>
```

```
class container { /* ... */};
```

```
template<>
```

```
struct container<Special> {
```

```
    Special content{0};
```

```
};
```

```
int main() {
```

```
    container<Special> cont; //OK
```

```
}
```

# std::is\_same

```
template<class U, class V>
```

```
struct is_same {
```

```
    static const bool value = false;
```

```
};
```

```
template<class T>
```

```
struct is_same<T, T> {
```

```
    static const bool value = true;
```

```
};
```

# std::is\_same

```
template<class U, class V>
```

```
struct is_same { /* ... */ };
```

```
template<class T>
```

```
struct is_same<T, T> { /* ... */ };
```

```
int main() {
```

```
    static_assert(is_same<int*****, int*****>::value);
```

```
    static_assert(not is_same<int, char &>::value);
```

```
}
```

# std::is\_same

```
template<class U, class V>  
  
struct is_same { /* ... */ };
```

```
template<class T>
```

```
struct is_same<T, T> { /* ... */ };
```

```
int main() {  
  
    static_assert(is_same<int*****, int*****>::value);  
  
    static_assert(not is_same<int, char &>::value);  
  
}
```

# std::is\_same

```
template<class U, class V>
```

```
struct is_same { /* ... */};
```

```
template<class T>
```

```
struct is_same<T, T> { /* ... */};
```

```
int main() {
```

```
    static_assert(is_same<int*****, int*****>::value);
```

```
    static_assert(not is_same<int, char &>::value);
```

```
}
```

# std::remove\_reference

```
template<class T>
struct remove_reference {
    using type = T;
};
```

```
template<class T>
struct remove_reference<T&> {
    using type = T;
};
```

```
template<class T>
struct remove_reference<T&&> {
    using type = T;
};
```

# std::remove\_reference

```
template<class T>

struct remove_reference {

    using type = T;

};
```

```
template<class T>

struct remove_reference<T&> { // If l-reference

    using type = T;

};
```

```
template<class T>

struct remove_reference<T&&> {

    using type = T;

};
```

# std::remove\_reference

```
template<class T>

struct remove_reference {

    using type = T;

};
```

```
template<class T>

struct remove_reference<T&> { // If l-reference

    using type = T;

};
```

```
template<class T>

struct remove_reference<T&&> { //If r-reference

    using type = T;

};
```



# std::remove\_reference

```
template<class T>
struct remove_reference { //Any other type (not reference)
    using type = T;
};
```

```
template<class T>
struct remove_reference<T&> { // If l-reference
    using type = T;
};
```

```
template<class T>
struct remove_reference<T&&> { //If r-reference
    using type = T;
};
```

# std::remove\_reference

```
// is_same def  
  
// remove_reference def  
  
int main() {  
  
    static_assert (is_same<remove_reference <int&&>::type, int>::value);  
  
    static_assert (is_same<remove_reference <char**&>::type, char**>::value);  
  
    static_assert (is_same<remove_reference <bool*>::type, bool*>::value);  
  
}
```

# std::remove\_reference

```
//...
```

```
template<class T>
```

```
struct remove_reference<T&&> { // If r-reference
```

```
using type = T;
```

```
};
```

```
//...
```

```
int main() {
```

```
    static_assert(is_same<remove_reference<int&&>::type, int>::value);
```

```
    static_assert(is_same<remove_reference<char**&>::type, char**>::value);
```

```
    static_assert(is_same<remove_reference<bool*>::type, bool*>::value);
```

```
}
```

# std::remove\_reference

```
//...  
  
template<class T>  
  
struct remove_reference<T&> { // If l-reference  
  
    using type = T;  
  
};  
  
//...  
  
int main() {  
  
    static_assert (is_same<remove_reference<int&&>::type, int>::value);  
  
    static_assert (is_same<remove_reference<char**&>::type, char**>::value);  
  
    static_assert (is_same<remove_reference<bool*>::type, bool*>::value);  
  
}
```

# std::remove\_reference

```
//...
```

```
template<class T>
```

```
struct remove_reference { //Any other type (not reference)
```

```
    using type = T;
```

```
};
```

```
//...
```

```
int main() {
```

```
    static_assert (is_same<remove_reference <int&&>::type, int>::value);
```

```
    static_assert (is_same<remove_reference <char**&>::type, char**>::value);
```

```
    static_assert (is_same<remove_reference <bool*>::type, bool*>::value);
```

```
}
```

# std::is\_const

```
template<class T>
struct is_const {
    static const bool value = false;
};
```

```
template<class T>
struct is_const<const T> {
    static const bool value = true;
};
```

`std::is_function`

# std::is\_function

```
// primary template
template<class>
struct is_function : std::false_type { };

// specialization for regular functions
template<class Ret, class... Args>
struct is_function<Ret(Args...)> : std::true_type {};

// specialization for variadic functions such as std::printf
template<class Ret, class... Args>
struct is_function<Ret(Args.....)> : std::true_type {};

