# Skatkov Alex

Implementation of std::common type

# On topic

- Metafunctions using specializations
- Metafunctions using SFINAE
- Implementation of std::common\_type

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

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

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

```
struct Special {
   Special(int){}
   Special() = delete;
};
template<class T>
class container {
   T content{};
};
int main() {
   container<Special> cont;
```

```
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;
```

```
struct Special { /* ... */ };
template<class T>
class container { /* ... */ };
template<>
struct container<Special> {
   Special content{0};
};
int main() {
   container<Special> cont;
```

```
struct Special { /* ... */ };
template<class T>
class container { /* ... */ };
template<>
struct container<Special> {
  Special content{0};
};
int main() {
   container<Special> cont; //OK
```

```
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;
};
```

```
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);
```

```
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);
```

```
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);
```

```
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;
};
```

```
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;
};
```

```
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;
};
```

```
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;
};
```

```
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);
}
```

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

```
//...
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);
```

```
//...
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);
```

```
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;
};
```

```
// 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 cy-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 {};
```

```
template class>
struct is_function : class... | specialization for function types that have ref-qualifiers

// specializatio : class... | specialization for function types that have ref-qualifiers

// specialization function function types that have ref-qualifiers

// specialization function func
      // specialization for function Args? std::true_type {};

// specializatic template<class Ret, class...) &> : std::true_type

template<class Ri struct is function Ret(Args...) const &> : std::true_type

template<class Ri struct is function template<class Ret. struct is_function template</pre>
                                                                                                                                                                                                                                             o template<class Ret, class... Args>
o template<class Ret, class... Args>
struct is_function<Ret(Args...) Args>
template<class Ret, class... Args>
  // specialization template<class Ret, class ...) const &> : std::true_type {};

// specialization template<class Ret, class ...) volatile &> : std::true_type

template<class Ret, struct is function</p>

// specialization template

struct is function

struct is function

struct is function

special:
  template<class Ret, class..." volatile &> : std::true_type {};

template<class Ret, class... const volatile &> : std::true_type {};

template<class Ret, template<class Ret, class... const volatile &> : std::true_type {};

struct is_function

    // specialization for struct is functions Ret, class Re
    template<class Ret, class Ret, cl
  template<class Ret, class Ret(Args. Args volatile & std::true_type {};

template<class Ret, class Ret(Args. Args volatile & std::true_type

template<class Ret, cl. template</class Ret, class Ret, class Ret (Args. Args const volatile & struct is_function</p>
template<class Ret, class Ret, class Ret, class Ret (Args. Args const volatile & struct is_function</p>
template<class Ret, class Ret, class Ret, class Ret (Args. Args const volatile & struct is_function</p>
template<class Ret, class Ret, cl
      struct is_function<Ret(A template<class Ret, class class Ret, class class Ret, class struct is_function<Ret(A template<class Ret, class class class class Ret, class struct is_function<Ret(A template<class Ret, class Ret, class struct is_function<Ret(Art template<class Ret, class Ret, c
    struct is_function<Ret(Arg. template<class Ret, class...) const && : std::true_type {};

struct is_function<Ret(Arg. template<class class class...) const && : std::true_type {};

template<class Ret, class struct is_function<Ret(Arg. template<class class clas
  struct is_function<Ret(Args template<class Ret, class...) const && : std::true_type {};

struct is_function<Ret(Arg template<class Ret, class...) volatile && : std::true_type

template<class Ret, class. struct is function<Ret(Args...) const volatile && : std::true_type

struct is_function<Ret(Args template<class Ret, class...) const volatile && : std::true_type

struct is_function<Ret(Args template<class Ret, class...) const volatile && : std::true_type

struct is_function<Ret(Args template<class Ret, class...) const volatile && : std::true_type

struct is_function<Ret(Args template<class Ret, class...) const volatile && : std::true_type

struct is_function<Ret(Args template<class Ret, class...) const volatile && : std::true_type

struct is_function<Ret(Args template<class Ret, class...) const volatile && : std::true_type

struct is_function<Ret(Args template<class Ret, class Re
                                                                                                                                                                                                                                                                                                                                                                                      template<class Ret, class... Args>
struct is functionclass... Args>
template<class Ret, class... Args>
template<class Ret, class... Args>
template<class Ret, class...</pre>
                                                                                                                                                                                                                                                                                                                                                                                                      template<class Ret, class... Args>
struct is function
Ret, class... Args>
struct is function
Ret, class... Args>
template<class Ret. class...
                                                                                                                                                                                                                                                                                                                                                                                                                     template<class Ret, class... Args> const &&> : std::true_type {};

template<class Ret, class... Args>
struct is function<Ret(Args... Args>
template<class Ret, class...
template<
            struct is_function<Ret(Args.template<class new, Ret(Args...Args...Args...) construct is_function<Ret(Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args...Args..
                                                                                                                                                                                                                                                                                                                                                                                                                                 template<class Ret, class... Args> volatile &&> : std::true_type {};
