

# 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

```
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::remove\_reference

```
template<class T>
```

```
using remove_reference_t = typename remove_reference<T>::type; //since c++11
```

# 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 (Args...) &&> : 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 {};

// specialization
template<class Ret, class... Args>
struct is_function<Ret (Args...) &&> : 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 {};

// specialization for
template<class Ret, class... Args>
struct is_function<Ret (Args...) &&&> : 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 {};

// specialization
template<class Ret, class... Args>
struct is_function<Ret (Args...) &&&&> : 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 {};

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

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

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

// specializations for function types that have ref-qualifiers and noexcept (C++17 and later)
template<class Ret, class... Args>
struct is_function<Ret(Args...) && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(Args...) const && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(Args...) volatile && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(Args...) const volatile && noexcept> : std::true_type {};

// specializations for function types that have ref-qualifiers and noexcept (C++17 and later)
template<class Ret, class... Args>
struct is_function<Ret(Args...) && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(Args...) const && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(Args...) volatile && noexcept> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(Args...) 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;

};
```

```
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>);
```



# 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;

};
```

```
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;
```

```
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
```



# 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
```

# 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
```

# 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,void>. 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,void>. 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,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&) { // enable_if<true,void>. 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,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&) { // enable_if<true,void>. 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_impl<T, 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_impl<T, 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_impl<T, 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_impl<T, std::void_t<decltype(T(Args{}...))>, Args...> {
```

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

```
};
```

```
static_assert(is_constructible_impl<std::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_impl<T, std::void_t<decltype(T(Args{}...))>, Args...> {
```

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

```
};
```

```
static_assert(is_constructible_impl<std::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_impl<T, std::void_t<decltype(T(Args{}...))>, Args...> {
```

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

```
};
```

```
static_assert(is_constructible_impl<std::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_impl<T, std::void_t<decltype(T(Args{}...))>, Args...> {
```

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

```
};
```

```
static_assert(is_constructible_impl<std::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_impl<T, std::void_t<decltype(T(Args{}...))>, Args...> {
```

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

```
};
```

```
static_assert(is_constructible_impl<std::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_impl<T, std::void_t<decltype(T(Args{}...))>, Args...> {
```

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

```
};
```

```
static_assert(is_constructible_impl<std::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_impl<T, std::void_t<decltype(T(Args{}...))>, Args...> { //SF
```

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

```
};
```

```
static_assert(is_constructible_impl<std::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_impl<T, std::void_t<decltype(T(Args{}...))>, Args...> { //SF
```

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

```
};
```

```
static_assert(is_constructible_impl<std::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_impl<T, std::void_t<decltype(T(Args{}...))>, Args...> { //SF
```

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

```
};
```

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

???

# std::is\_default\_constructible

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

???

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

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

# std::is\_default\_constructible

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

???

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

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

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

# std::is\_default\_constructible

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

???

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

```
struct is_constructible : is_constructible_impl<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_impl<T, 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_impl<T, 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_impl<T, 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 expression 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` expressions).

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

- 2) Otherwise, if E2 or E3 are glvalue bit-fields of the same value category as T, respectively, the operands are considered to be of type cv T for the remainder of this section.

- 3) Otherwise, if E2 and E3 have different types, at least one of which is a `throw-expression`, then an `implicit conversion sequence` is ignored from each of the operands, as described below. An operand (call it X) of type TY as follows:

- 3.1) If X is an lvalue, the target type is TY&, and the reference member of the other operand (call it Y) of type TY&, and the reference member of the other operand (call it Y) of type TY&.
- 3.2) If X is an xvalue, the target type is TY&, and the reference member of the other operand (call it Y) of type TY&.
- 3.3) If X is a prvalue, or if neither the above conversion sequence nor the other operand (call it Y) of type TY& is a (possibly cv-qualified) class type, 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 TX.