// specialization for function types that have cv-qualifiers
template<class Ret, class... Args>
struct is_function<Ret(Args...) const> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(Args...) volatile> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(Args...) const volatile> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(Args.....) const> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(Args.....) volatile> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(Args.....) const volatile> : std::true_type {};
```



# std::is\_function

```
// primary template
template<class>
struct is_function : std::false_type {};

// specialization for function types that have ref-qualifiers
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) &&> : std::true_type {};

// specialization for function types that have cv-qualifiers
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) const &> : std::true_type {};

// specialization for function types that have volatile
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) volatile &> : std::true_type {};

// specialization for function types that have cvref
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) &&&> : std::true_type {};

// specialization for function types that have cvref and volatile
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) volatile &&&> : std::true_type {};

// specialization for function types that have cvref and const
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) const &&&> : std::true_type {};

// specialization for function types that have cvref, volatile and const
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) volatile &&&> : std::true_type {};

// specialization for function types that have cvref and volatile
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) volatile &&> : std::true_type {};

// specialization for function types that have cvref and const
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) const &&> : std::true_type {};

// specialization for function types that have cvref, volatile and const
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) volatile &&> : std::true_type {};

// specialization for function types that have cvref and volatile
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) volatile &&> : std::true_type {};

// specialization for function types that have cvref and const
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) const &&> : std::true_type {};

// specialization for function types that have cvref, volatile and const
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) volatile &&> : std::true_type {};
```

# std::is\_function

```
// primary template
template<class>
struct is_function : std::false_type {};

// specializations for function types that have ref-qualifiers
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) &> : std::true_type {};

// specializations for noexcept versions of all the above (C++17 and later)
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) const noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) volatile noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) const volatile noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) const && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) volatile && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) const volatile && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) && && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) const && && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) volatile && && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) const volatile && && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) && && && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) const && && && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) volatile && && && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(AArgs...) const volatile && && && noexcept> : std::true_type {};
```

# std::is\_function

```
template<class T>

struct is_function {

    static const bool value = not std::is_const<const T>::value;

};
```

```
template<class T>

struct is_function<T&&> {

    static const bool value = false;

};
```

```
template<class T>

struct is_function<T&&&> {

    static const bool value = false;

};
```

# std::is\_function

```
template<class T>

struct is_function {

    static const bool value = not std::is_const<const T>::value; //Only functions and references have this trait

};
```

```
template<class T>

struct is_function<T&&> {

    static const bool value = false;

};
```

```
template<class T>

struct is_function<T&&&> {

    static const bool value = false;

};
```

# std::is\_function

```
template<class T>

struct is_function {

    static const bool value = not std::is_const<const T>::value; //Only functions and references have this trait

};

template<class T>

struct is_function<T&&> {

    static const bool value = false;

};

template<class T>

struct is_function<T&&> {

    static const bool value = false;

};

using l ref t = int&;
using r ref t = int&&;
using funct_t = int(char*, bool);
```

# std::is\_function

```
template<class T>

struct is_function {

    static const bool value = not std::is_const<const T>::value; //Only functions and references have this trait

};
```

```
template<class T>

struct is_function<T&&> {

    static const bool value = false;

};
```

```
template<class T>

struct is_function<T&&> {

    static const bool value = false;

};
```

```
using l_ref_t = int&;
using r_ref_t = int&&;
using funct_t = int(char*, bool);
```

```
static_assert(std::is_same_v<l_ref_t, const l_ref_t>);
```

# std::is\_function

```
template<class T>

struct is_function {

    static const bool value = not std::is_const<const T>::value; //Only functions and references have this trait

};

template<class T>

struct is_function<T&&> {

    static const bool value = false;

};

using l_ref_t = int&;
using r_ref_t = int&&;
using funct_t = int(char*, bool);

static assert(std::is_same_v<l_ref_t, const l_ref_t>);
static_assert(std::is_same_v<r_ref_t, const r_ref_t>);

template<class T>

struct is_function<T&&> {

    static const bool value = false;

};
```

# std::is\_function

```
template<class T>

struct is_function {

    static const bool value = not std::is_const<const T>::value; //Only functions and references have this trait

};
```

```
template<class T>
```

```
struct is_function<T&&> {

    static const bool value = false;

};
```

```
using l_ref_t = int&;
using r_ref_t = int&&;
using funct_t = int(char*, bool);
```

```
static_assert(std::is_same_v<l_ref_t, const l_ref_t>);
static_assert(std::is_same_v<r_ref_t, const r_ref_t>);
static_assert(std::is_same_v<funct_t, const funct_t>);
```

```
template<class T>
```

```
struct is_function<T&&> {

    static const bool value = false;

};
```



# std::is\_function

```
template<class T>

struct is_function {

    static const bool value = not std::is_const<const T>::value; //Only functions and references have this trait

};
```

```
template<class T>
```

```
struct is_function<T&&> {
```

```
    static const bool value = false;
```

```
};
```

```
template<class T>
```

```
struct is_function<T&&> {
```

```
    static const bool value = false;
```

```
};
```

```
using l ref t = int&;
using r ref t = int&&;
using funct_t = int(char*, bool);
```