struct is_function<Ret(Args...Args>
template<class Ret, class...
template<class Ret, class...
                                                                                                                                                                                                                                                                                                                                                                                                                                              template<class Ret, class... Args> const volatile &&> : std::true_type {};
struct is_function<Ret(Args.....)
                                                                                                                                                                                                                                                                                                                                                                                                                   templatecclass Ret, class...
```

```
// specializations for noexcept versions of all the above (C++17 and later)
                   cemplate

cemplate</pr>
struct is_function :

// specialization for function types that have ref-qualifiers

struct is_function :

// specialization for function Args?

// specialization for class...) &> : std::true_type {};

// specialization for function for class...) &> : std::true_type {};

// specialization for function function for function function for function function function function for function function for function functio
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          template<class Ret. class... Args>
struct is functionret(Args...) noexcept> : std::true_type {};
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                template<class Ret, class...
struct is_function<Ret(Args...Args>
template<class Ret, class...Args>
template<class Ret, class...Args>
noexcept>: std::true type {};
                 // specializatic template<class Ret, class...) &> : std::true_type {};

// specializatic template<class Ret, class... } const &> : std::true_type

template<class Ri struct is function Ret(Args... ) const &> : std::true_type
                                                                                                                                                                                                                 late

| Struct is class Ret (Args. Args. A
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                templatecclass Ret, class... Args>
struct is functioneRet(Args...) volatile noexcept> : std::true_type {};
               demplates function templates functions Return and Struct is function templates functions Return and Struct is functions and Struct
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     ctemplatecclass Ret, class:
    struct is functionched (Args: Args>
    templatecclass Ret, class: Args>
    volatile noexcept> : std::true_type {};
                          template<class Restruct as function to template is function Reteast and Reteas
struct is_function

templates function

templates function

templates function

templates function
templates function

templates function

templates function

templates function

templates function

templates function

templates function

templates function

templates function

templates function

templates function

templates function

template
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     templatesclass Ret, class...
struct is function=Ret(args...) & Args>
templatesclass Ret, class... Args>
class... Args>
... Args>
... std::true_type {};
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           templatecclass Ret, class...

struct is function-chet (Args...) const & noexcept> : std::true_type {};
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      templatecclass Ret, class:
struct is functioncRet(Args...) volatile & noexcept> : std::true_type {};
                     templatecclass Ret, class struct is function-class ... Args. templatecclass Ret, class ... Args. templatecclass Ret, class ... Args. volatile & noexcept> : std::true type {};
                     // specialization for struct 13 class Non-Ret(No. 1) // specialization for struct 13 class Non-Ret(No. 1) // struct 15 function for struc
          struct is_function<Ret(struct is_function<Ret, class ret. (lass ret. (lass ret. (lass) args)

template<class Ret, class ret. (lass ret. (lass) args)

struct is_function<Ret(struct is_class) args)

template<class Ret. (lass ret. (lass) args)

struct is_function<Ret(args) args

struct is_function<Ret(a
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               templatecclass Ret, class... Args>
struct is function Ret, class... Args>
templatecclass Ret, class... Args>
const & noexcept> : std::true type {};
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  templatecclass Ret, class...
struct is function-Ret(Args. Args. templatecclass Ret, class... Args.)
volatile & noexcept> : std::true_type {};
                          template<class Ret, clitemplate is function-Ret (struct is function-Ret (struct is function-Ret) is function-Ret (struc
                        struct is function
Ret (Args...) construct is function
                        template<class Ret, clas struct is function</td>
is class non-Ret (Args. Args. Volatile & noexcepts. std::true type template<class Ret, class struct is function</td>
is function
is 
                          If templatecclass Ret, class...
struct is function-Ret(Args...) const volatile & noexcept: std::true_type {};
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   ct is class tion-Ret class. Itemplates functions Ret (Args. Args. Args.) Const volatile & noexcepts (Args.) Struct is functions Ret (Args. Args.) & noexcepts (Args.) Struct is functions Ret (Args.) & noexcepts (Args.) & struct is functions Ret (Args.) & noexcepts (Args.) & struct is functions Ret (Args.) & noexcepts (Args.) & struct is functions Ret (Args.) & noexcepts (Args.) & struct is functions Ret (Args.) & noexcepts (Args.) & struct is functions Ret (Args.) & noexcepts (Args.
                                                                                                                                                                                                                                                                                                                                                                     truct sclass ion-Ret (lass ... template function Ret, class ...)

template class function Ret (largs ... template function Ret ...
                                                                                                                                                                                                                                                                                                                                                                                  ate function Ret return class struct is function, class return class return class return return is function. Class return return
                                                                                                                                                                                                                                                                                                                                                                                     t is function—Ret (Args · struct is function—Ret (Args · Args · const & noexcept · std::true type {};