```
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 that is the least common ancestor of the two types in the standard conversion hierarchy.
- 3.4) If both sequences can be formed (E2 to target type of E3 and E3 to target type of E2) and at least one of which is of type `std::nullptr_t`, then the result is `std::nullptr_t`.
- 3.5) If both E2 and E3 are null pointer constants, and at least one of which is of type `std::nullptr_t`, then the result is `std::nullptr_t`.
- 3.6) In all other cases, the program is ill-formed.

```
struct A {  
    int* m_ptr;  
};  
A 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*>);  
// 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
```

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



# std::common\_type

How it works?

# std::common\_type

How it works?

```
template<class U, class V>  
using cond_t = decltype(false ? std::declval<U>() : std::declval<V>());
```

# std::common\_type

How it works?

```
template<class U, class V>  
using cond_t = decltype(false ? std::declval<U>() : std::declval<V>());
```

# std::common\_type

How it works?

```
template<class U, class V>
using cond_t = decltype(false ? std::declval<U>() : std::declval<V>());

static_assert(std::is_same_v<
    cond_t<int, unsigned long>,
    unsigned long
>);
```

# std::common\_type

```
template<class T1, class T2>  
  
auto max(const T1& t1, const T2& t2) {  
    return t1 > t2 ? t1 : t2;  
}
```

# std::common\_type

```
template<class T1, class T2>

auto max(const T1& t1, const T2& t2) {

    return t1 > t2 ? t1 : t2;

}
```

```
int main() {

    unsigned var1 = 10;

    int var2 = -5;

    std::cout<<max(var1, var2);

}
```

# std::common\_type

```
template<class T1, class T2>

auto max(const T1& t1, const T2& t2) {

    return t1 > t2 ? t1 : t2;

}


int main() {

    unsigned var1 = 10;

    int var2 = -5;

    std::cout<<max(var1, var2); //4294967291

}
```

# std::common\_type

```
template<class T1, class T2> //[T1 = unsigned, T2 = int]
```

```
auto max(const T1& t1, const T2& t2) {
```

```
    return t1 > t2 ? t1 : t2;
```

```
}
```

```
int main() {
```

```
    unsigned var1 = 10;
```

```
    int var2 = -5;
```

```
    std::cout<<max(var1, var2); //4294967291
```

```
}
```



# std::common\_type

```
template<class T1, class T2> //[T1 = unsigned, T2 = int]

auto max(const T1& t1, const T2& t2) {

    return t1 > t2 ? t1 : t2;

}

static_assert(-5 > 10u);

int main() {

    unsigned var1 = 10;

    int var2 = -5;

    std::cout<<max(var1, var2); //4294967291

}
```

# std::common\_type

```
template<class T1, class T2> //[T1 = unsigned, T2 = int]

auto max(const T1& t1, const T2& t2) {

    return t1 > t2 ? t1 : t2;

}

static_assert(-5 > 10u);

int main() {

    unsigned var1 = 10;

    int var2 = -5;

    std::cout<<max(var1, var2); //4294967291

}
```

# std::common\_type

```
template<class T1, class T2> //[T1 = unsigned, T2 = int]

auto max(const T1& t1, const T2& t2) {

    return t1 > t2 ? t1 : t2; //static_cast<unsigned>(-5);

}

static_assert(-5 > 10u);

int main() {

    unsigned var1 = 10;

    int var2 = -5;

    std::cout<<max(var1, var2); //4294967291

}
```

# std::common\_type

```
template<class T1, class T2> //[T1 = unsigned, T2 = int]

auto max(const T1& t1, const T2& t2) {

    return t1 > t2 ? t1 : t2; //static_cast<unsigned>(-5);

}

static_assert(-5 > 10u);

int main() {

    unsigned var1 = 10;

    int var2 = -5;

    std::cout<<max(var1, var2); //4294967291

}
```