```
static assert(std::is_same_v<l ref t, const l ref t>);
static assert(std::is_same_v<r ref t, const r ref t>);
static_assert(std::is_same_v<funct_t, const funct_t>);
```

# std::is\_function

```
template<class T>

struct is_function {

    static const bool value = not std::is_const<const T>::value; //Only functions and references have this trait

};

template<class T>

struct is_function<T&&> {

    static const bool value = false;

    using l ref t = int&;
    using r ref t = int&&;
    using funct_t = int(char*, bool);

    static assert(std::is_same_v<l ref t, const l ref t>);
    static assert(std::is_same_v<r ref t, const r ref t>);
    static_assert(std::is_same_v<funct_t, const funct_t>);

};

template<class T>

struct is_function<T&&> {

    static const bool value = false;

};

template<class U, class V>
constexpr bool is_same_v = is_same<U, V>::value; //since c++14
```

# std::is\_function

```
template<class T>

struct is_function {

    static const bool value = not std::is_const<const T>::value;

};
```

```
template<class T>

struct is_function<T&> { //Filtering out l-references

    static const bool value = false;

};
```

```
template<class T>

struct is_function<T&&> { //Filtering out r-references

    static const bool value = false;

};
```

# std::is\_array

```
template<class T>
struct is_array {
    static const bool value = false;
};
```

```
template<class T>
struct is_array<T[]> {
    static const bool value = true;
};
```

```
template<class T, auto N>
struct is_array<T[N]> {
    static const bool value = true;
};
```

# std::is\_array

```
template<class T>

struct is_array {

    static const bool value = false;

};
```

```
template<class T>

struct is_array<T[]> { // Specialization for unknown length array type

    static const bool value = true;

};
```

```
template<class T, auto N>

struct is_array<T[N]> {

    static const bool value = true;

};
```

# std::is\_array

```
template<class T>

struct is_array {

    static const bool value = false;

};
```

```
template<class T>

struct is_array<T[]> { // Specialization for unknown length array type

    static const bool value = true;

};
```

```
template<class T, auto N>

struct is_array<T[N]> { // Specialization for length-known array type

    static const bool value = true;

};
```

# std::is\_array

```
template<class T>  
  
struct is_array { // Primary template for non-array types  
  
    static const bool value = false;  
  
};
```

```
template<class T>  
  
struct is_array<T[]> { // Specialization for unknown length array type  
  
    static const bool value = true;  
  
};
```

```
template<class T, auto N>  
  
struct is_array<T[N]> { // Specialization for length-known array type  
  
    static const bool value = true;  
  
};
```

# std::remove\_extent

```
template<class T>
struct remove_extent {
    using type = T;
};
```

```
template<class T, auto N>
struct remove_extent<T[N]> {
    using type = T;
};
```

```
template<class T>
struct remove_extent<T[]> {
    using type = T;
};
```



# std::remove\_extent

```
template<class T>
struct remove_extent {
    using type = T;
};
```

```
template<class T, auto N>
struct remove_extent<T[N]> {
    using type = T;
};
```

```
template<class T>
struct remove_extent<T[]> {
    using type = T;
};
```

# std::remove\_extent

```
template<class T>
struct remove_extent {
    using type = T;
};
```

```
template<class T, auto N>
struct remove_extent<T[N]> {
    using type = T;
};
```

```
template<class T>
struct remove_extent<T[]> {
    using type = T;
};
```

# std::enable\_if