const template < class Ret (Args · struct is function—Ret (Args · Args · const & noexcept · std::true type {});

struct is function—Ret (Args · struct is function—Ret (Args · Args · const & noexcept · std::true type {});

emplate < class Ret (Args · class · const volatile & noexcept · std::true type {});

struct is Ret (Args · class · const volatile & noexcept · std::true type {});

etruct is Ret (Args · const volatile & noexcept · std::true type {});

etruct is Ret (Args · const volatile & noexcept · std::true type {});
```

```
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;
};
```

```
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;
};
```

```
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;
};
```

```
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>
                                                               static_assert(std::is_same_v<l_ref_t, const l_ref_t>);
struct is_function<T&> {
  static const bool value = false;
};
template<class T>
struct is_function<T&&> {
  static const bool value = false;
};
```

```
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>
                                                              static_assert(std::is_same_v<l_ref_t, const l_ref_t>);
struct is_function<T&> {
                                                              static_assert(std::is_same_v<r_ref_t, const r_ref_t>);
  static const bool value = false;
};
template<class T>
struct is_function<T&&> {
  static const bool value = false;
};
```

```
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>
                                                             static_assert(std::is_same_v<l_ref_t, const l_ref_t>);
struct is_function<T&> {
                                                             static_assert(std::is_same_v<r_ref_t, const r_ref_t>);
  static const bool value = false;
                                                             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 {
  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_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 const bool value = false;
                                                             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 {
  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_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 const bool value = false;
                                                            static_assert(std::is_same_v<funct_t, const funct_t>);
};
                                                template<class U, class V>
template<class T>
                                                constexpr bool is_same_v = is_same<U, V>::value; //since c++14
struct is_function<T&&> {
  static const bool value = false;
};
```

```
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;
};
```

```
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;
};
```

```
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;
};
```

```
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;
};
```

```
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;
};
```

```
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> { };
```

```
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> { };
```

```
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> { };
```

```
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
```

```
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
```

```
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
//OK
```

```
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
// Error
```

```
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> { };
```

```
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
// Error
```

```
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 IfTrue>
struct enable_if;
template<class IfTrue>
struct enable if<true,IfTrue> {
  using type = IfTrue;
};
template<class IfTrue>
struct enable_if<false, IfTrue> { };
```

```
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> { };
```



```
template<class T>
std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) {
   std::cout<<"I'm foo(int)\n";</pre>
template<class T>
std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) {
   std::cout<<"I'm foo(char)\n";</pre>
int main() {
   foo(5);
   foo('x');
```

```
template<class T>
std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) {
   std::cout<<"I'm foo(int)\n";</pre>
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";</pre>
int main() {
   foo(5);
   foo('x');
```

```
template<class T>
std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) {
   std::cout<<"I'm foo(int)\n";</pre>
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";</pre>
int main() {
   foo(5);
   foo('x');
```

```
template<class T>
std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) {
   std::cout<<"I'm foo(int)\n";</pre>
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";</pre>
int main() {
   foo(5);
   foo('x');
```

```
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";</pre>
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";</pre>
int main() {
   foo(5);
   foo('x');
```

```
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";</pre>
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";</pre>
int main() {
   foo(5); // I'm foo(int)
   foo('x');
```

```
template<class T>
std::enable_if_t<std::is_same_v<T,int>, void> foo(const T&) {
   std::cout<<"I'm foo(int)\n";</pre>
template<class T>
std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) {
   std::cout<<"I'm foo(char)\n";</pre>
int main() {
  foo(5); // I'm foo(int)
  foo('x');
```

```
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";</pre>
template<class T>
std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) {
   std::cout<<"I'm foo(char)\n";</pre>
int main() {
   foo(5); // I'm foo(int)
   foo('x');
```

```
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";</pre>
template<class T>
std::enable_if_t<std::is_same_v<T,char>, void> foo(const T&) {
   std::cout<<"I'm foo(char)\n";</pre>
int main() {
   foo(5); // I'm foo(int)
   foo('x');
```

```
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";</pre>
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";</pre>
int main() {
   foo(5); // I'm foo(int)
  foo('x');
```

```
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";</pre>
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";</pre>
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;
};
```

```
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;
};
```

```
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);
```

```
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*]
```

```
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
```

```
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
```

```
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*{})
```

```
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... = \frac{char^*}{}]
                                     std::unique_ptr<int>(char*{})
```

```
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... = \frac{char^*}{}]
                                     std::unique_ptr<int>(char*{})
                                                                       //NOT OK
```

```
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... = \frac{char^*}{}]
                                     std::unique_ptr<int>(char*{})
                                                                       //NOT OK
```

```
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... = \frac{char^*}{}]
                                     std::unique_ptr<int>(char*{})
                                                                       //NOT OK
```

