

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

std::is_function

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

};

using l_ref t = int&;
using r_ref t = int&&;
using funct_t = int(char*, bool);

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>

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>);

};

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>

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;
};

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>

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 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>

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 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>

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

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;

};
```

```
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

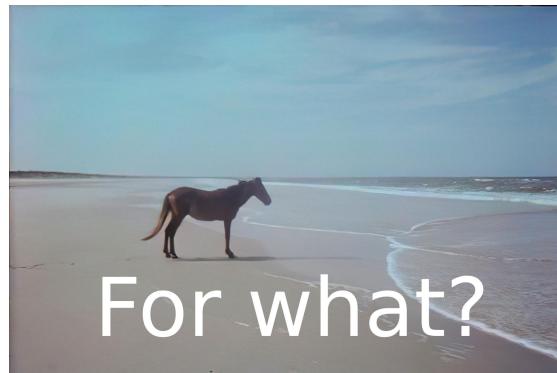
```
template<bool b, class IfTrue>
struct enable_if;

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
```



For what?

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_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`

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

`std::is_default_constructible`

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

???

`std::is_default_constructible`

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

???

std::is_default_constructible

```
static_assert(is_constructible<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<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<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 is a bit field. Such conditional operator was commonly used in C++11 code prior to C++14.

```
std::string str = 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 the prvalue of type `void`).

$2+2==4 ? \text{throw } 123 : \text{throw } 456;$

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 reference cv where cv is the union of cv1 and cv2.

3) Otherwise, if E2 and E3 have different types, at least one of which is a glvalue of the same value category and have the same type except whether a conversion function is detected (since C++14) from each of the other operand, as described below. An operand (call it X) of

3.1) If Y is an lvalue, the target type is TY&, and the reference m₁ is an xvalue, the target type is TY&&, and the reference m₂ is a (possibly cv-qualified) class type,

3.2) If Y is a prvalue, or if neither the above conversion sequence is the target type is TY,

3.3.1) if TX and TY are the same class type (ignoring cv-qualification), the target type is TX, the target type is TY,

3.3.2) otherwise, if TY is a base class of TX, the target

```
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 pointer, and function-to-pointer standard conversion sequence can be formed (E2 to target type of E3). This section is incomplete.

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

- 4) If E2 and E3 are glvalues of the same type and the same value category, then the result has the same type and value category, and is a bit-field if at least one of E2 and E3 is a bit-field.

5) 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-value, 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 **usual arithmetic conversions** are applied to bring them to common type, and that type is the result.

6.3) if both E2 and E3 are pointers, or one is a pointer and the other is a null pointer constant, then pointer 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.