```
template<bool b, class IfTrue>
```

```
struct enable_if;
```

```
template<class IfTrue>
```

```
struct enable_if<true,IfTrue> {
```

```
    using type = IfTrue;
```

```
};
```

```
template<class IfTrue>
```

```
struct enable_if<false,IfTrue> { };
```

# std::enable\_if

```
template<bool b, class IfTrue>
```

```
struct enable_if;
```

```
template<class IfTrue>
```

```
struct enable_if<true, IfTrue> {
```

```
    using type = IfTrue;
```

```
};
```

```
template<class IfTrue>
```

```
struct enable_if<false, IfTrue> { };
```

# std::enable\_if

```
template<bool b, class IfTrue>
```

```
struct enable_if;
```

```
template<class IfTrue>
```

```
struct enable_if<true,IfTrue> {
```

```
    using type = IfTrue;
```

```
};
```

```
template<class IfTrue>
```

```
struct enable_if<false,IfTrue> { };
```

# std::enable\_if

```
template<bool b, class IfTrue>
```

```
struct enable_if;
```

```
template<class IfTrue>
```

```
struct enable_if<true,IfTrue> {
```

```
    using type = IfTrue;
```

```
};
```

```
template<class IfTrue>
```

```
struct enable_if<false,IfTrue> { };
```

```
std::enable_if<true, void>::type
```

```
std::enable_if<true, int>::type
```

```
std::enable_if<true, char>::type
```

```
std::enable_if<false, int>::type
```

# std::enable\_if

```
template<bool b, class IfTrue>
```

```
struct enable_if;
```

```
std::enable_if<true, void>::type //OK
```

```
std::enable_if<true, int>::type //OK
```

```
std::enable_if<true, char>::type //OK
```

```
std::enable_if<false, int>::type
```

```
template<class IfTrue>
```

```
struct enable_if<true, IfTrue> {
```

```
    using type = IfTrue;
```

```
};
```

```
template<class IfTrue>
```

```
struct enable_if<false, IfTrue> { };
```

# std::enable\_if

```
template<bool b, class IfTrue>
```

```
struct enable_if;
```

```
std::enable_if<true, void>::type //OK
```

```
std::enable_if<true, int>::type //OK
```

```
std::enable_if<true, char>::type //OK
```

```
std::enable_if<false, int>::type
```

```
template<class IfTrue>
```

```
struct enable_if<true, IfTrue> {
```

```
    using type = IfTrue;
```

```
};
```

```
template<class IfTrue>
```

```
struct enable_if<false, IfTrue> { };
```



# std::enable\_if

```
template<bool b, class IfTrue>
```

```
struct enable_if;
```

```
std::enable_if<true, void>::type //OK  
std::enable_if<true, int>::type //OK  
std::enable_if<true, char>::type //OK  
std::enable_if<false, int>::type // Error
```

```
template<class IfTrue>
```

```
struct enable_if<true, IfTrue> {
```

```
    using type = IfTrue;
```

```
};
```

```
template<class IfTrue>
```

```
struct enable_if<false, IfTrue> { };
```

# std::enable\_if

```
template<bool b, class IfTrue>
```

```
struct enable_if;
```

```
template<class IfTrue>
```

```
struct enable_if<true, IfTrue> {
```

```
    using type = IfTrue;
```

```
};
```

```
template<class IfTrue>
```

```
struct enable_if<false, IfTrue> { };
```

```
std::enable_if<true, void>::type //OK  
std::enable_if<true, int>::type //OK  
std::enable_if<true, char>::type //OK  
std::enable_if<false, int>::type // Error
```

# std::enable\_if

```
template<bool b, class IfTrue>
```

```
struct enable_if;
```

```
template<class IfTrue>
```

```
struct enable_if<true, IfTrue> {
```

```
    using type = IfTrue;
```

```
};
```

```
template<class IfTrue>
```

```
struct enable_if<false, IfTrue> { };
```

```
std::enable_if<true, void>::type //OK  
std::enable_if<true, int>::type //OK  
std::enable_if<true, char>::type //OK  
std::enable_if<false, int>::type // Error
```

# std::enable\_if

```
template<bool b, class IfTrue>
```

```
struct enable_if;
```

```
template<class IfTrue>
```

```
struct enable_if<true, IfTrue> {
```

```
    using type = IfTrue;
```

```
};
```

```
template<class IfTrue>
```

```
struct enable_if<false, IfTrue> { };
```

```
std::enable_if<true, void>::type //OK
std::enable_if<true, int>::type //OK
std::enable_if<true, char>::type //OK
std::enable_if<false, int>::type // Error
'type' is not a member of 'std::enable_if<false, int>'
```

# std::enable\_if

```
template<bool b, class IfTrue>
```

```
struct enable_if;
```

```
template<class IfTrue>
```

```
struct enable_if<true,IfTrue> {
```

```
    using type = IfTrue;
```

```
};
```

```
template<class IfTrue>
```

```
struct enable_if<false,IfTrue> { };
```

```
std::enable_if<true, void>::type //OK
```

```
std::enable_if<true, int>::type //OK
```

```
std::enable_if<true, char>::type //OK
```

```
std::enable_if<false, int>::type // Error
```

```
'type' is not a member of 'std::enable_if<false, int>'
```

```
template<bool b, class T>
```

```
using enable_if_t = typename enable_if<b,T>::type; //since c++11
```

# std::enable\_if

```
template<bool b, class IfTrue>
```

```
struct enable_if;
```

```
template<class IfTrue>
```

```
struct enable_if<true, IfTrue> {
```

```
    using type = IfTrue;
```

```
};
```

```
template<class IfTrue>
```

```
struct enable_if<false, IfTrue> { };
```

```
std::enable_if<true, void>::type //OK
std::enable_if<true, int>::type //OK
std::enable_if<true, char>::type //OK
std::enable_if<false, int>::type // Error
'type' is not a member of 'std::enable_if<false, int>'
```

```
template<bool b, class T>
```

```
using enable_if_t = typename enable_if<b, T>::type; //since c++11
```



**SFINAE** (Substitution Fail Is Not An Error)

# SFINAE (Substitution Failure Is Not An Error)

```
template<class T>

std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) {

    std::cout<<"I'm foo(int)\n";

}
```

```
template<class T>

std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) {

    std::cout<<"I'm foo(char)\n";

}
```

```
int main() {

    foo(5);

    foo('x');

}
```



# SFINAE (Substitution Failure Is Not An Error)

```
template<class T>

std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) {

    std::cout<<"I'm foo(int)\n";

}
```

```
template<class T>

std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) { // Substitution Failure (not member 'type' in std::enable_if<false,void>)

    std::cout<<"I'm foo(char)\n";

}
```

```
int main() {

    foo(5);

    foo('x');