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

```
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...
```

```
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;
};
```

```
template<class T>
T&& declval() {};
```

```
template<class T>

T&& declval() {};

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

```
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&>);
```

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

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

```
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;
};
```

```
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;
};
```

```
auto var = true ? bool{} : char{};
What is a type of `var`?
```

```
auto var = true ? bool{} : char{};  //bool?
What is a type of `var`?
```

```
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`?
```

```
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`?
```

```
auto var = true ? bool{} : char{};
                                                //bool?
What is a type of `var`?
                                                //<mark>int</mark> !
bool b{};
std::cin>>b;
auto var2 = b ? bool{} : char{};
                                                //???
What is a type of `var2`?
                                                //<mark>int</mark> !
```

How it works?

# std::common type operand for the remained of this description (starting at (4))

As a sequence can be formed (note that it may still be ill-formed e.g. due to access the romainad of this description at (41) 1) If either E2 or E3 has type void, then one of the following must be true, or the program is operang for the remained or this description (starting at (4))

3.6) If no conversion sequence can be formed, the operands are left unchanged for the remainder of this 1) If either EZ or E3 has type vold, then one of the following must be true, or the program is

1.1) Either E2 or E3 (but not both) is a (possibly parenthesized) throw-expression. If the other expression. If the other expression and the value category of the other expression. If the other expression and the value category of the other expression. description sequence can be formed, the operands are left unchanged for the remainder of this description (starting at (4))

description sequence can be formed, the operands are left unchanged for the remainder of the original states the same type and the same value category, and is a bit-field if at least one of E2 and E3 is a bit-field.

Sither E2 or E3 has type would, then one of the following must be true, or the prospersion. The rest of E2 are givalues of the same type and the same value category, and is a bit-field if at least one of E2 and E3 is a bit-field.

Sither E2 or E3 has type and the value category of the other examples are left unchanged for the remainder of this description sequence can be formed, the operands are left unchanged for the remainder of this description (starting at (4)).

Sometime E2 or E3 has type would, then one of the following must be true, or the prospersion sequence can be formed, the operands are left unchanged for the remainder of this description (starting at (4)).

Sometime E2 or E3 has type would, then one of the following must be true, or the prospersion sequence can be formed, the operands are left unchanged for the remainder of this description (starting at (4)).

Sometime E2 or E3 has type would, then one of the following must be true, or the prospersion sequence can be formed, the operands are left unchanged for the remainder of this description (starting at (4)).

Sometime E2 or E3 has type would and the converted operands is used in place of the original and the same type and the same value category, and is a bit-field if at least one of E2 and E3 is a bit-field.

Sometime E2 or E3 has type and the same value category, and is a bit-field if at least one of E2 and E3 is a bit-field in the original and the value category.

Sometime E2 or E3 has type and the same value category, and is a bit-field in the province of E2 and E3 is a bit-field in the province of E2 and E3 is a bit-field in the province of E2 and E3 is a bit-field in the province of E2 and E3 is a bit-f operator has the type and the value category of the other expression. If the other exp the result is a bit field. Such conditional operator was commonly used in C++11 content to C++14 content t 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 pryvalue, if E2 and E3 do not have the same type, and either has (possibly cv-qualified) of the built-in candidates below to attempt to convert the 5) Otherwise, the result is a prvalue. If E2 and E3 do not have the same type, and either has (possibly cv-qual one-rands to built-in types if the overload recolution falls the program is ill-formed. Otherwise the selectors. 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 and the converted operands are used in place of the orininal operands for step 6. std::string str = 2+2==4 ? "ok" : throw std::logic\_error("2+2 != / operands to built-in types. If the overload resolution fails, the program is ill-formed. Otherwise, the selected operands are used in place of the original operands for step 6. conversions are applied and the converted operands are used in place of the original operands for step o.