# std::common\_type

```
struct T1;  
  
struct T2;  
  
struct T1 {  
    operator T2();  
};  
  
struct T2 {  
    operator T1();  
};
```

```
int main() {  
    static_assert(std::is_same_v<  
        cond_t<T1, T2>,  
        T1  
    >);  
}
```

# std::common\_type

```
struct T1;  
  
struct T2;  
  
struct T1 {  
    operator T2();  
};  
  
struct T2 {  
    operator T1();  
};
```

```
int main() {  
    static_assert(std::is_same_v<  
        cond_t<T1, T2>,  
        T1  
    >);  
}
```

# std::common\_type

```
struct T1;  
  
struct T2;  
  
struct T1 {  
    operator T2();  
};  
  
struct T2 {  
    operator T1();  
};
```

```
template<class U, class V>  
using cond_t = decltype(false ? std::declval<U>() : std::declval<V>());  
  
int main() {  
    static_assert(std::is_same_v<  
        cond_t<T1, T2>,  
        T1  
    >);  
}
```

# std::common\_type

```
struct T1 {};  
  
struct T2 : T1 {};  
  
struct T3 : T2 {};  
  
int main() {  
    static_assert(std::is_same_v<  
        cond_t<T1, T3>,  
        T1  
    >);  
}
```



# std::common\_type

```
struct T1 {};
```

```
struct T2 : T1 {};
```

```
struct T3 : T2 {};
```

```
int main() {
```

```
    static_assert(std::is_same_v< //Assertion fail!
```

```
        cond_t<T1,T3>,
```

```
        T1
```

```
    >);
```

```
}
```

# std::common\_type

```
struct T1 {};  
  
struct T2 : T1 {};  
  
struct T3 : T2 {};  
  
int main() {  
    static_assert(std::is_same_v< //Assertion fail!  
        cond_t<T1, T3>,  
        T1  
    >);  
}
```

# std::common\_type

```
struct T1 {};  
  
struct T2 : T1 {};  
  
struct T3 : T2 {};  
  
int main() {  
    static_assert(std::is_same_v< cond_t<T1, T3>, T1  
>);  
}
```

```
template<class U, class V>  
using cond_t = decltype(false ? std::declval<U>() : std::declval<V>());
```

# std::common\_type

```
struct T1 {};  
  
struct T2 : T1 {};  
  
struct T3 : T2 {};  
  
int main() {  
    static_assert(std::is_same_v< //Assertion fail!  
        cond_t<T1, T3>,  
        T1  
    >);  
}
```

```
template<class U, class V>  
using cond_t = decltype(false ? std::declval<U>() : std::declval<V>());  
  
template<class T>  
T&& declval() { };
```

# std::common\_type

```
struct T1 {};  
  
struct T2 : T1 {};  
  
struct T3 : T2 {};  
  
int main() {  
    static_assert(std::is_same_v< //Assertion fail!  
        cond_t<T1,T3>,  
        T1&&  
    >);  
}
```

```
template<class U, class V>  
using cond_t = decltype(false ? std::declval<U>() : std::declval<V>());  
  
template<class T>  
T&& declval() { };
```

# std::common\_type

```
struct T1 {};
```

```
struct T2 : T1 {};
```

```
struct T3 : T2 {};
```

```
int main() {
```

```
    static_assert(std::is_same_v< //OK
```

```
        cond_t<T1, T3>,
```

```
        T1&&
```

```
    >);
```

```
}
```

`std::common_type`

# std::common\_type

Our targets:



# `std::common_type`

Our targets:

- Any amount of types (Variadic templates + recursion)

# `std::common_type`

Our targets:

- Any amount of types (Variadic templates + recursion)
- No member 'type' if no conversions available (SFINAE)

# std::common\_type

Our targets:

- Any amount of types (Variadic templates + recursion)
- No member 'type' if no conversions available (SFINAE)
- Some changes with references and function types (std::decay)

# std::common\_type

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

```
using cond_t = decltype(false? std::declval<U>() : std::declval<V>());
```