}
```

# SFINAE (Substitution Failure Is Not An Error)

```
template<class T>

std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) {

    std::cout<<"I'm foo(int)\n";

}

template<class T>

std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) { // Substitution Failure (not member 'type' in std::enable_if<false,void>)

    std::cout<<"I'm foo(char)\n";

}

int main() {

    foo(5);

    foo('x');

}
```

# SFINAE (Substitution Failure Is Not An Error)

```
template<class T>
```

```
std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) {
```

```
    std::cout<<"I'm foo(int)\n";
```

```
}
```

```
template<class T>
```

```
std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) { // Substitution Failure (not member 'type' in std::enable_if<false,void>)
```

```
    std::cout<<"I'm foo(char)\n";
```

```
}
```

```
int main() {
```

```
    foo(5);
```

```
    foo('x');
```

```
}
```

# SFINAE (Substitution Failure Is Not An Error)

```
template<class T>

std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) { //enable_if<true,int>. OK

    std::cout<<"I'm foo(int)\n";

}

template<class T>

std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) { // Substitution Failure (not member 'type' in std::enable_if<false,void>)

    std::cout<<"I'm foo(char)\n";

}

int main() {

    foo(5);

    foo('x');

}
```

# SFINAE (Substitution Failure Is Not An Error)

```
template<class T>

std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) { //enable_if<true,int>. OK

    std::cout<<"I'm foo(int)\n";

}

template<class T>

std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) { // Substitution Failure (not member 'type' in std::enable_if<false,void>)

    std::cout<<"I'm foo(char)\n";

}

int main() {

    foo(5); // I'm foo(int)

    foo('x');

}
```

# SFINAE (Substitution Failure Is Not An Error)

```
template<class T>

std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) {

    std::cout<<"I'm foo(int)\n";

}
```

```
template<class T>

std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) {

    std::cout<<"I'm foo(char)\n";

}
```

```
int main() {

    foo(5); // I'm foo(int)

    foo('x');

}
```

# SFINAE (Substitution Failure Is Not An Error)

```
template<class T>

std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) { // Substitution Failure (not member 'type' in std::enable_if<false,void>)

    std::cout<<"I'm foo(int)\n";

}
```

```
template<class T>

std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) {

    std::cout<<"I'm foo(char)\n";

}
```

```
int main() {

    foo(5); // I'm foo(int)

    foo('x');

}
```

# SFINAE (Substitution Failure Is Not An Error)

```
template<class T>

std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) { // Substitution Failure (not member 'type' in std::enable_if<false,void>)

    std::cout<<"I'm foo(int)\n";

}
```

```
template<class T>

std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) {

    std::cout<<"I'm foo(char)\n";

}
```

```
int main() {

    foo(5); // I'm foo(int)

    foo('x');

}
```



# SFINAE (Substitution Failure Is Not An Error)

```
template<class T>

std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) { // Substitution Failure (not member 'type' in std::enable_if<false,int>)

    std::cout<<"I'm foo(int)\n";

}
```

```
template<class T>

std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) { // enable_if<true,char>. OK.

    std::cout<<"I'm foo(char)\n";

}
```

```
int main() {

    foo(5); // I'm foo(int)

    foo('x');

}
```

# SFINAE (Substitution Failure Is Not An Error)

```
template<class T>

std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) { // Substitution Failure (not member 'type' in std::enable_if<false,int>)

    std::cout<<"I'm foo(int)\n";

}


```

```
template<class T>

std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) { // enable_if<true,char>. OK.

    std::cout<<"I'm foo(char)\n";

}


```

```
int main() {

    foo(5); // I'm foo(int)

    foo('x'); // I'm foo(char)

}


```

# std::void\_t

```
template<class...>  
using void_t = void;
```

# std::void\_t

```
template<class...>
```

```
using void_t = void;
```

```
static_assert(std::is_same_v<std::void_t<char, int, decltype(std::vector<int>{}.begin())>, void>);
```

`std::is_default_constructible`

# std::is\_default\_constructible

```
template< class T, class = void, class... Args>
```

```
struct is_constructible_impl {
```

```
    static const bool value = false;
```

```
};
```

```
template<class T, class... Args>
```

```
struct is_constructible_impkT, std::void_t<decltype(T(Args{}...))>, Args...> {
```

```
    static const bool value = true;
```

```
};
```

# std::is\_default\_constructible

```
template< class T, class = void, class... Args>
```

```
struct is_constructible_impl {
```

```
    static const bool value = false;
```

```
};
```

```
template<class T, class... Args>
```

```
struct is_constructible_impkT, std::void_t<decltype(T(Args{}...))>, Args...> {
```

```
    static const bool value = true;
```

```
};
```

# std::is\_default\_constructible

```
template< class T, class = void, class... Args>
```

```
struct is_constructible_impl {
```

```
    static const bool value = false;
```

```
};
```

```
template<class T, class... Args>
```

```
struct is_constructible_impkT, std::void_t<decltype(T(Args{}...))>, Args...> {
```