6) The Ivalue-to-tvalue, array-to-pointer, and function-to-pointer conversions are applied to the second and third 1.2) Both E2 and E3 are of type void (including the case when they are both thr operands. Inen,

6.1) If both E2 and E3 now have the same type, the result is a prvalue of that type designating a temporary object function of the control 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 arter evaluating E1.

6.2) If both E2 and E3 have arithmetic or enumeration type: the usual arithmetic conversions are applied to Oring 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 pointer constant. prvalue of type void. 3) If both E2 and E3 are pointers, or one is a pointer and the other is a null pointer constant, then pointer result.

The pointer is a null pointer constant, then pointer constant, then pointer constant, then pointer to common type, and that type is the 2) Otherwise, if EZ or E3 are glyalue bit-fields of the same value category ar T respectively the operands are considered to be of type of T for the re 2+2==4 ? throw 123 : throw 456; 2) Otherwise, if E2 or E3 are givalue bit-fields of the same value category ar T, respectively, the operands are considered to be of type cv T for the re where cv is the union of cv1 and cv2 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 otherwise, if E2 and E3 have different types, at least one of which is a otherwise of the came value category and have the same type exc int\*intPtr;
static\_assert(std::is\_same\_v<decltype(true?nullptr:intPtr), int\*>); // nullptr becoming int 3) Otherwise, if E2 and E3 have different types, at least one of which is a great glvalues of the same value category and have the same type exceeds to form an impolicit conversion sequence in marine member are are glyalues of the same value category and have the same type exc made to form an implicit conversion sequence ignoring member acro whether a conversion function is delated sense from the part of the made to form an implicit conversion sequence ignoring member accomplete a conversion function is deleted time C++141 from each of the conversion function function is deleted time C++141 from each of the conversion function 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 to m Whether a conversion function is deleted (since C++14) from each of by the other operand, as described below, an operand (call it X) of the other operand (rail it Y) of the TV as follows: 4) If both E2 and E3 are pointers to members, or one is a pointer to member and the other is a null pointer common type. and that type is the result. by the other operand, as described below. An operan of the other operand (call it Y) of type TY as follows: 3.1) If Y is an Ivalue, the target type is TY&, and the reference m 3.2) If Y is an avalue, the target type is TYSS, and the reference in Y is an avalue, the target type is TYSS. 3.3) If Y is a prvalue, the larget type is 1100, and the reference 3.3.1) if TX and TY are the same class type (ignoring cv-r } a;
int\* A::\* memptr = 6A::m\_ptr; // memptr is a pointer to member m ptr of A
ctatic accordictd..ic cama vedacltvnaffalca7mamptr:nullntrl. int\*A::\*>); int\* A::\* memptr = &A::m\_ptr; // memptr is a pointer to member m\_ptr of A
static assert(std::is\_same\_vcdecltype(false?memptr:nullptr), int\*A::\*>); 3.3.2) otherwise, if TY is a base class of TX, the target C\_assert(stu::1s\_same\_v<aectiype(latse/memPtr:nutlptr), 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 \*>);

\*\*memptr makes nullptr as type of pointer to member m.ptr of A

\*\*memptr ie now first nointer to int and nullptr, int \*>); itic\_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 struct A {}; nsing t = const g;
T(): // Y = A() struct B : A {}; 6.6) In all other cases, the program is ill-formed. his section is incomplete

A a service type is the type teach of incomplete

3.3.3) otherwise, the target type is the type of each of type of each of the type of each of the type of each of type of each of the type of each of

How it works?

How it works?