# std::common\_type

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

```
using cond_t = decltype(false? std::declval<U>() : std::declval<V>());
```

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

```
struct common_type {};
```

# std::common\_type

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

```
using cond_t = decltype(false? std::declval<U>() : std::declval<V>());
```

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

```
struct common_type {};
```

```
template<class T>
```

```
struct common_type<T> : common_type<T,T> {};
```

# std::common\_type

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

```
using cond_t = decltype(false? std::declval<U>() : std::declval<V>());
```

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

```
struct common_type {};
```

```
template<class T>
```

```
struct common_type<T> : common_type<T,T> {};
```

- If `sizeof...(T)` is one (i.e., `T...` contains only one type `T0`), the member type names the same type as `std::common_type<T0, T0>::type` if it exists; otherwise there is no member type.

`std::common_type`

`T1 T2 T3 T4 T5 ...`

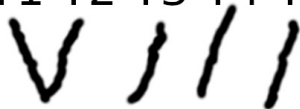


`std::common_type`

T1 T2 T3 T4 T5 ...  
V // //  
T' T3 T4 T5 ...

`std::common_type`

T1 T2 T3 T4 T5 ...



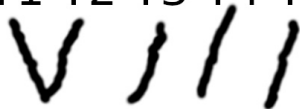
T' T3 T4 T5 ...



T'' T4 T5 ...

`std::common_type`

T1 T2 T3 T4 T5 ...



T' T3 T4 T5 ...



T'' T4 T5 ...

...

`std::common_type`

T1 T2 T3 T4 T5 ...  
? V / / /  
T' T3 T4 T5 ...  
? V / /  
T'' T4 T5 ...  
...

# std::common\_type

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

struct common_type_2_impl { };

template<class U, class V>

struct common_type_2_impl<U, V, std::void_t<cond_t<U, V>>> {

    using type = cond_t<U, V>;

};
```

# std::common\_type

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

struct common_type_2_impl { };

template<class U, class V>

struct common_type_2_impl<U, V, std::void_t<cond_t<U, V>>> {

    using type = std::decay_t<cond_t<U, V>>;

};
```

# std::common\_type

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

```
struct common_type_2_impl { };
```

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

```
struct common_type_2_impl<U, V, std::void_t<cond_t<U, V>>> {
```

```
    using type = std::decay_t<cond_t<U, V>>;
```

```
};
```

- Otherwise, if `std::decay<decltype(false ? std::declval<T1>() : std::declval<T2>())>::type` is a valid type, the member type denotes that type;

# std::common\_type

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

```
struct common_type_2_impl { };
```

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

```
struct common_type_2_impl<U, V, std::void_t<cond_t<U, V>>> {
```

```
    using type = std::decay_t<cond_t<U, V>>;
```

```
};
```



# std::common\_type

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

```
struct common_type<U, V> : common_type_2_impl<U, V> { };
```

- If applying `std::decay` to at least one of T1 and T2 produces a different type, the member type names the same type as `std::common_type<std::decay<T1>::type, std::decay<T2>::type>::type`, if it exists; if not, there is no member type.

# std::common\_type

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

```
struct common_type<U, V> : common_type_2_impl<std::decay_t<U>, std::decay_t<V>> { };
```

- If applying `std::decay` to at least one of T1 and T2 produces a different type, the member type names the same type as `std::common_type<std::decay<T1>::type, std::decay<T2>::type>::type`, if it exists; if not, there is no member type.