```
    static const bool value = true;
```

```
};
```



# std::is\_default\_constructible

```
template< class T, class = void, class... Args>
```

```
struct is_constructible_impl {
```

```
    static const bool value = false;
```

```
};
```

```
template<class T, class... Args>
```

```
struct is_constructible_impkT, std::void_t<decltype(T(Args{}...))>, Args...> {
```

```
    static const bool value = true;
```

```
};
```

```
static_assert(is_constructible_impkstd::unique_ptr<int>, void, int*>::value);
```

# std::is\_default\_constructible

```
template< class T, class = void, class... Args>

struct is_constructible_impl {

    static const bool value = false;

};

template<class T, class... Args>

struct is_constructible_impkT, std::void_t<decltype(T(Args{}...))>, Args...> {

    static const bool value = true;

};

static_assert(is_constructible_impkstd::unique_ptr<int>, void, int*>::value);

[T = std::unique_ptr<int>]

[Args... = int*]
```

# std::is\_default\_constructible

```
template< class T, class = void, class... Args>

struct is_constructible_impl {

    static const bool value = false;

};

template<class T, class... Args>

struct is_constructible_impkT, std::void_t<decltype(T(Args{}...))>, Args...> {

    static const bool value = true;

};

static_assert(is_constructible_impkstd::unique_ptr<int>, void, int*>::value);

[T = std::unique_ptr<int>]

[Args... = int*]                std::unique_ptr<int>(int*{}) //OK
```

# std::is\_default\_constructible

```
template< class T, class = void, class... Args>
```

```
struct is_constructible_impl {
```

```
    static const bool value = false;
```

```
};
```

```
template<class T, class... Args>
```

```
struct is_constructible_impkT, std::void_t<decltype(T(Args{}...))>, Args...> {
```

```
    static const bool value = true;
```

```
};
```

```
static_assert(is_constructible_impkstd::unique_ptr<int>, void, int*>::value);
```

```
[T = std::unique_ptr<int>]
```

```
[Args... = int*]
```

```
std::unique_ptr<int>(int*{})//OK
```

# std::is\_default\_constructible

```
template< class T, class = void, class... Args>
```

```
struct is_constructible_impl {
```

```
    static const bool value = false;
```

```
};
```

```
template<class T, class... Args>
```

```
struct is_constructible_impkT, std::void_t<decltype(T(Args{}...))>, Args...> {
```

```
    static const bool value = true;
```

```
};
```

```
static_assert(is_constructible_impkstd::unique_ptr<int>, void, int*>::value);
```

```
[T = std::unique_ptr<int>]
```

```
[Args... = int*]
```

```
std::unique_ptr<int>{int*{}}
```

# std::is\_default\_constructible

```
template< class T, class = void, class... Args>
```

```
struct is_constructible_impl {
```

```
    static const bool value = false;
```

```
};
```

```
template<class T, class... Args>
```

```
struct is_constructible_impkT, std::void_t<decltype(T(Args{}...))>, Args...> {
```

```
    static const bool value = true;
```

```
};
```

```
static_assert(is_constructible_impkstd::unique_ptr<int>, void, char*>::value);
```

```
[T = std::unique_ptr<char*>]
```

```
[Args... =char*]
```

```
std::unique_ptr<int>(char*{})
```

# std::is\_default\_constructible

```
template< class T, class = void, class... Args>
```

```
struct is_constructible_impl {
```

```
    static const bool value = false;
```

```
};
```

```
template<class T, class... Args>
```

```
struct is_constructible_impkT, std::void_t<decltype(T(Args{}...))>, Args...> { //SF
```

```
    static const bool value = true;
```

```
};
```

```
static_assert(is_constructible_impkstd::unique_ptr<int>, void, char*>::value);
```

```
[T = std::unique_ptr<char*>]
```

```
[Args... =char*]
```

```
std::unique_ptr<int>(char*{}) //NOT OK
```

# std::is\_default\_constructible

```
template< class T, class = void, class... Args>
```

```
struct is_constructible_impl {
```

```
    static const bool value = false; //That's it!
```

```
};
```

```
template<class T, class... Args>
```

```
struct is_constructible_impkT, std::void_t<decltype(T(Args{}...))>, Args...> { //SF
```

```
    static const bool value = true;
```

```
};
```

```
static_assert(is_constructible_impkstd::unique_ptr<int>, void, char*>::value);
```

```
[T = std::unique_ptr<char*>]
```

```
[Args... =char*]
```

```
std::unique_ptr<int>(char*{}) //NOT OK
```



# std::is\_default\_constructible

```
template< class T, class = void, class... Args>
```

```
struct is_constructible_impl {
```

```
    static const bool value = false; //That's it!
```

```
};
```

```
template<class T, class... Args>
```

```
struct is_constructible_impkT, std::void_t<decltype(T(Args{}...))>, Args...> { //SF
```

```
    static const bool value = true;
```

```
};
```

```
static_assert(is_constructible_impkstd::unique_ptr<int>, void, char*>::value); //Assertion fail!
```

```
[T = std::unique_ptr<char*>]
```

```
[Args... =char*]
```

```
std::unique_ptr<int>(char*{}) //NOT OK
```

# std::is\_default\_constructible

