# 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

```cpp
// 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>

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

```cpp
template<class T>

struct is_function {

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

};
```

```cpp
template<class T>

struct is_function<T&> {

   static const bool value = false;

};
```

```cpp
template<class T>

struct is_function<T&&> {

   static const bool value = false;

};
```

```cpp
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

```cpp
template<class T>

struct is_function {

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

};
```

```cpp
template<class T>

struct is_function<T&> {

   static const bool value = false;

};
```

```cpp
template<class T>

struct is_function<T&&> {

   static const bool value = false;

};
```

```cpp
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>);
```

```
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,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_defaut_constructible

# std::is_defaut_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_defaut_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_defaut_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_defaut_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_defaut_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_defaut_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_defaut_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_defaut_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_defaut_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_defaut_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_defaut_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_defaut_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_defaut_constructible

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

# std::is_defaut_constructible

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

???

# std::is_defaut_constructible

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

???

# std::is_defaut_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_defaut_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_defaut_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_defaut_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_defaut_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_defaut_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?