```
How it works?

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

```
How it works?

template<class U, class V>

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

```
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<</pre>
       cond_t<int,unsigned long>,
       unsigned long
       >);
```

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

```
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);</pre>
```

```
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</pre>
```

```
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</pre>
```

```
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
```

```
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
```

```
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</pre>
```

```
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
```

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

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

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

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

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

```
template<class U, class V>
struct T1 {};
                            using cond_t = decltype(false ? std::declval<U>() : std::declval<V>());
struct T2 : T1 {};
struct T3 : T2 {};
int main() {
   static_assert(std::is_same_v< //Assertion fail!</pre>
       cond_t<T1, T3>,
       T1
   >);
```

```
template<class U, class V>
struct T1 {};
                            using cond_t = decltype(false ? std::declval<U>() : std::declval<V>());
struct T2 : T1 {};
struct T3 : T2 {};
int main() {
   static_assert(std::is_same_v< //Assertion fail!</pre>
       cond_t<T1, T3>,
       T1
   >);
```

```
template<class U, class V>
struct T1 {};
                            using cond_t = decltype(false ? std::declval<U>() : std::declval<V>());
struct T2 : T1 {};
struct T3 : T2 {};
                                                            template<class T>
                                                            T&& declval() { };
int main() {
   static_assert(std::is_same_v< //Assertion fail!</pre>
       cond_t<T1, T3>,
       T1
  >);
```

```
template<class U, class V>
struct T1 {};
                             using cond_t = decltype(false ? std::declval<U>() : std::declval<V>());
struct T2 : T1 {};
struct T3 : T2 {};
                                                               template<class T>
                                                               T&& declval() { };
int main() {
   static_assert(std::is_same_v< //Assertion fail!</pre>
       cond_t<T1, T3>,
       T1<mark>&&</mark>
   >);
```

```
struct T1 {};
struct T2 : T1 {};
struct T3 : T2 {};
int main() {
   static_assert(std::is_same_v< //0K</pre>
       cond_t<T1, T3>,
       T1&&
  >);
```

Our targets:

#### Our targets:

- Any amount of types (Variadic templates + recursion)

#### Our targets:

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

#### 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)

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

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

```
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.
```

T1 T2 T3 T4 T5 ...

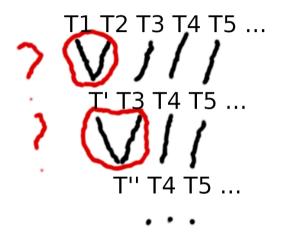




```
T1 T2 T3 T4 T5 ...

T' T3 T4 T5 ...

T'' T4 T5 ...
```



```
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>;
};
```

```
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>>;
};
```

```
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;
```

```
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>>;
};
```

```
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.
```

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.
```

template<class U, class V>

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

```
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;
```

```
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);
```

```
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);
```

```
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);
```

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

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

```
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;
}
```

```
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;
}
```

```
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
}
```