```
static_assert(is_constructible_imp<std::unique_ptr<int>, void, int*>::value);
```

# std::is\_default\_constructible

```
static_assert(is_constructible_imp<std::unique_ptr<int>, void, int*>::value);
```

```
???
```

# std::is\_default\_constructible

```
static_assert(is_constructible_impl<std::unique_ptr<int>, void, int*>::value);
```

???

# std::is\_default\_constructible

```
static_assert(is_constructible_imp<std::unique_ptr<int>, void, int*>::value);
```

```
???
```

```
template<class T, class... Args>
```

```
struct is_constructible : is_constructible_imp<T, void, Args...> { };
```

# std::is\_default\_constructible

```
static_assert(is_constructible_imp<std::unique_ptr<int>, void, int*>::value);
```

```
???
```

```
template<class T, class... Args>
```

```
struct is_constructible : is_constructible_imp<T, void, Args...> { };
```

```
static_assert(is_constructible<std::unique_ptr<int>, int*>::value);
```

# std::is\_default\_constructible

```
static_assert(is_constructible_imp<std::unique_ptr<int>, void, int*>::value);
```

???

```
template<class T, class... Args>
```

```
struct is_constructible : is_constructible_imp<T, void, Args...> { };
```

```
static_assert(is_constructible<std::unique_ptr<int>, int*>::value);
```

```
//Looks like all is good...
```

# std::is\_default\_constructible

```
template< class T, class = void, class... Args>
```

```
struct is_constructible_impl {
```

```
    static const bool value = false;
```

```
};
```

```
template<class T, class... Args>
```

```
struct is_constructible_impkT, std::void_t<decltype(T(Args{}...))>, Args...> {
```

```
    static const bool value = true;
```

```
};
```



# std::declval

```
template<class T>
```

```
T&& declval() {};
```

# std::declval

```
template<class T>
```

```
T&& declval() {};
```

```
static_assert(std::is_same_v<decltype(std::declval<int>()), int&&>);
```

# std::declval

```
template<class T>
```

```
T&& declval() {};
```

```
static_assert(std::is_same_v<decltype(std::declval<int>()), int&&>);
```

```
static_assert(std::is_same_v<decltype(std::declval<char&>()), char&>);
```

# std::declval

```
template<class T>
```

```
T&& declval() {};
```

```
static_assert(std::is_same_v<decltype(std::declval<int>()), int&&>);
```

```
static_assert(std::is_same_v<decltype(std::declval<char&>()), char&>);
```

```
static_assert(std::is_same_v<  
    decltype(std::declval<std::vector<int>>().begin()),  
    std::vector<int>::iterator>  
);
```

# std::declval

```
template<class T>
```

```
T&& declval() {};
```

```
static_assert(std::is_same_v<decltype(std::declval<int>()), int&&>);
```

```
static_assert(std::is_same_v<decltype(std::declval<char&>()), char&>);
```

```
static_assert(std::is_same_v<  
    decltype(std::declval<std::vector<int>>().begin()),  
    std::vector<int>::iterator>  
);
```

# std::is\_default\_constructible

```
template< class T, class = void, class... Args>
```

```
struct is_constructible_impl {
```

```
    static const bool value = false;
```

```
};
```

```
template<class T, class... Args>
```

```
struct is_constructible_impkT, std::void_t<decltype(T(Args{}...))>, Args...> {
```

```
    static const bool value = true;
```

```
};
```

# std::is\_default\_constructible

```
template< class T, class = void, class... Args>
```

```
struct is_constructible_impl {
```

```
    static const bool value = false;
```

```
};
```

```
template<class T, class... Args>
```

```
struct is_constructible_impkT, std::void_t<decltype(T(std::declval<Args>()...))>, Args...> {
```

```
    static const bool value = true;
```

```
};
```

`std::common_type`



# std::common\_type

```
auto var = true ? bool{} : char{};
```

What is a type of `var`?

# std::common\_type

```
auto var = true ? bool{} : char{}; // bool?
```

What is a type of `var`?

# std::common\_type

```
auto var = true ? bool{} : char{}; // bool?
```

What is a type of `var`?

```
bool b{};
```

```
std::cin>>b;
```

```
auto var2 = b ? bool{} : char{};
```

What is a type of `var2`?

# std::common\_type

```
auto var = true ? bool{} : char{}; // bool?
```

What is a type of `var`?

```
bool b{};
```

```
std::cin >> b;
```

```
auto var2 = b ? bool{} : char{}; // ???
```

What is a type of `var2`?

# std::common\_type

```
auto var = true ? bool{} : char{}; // bool?
```

What is a type of `var`? // int !

```
bool b{};
```

```
std::cin>>b;
```

```
auto var2 = b ? bool{} : char{}; // ???
```

What is a type of `var2`? // int !

# std::common\_type

How it works?

# std::common\_type

How it works?

- 1) If either E2 or E3 has type `void`, then one of the following must be true, or the program is ill-formed:
- 1.1) Either E2 or E3 (but not both) is a (possibly parenthesized) `throw-expression`. The result of the `throw-expression` has the type and the value category of the other expression. If the other expression is a `throw-expression`, the result is a bit field. Such conditional operator was commonly used in C++11 code prior to C++14.