# std::common\_type

```
template<class AlwaysVoid, class U, class V, class... Args>
```

```
struct common_type_multi_impl { };
```

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

```
struct common_type_multi_impl<std::void_t<common_type<U, V>>, U, V, Args...>
```

```
    : common_type<typename common_type<U, V>::type, Args...> { };
```

# std::common\_type

```
template<class AlwaysVoid, class U, class V, class... Args>
```

```
struct common_type_multi_impl { };
```

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

```
struct common_type_multi_impl<std::void_t<common_type<U, V>>, U, V, Args...>
```

```
    : common_type<typename common_type<U,V>::type,Args...> { };
```

# std::common\_type

```
template<class AlwaysVoid, class U, class V, class... Args>
```

```
struct common_type_multi_impl { };
```

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

```
struct common_type_multi_impl<std::void_t<common_type<U, V>>, U, V, Args...>
```

```
    : common_type<typename common_type<U, V>::type, Args...> { };
```

# std::common\_type

```
template<class AlwaysVoid, class U, class V, class... Args>
```

```
struct common_type_multi_impl { };
```

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

```
struct common_type_multi_impl<std::void_t<common_type<U, V>>, U, V, Args...>
```

```
    : common_type<typename common_type<U, V>::type, Args...> { };
```

# std::common\_type

```
template<class AlwaysVoid, class U, class V, class... Args>
```

```
struct common_type_multi_impl { };
```

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

```
struct common_type_multi_impl<std::void_t<common_type<U, V>>, U, V, Args...>
```

```
    : common_type<typename common_type<U, V>::type, Args...> { };
```

# std::common\_type

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

```
struct common_type<U, V, Args...> : common_type_multi_impl<void, U, V, Args...> {};
```



For what?

# For what?

```
template<class T1, class T2, class T3>

auto multi_conditional(const T1& t1, const T2& t2, const T3& t3) {

    /*conditions, hard logic...*/

    return t1;

    /*conditions, hard logic...*/

    return t2;

    /*conditions, hard logic...*/

    return t3;

}
```

# For what?

```
template<class T1, class T2, class T3>

auto multi_conditional(const T1& t1, const T2& t2, const T3& t3) {

    /*conditions, hard logic...*/

    return t1;

    /*conditions, hard logic...*/

    return t2;

    /*conditions, hard logic...*/

    return t3;

}

int main() {
    multi_conditional(5, 'c', 5u);
}
```

# For what?

```
template<class T1, class T2, class T3>

auto multi_conditional(const T1& t1, const T2& t2, const T3& t3) {

    /*conditions, hard logic...*/

    return t1;

    /*conditions, hard logic...*/

    return t2; //inconsistent deduction for auto return type: 'int' and then 'char'

    /*conditions, hard logic...*/

    return t3; //inconsistent deduction for auto return type: 'int' and then 'unsigned int'

}

int main() {
    multi_conditional(5, 'c', 5u);
}
```

# For what?

```
template<class T1, class T2, class T3>

auto multi_conditional(const T1& t1, const T2& t2, const T3& t3) -> std::common_type_t<T1,T2,T3> {

    /*conditions, hard logic...*/

    return t1;

    /*conditions, hard logic...*/

    return t2;

    /*conditions, hard logic...*/

    return t3;

}

int main() {
    multi_conditional(5, 'c', 5u);
}
```

# Order matters!

```
struct T1 {  
    operator int();  
};
```

```
struct T2 {  
    operator T1();  
};
```

# Order matters!

```
struct T1 {  
    operator int();  
};
```

```
struct T2 {  
    operator T1();  
};
```

```
int main() {  
    std::common_type_t<T2, T1, int> var;  
    std::common_type_t<T2, int, T1> var;  
}
```

# Order matters!

```
struct T1 {  
    operator int();  
};
```

```
struct T2 {  
    operator T1();  
};
```

```
int main() {  
    // T2 -> T1 -> int  
    std::common_type_t<T2,T1,int> var; // int  
  
    std::common_type_t<T2,int,T1> var;  
}
```



# Order matters!

```
struct T1 {  
    operator int();  
};
```

```
struct T2 {  
    operator T1();  
};
```

```
int main() {  
    // T2 -> T1 -> int  
    std::common_type_t<T2,T1,int> var; // int  
    // T2 ?? int ?? T1  
    std::common_type_t<T2,int,T1> var; // fail  
}
```