```
std::string str = 2+2==4 ? "ok" : throw std::logic_error("2+2 != 4");
```

- 1.2) Both E2 and E3 are of type `void` (including the case when they are both `throw-expression`), and the value category of the other expression is `lvalue`.

```
2+2==4 ? throw 123 : throw 456;
```

- 2) Otherwise, if E2 or E3 are glvalue of the same value category as T, respectively, the operands are considered to be of type cv T for the purpose of forming the union of cv1 and cv2.

- 3) Otherwise, if E2 and E3 have different types, at least one of which is a pointer, and the other is a pointer to member, or one is a pointer to member and the other is a null pointer constant, then pointer-to-member conversions and qualification conversions are applied to bring them to common type, and that type is the result.

- 3.1) If Y is an lvalue, the target type is `TY&`, and the reference member of the other operand (call it X) of type `TY` as follows:
  - 3.1.1) If X is an lvalue, the target type is `TY&`, and the reference member of the other operand (call it Y) of type `TY` as follows:
    - 3.1.1.1) If TX and TY are the same class type (ignoring cv-qualification), the target type is `TY&`.
    - 3.1.1.2) otherwise, if TY is a base class of TX, the target type is `TY`.
    - 3.1.1.3) otherwise, the target type is the type of the reference member of the other operand (call it Z) of type `TY`.
  - 3.1.2) If X is a prvalue, or if neither the above conversion sequence is applicable, the target type is `TY`.
- 3.2) If Y is a prvalue, or if neither the above conversion sequence is applicable, the target type is `TY`.
- 3.3) If Y is a pointer, or if neither the above conversion sequence is applicable, the target type is `TY`.
- 3.3.1) If TX and TY are the same class type (ignoring cv-qualification), the target type is `TY`.
- 3.3.2) otherwise, if TY is a base class of TX, the target type is `TY`.
- 3.3.3) otherwise, the target type is the type of the reference member of the other operand (call it Z) of type `TY`.

```
struct A {};  
struct B : A {};  
using T = const B;  
A a = true ? A() : T(); // Y = A()
```

- 3.3.3) otherwise, the target type is the type of the reference member of the other operand (call it Z) of type `TY`.

- 3.4) If both sequences can be formed (E2 to target type of E3 and E3 to target type of E2), and the target type is the same, the program is ill-formed but it is the ambiguous conversion sequence, the program is ill-formed.

- 3.6) If no conversion sequence can be formed (note that it may still be ill-formed e.g. due to access to a non-static member function), the program is ill-formed.
- 4) If E2 and E3 are glvalues of the same type and the same value category, and is a bit-field if at least one of E2 and E3 is a bit-field, the result is a bit-field. Otherwise, the result is a prvalue. If E2 and E3 do not have the same type, and either has (possibly cv-qualified) class type, overload resolution is performed using the built-in candidates below to attempt to convert the operands to built-in types. If the overload resolution fails, the program is ill-formed. Otherwise, the selected conversions are applied and the converted operands are used in place of the original operands for step 6.

- 6) The lvalue-to-rvalue, array-to-pointer, and function-to-pointer conversions are applied to the second and third operands. Then,
  - 6.1) If both E2 and E3 now have the same type, the result is a prvalue of that type designating a temporary object (until C++17) whose result object is (since C++17) copy-initialized from whatever operand was selected after evaluating E1.
  - 6.2) If both E2 and E3 have arithmetic or enumeration type, the result is a prvalue of that type designating a temporary object (until C++17) whose result object is (since C++17) copy-initialized from whatever operand was selected after evaluating E1.
  - 6.3) If both E2 and E3 are pointers, or one is a pointer and the other is a null pointer constant, then pointer-to-member conversions and qualification conversions are applied to bring them to common type, and that type is the result.

```
int*IntPtr;  
static_assert(std::is_same_v<decltype(true?nullptr:IntPtr), int*>); // nullptr becoming int
```

- 6.4) If both E2 and E3 are pointers to members, or one is a pointer to member and the other is a null pointer constant, then pointer-to-member conversions and qualification conversions are applied to bring them to common type, and that type is the result.

```
struct A {  
    int* m_ptr;  
} a;  
int* A::* memPtr = &A::m_ptr; // memPtr is a pointer to member m_ptr of A  
static_assert(std::is_same_v<decltype(false?memPtr:nullptr), int*A::*>);  
// memPtr makes nullptr as type of pointer to member m_ptr of A  
static_assert(std::is_same_v<decltype(false?a.*memPtr:nullptr), int*>);  
// a.*memPtr is now just pointer to int and nullptr also becomes pointer to int
```

- 6.5) If both E2 and E3 are null pointer constants, and at least one of which is of type `std::nullptr_t`, then the result's type is `std::nullptr_t`.
- 6.6) In all other cases, the program is ill-formed.

This section is incomplete

Reason: any change to

make this more readable without losing the fine point? At the