

The Pattern Matching We Already Have

BRADEN GANETSKY



Cppcon
The C++ Conference

20
25



September 13 - 19

C++ pattern matching proposals



C++ pattern matching proposals

- Herb Sutter's P2392



C++ pattern matching proposals

- Herb Sutter's P2392
- Michael Park's P2688 (follow-up to P1371)



C++ pattern matching proposals



C++ pattern matching proposals

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };
```



C++ pattern matching proposals

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };
```



C++ pattern matching proposals

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };
```



C++ pattern matching proposals

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };
```



C++ pattern matching proposals

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };

// With P2392
int get_area(const Shape& shape) {
    return inspect (shape) {
        [r] as Circle => 3.14 * r * r;
        [w, h] as Rectangle => w * h;
    };
}
```



C++ pattern matching proposals

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };

// With P2392
int get_area(const Shape& shape) {
    return inspect (shape) {
        [r] as Circle => 3.14 * r * r;
        [w, h] as Rectangle => w * h;
    };
}
```



C++ pattern matching proposals

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };

// With P2392
int get_area(const Shape& shape) {
    return inspect (shape) {
        [r] as Circle => 3.14 * r * r;
        [w, h] as Rectangle => w * h;
    };
}
```



C++ pattern matching proposals

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };

// With P2392
int get_area(const Shape& shape) {
    return inspect (shape) {
        [r] as Circle => 3.14 * r * r;
        [w, h] as Rectangle => w * h;
    };
}
```



C++ pattern matching proposals

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };

// With P2392
int get_area(const Shape& shape) {
    return inspect (shape) {
        [r] as Circle => 3.14 * r * r;
        [w, h] as Rectangle => w * h;
    };
}
```



C++ pattern matching proposals

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };
```

```
// With P2392
int get_area(const Shape& shape) {
    return inspect (shape) {
        [r] as Circle => 3.14 * r * r;
        [w, h] as Rectangle => w * h;
    };
}
```

```
// With P2688
int get_area(const Shape& shape) {
    return shape match {
        Circle: let [r] => 3.14 * r * r;
        Rectangle: let [w, h] => w * h;
    };
}
```



C++ pattern matching proposals

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };
```

```
// With P2392
int get_area(const Shape& shape) {
    return inspect (shape) {
        [r] as Circle => 3.14 * r * r;
        [w, h] as Rectangle => w * h;
    };
}
```

```
// With P2688
int get_area(const Shape& shape) {
    return shape match {
        Circle: let [r] => 3.14 * r * r;
        Rectangle: let [w, h] => w * h;
    };
}
```



C++ pattern matching proposals

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };
```

```
// With P2392
int get_area(const Shape& shape) {
    return inspect (shape) {
        [r] as Circle => 3.14 * r * r;
        [w, h] as Rectangle => w * h;
    };
}
```

```
// With P2688
int get_area(const Shape& shape) {
    return shape match {
        Circle: let [r] => 3.14 * r * r;
        Rectangle: let [w, h] => w * h;
    };
}
```



C++ pattern matching proposals

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };
```

```
// With P2392
int get_area(const Shape& shape) {
    return inspect (shape) {
        [r] as Circle => 3.14 * r * r;
        [w, h] as Rectangle => w * h;
    };
}
```

```
// With P2688
int get_area(const Shape& shape) {
    return shape match {
        Circle: let [r] => 3.14 * r * r;
        Rectangle: let [w, h] => w * h;
    };
}
```



C++ pattern matching proposals

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };
```

```
// With P2392
int get_area(const Shape& shape) {
    return inspect (shape) {
        [r] as Circle => 3.14 * r * r;
        [w, h] as Rectangle => w * h;
    };
}
```

```
// With P2688
int get_area(const Shape& shape) {
    return shape match {
        Circle: let [r] => 3.14 * r * r;
        Rectangle: let [w, h] => w * h;
    };
}
```



C++ pattern matching proposals

```
struct Shape { virtual int get_area() const = 0; };
struct Circle : Shape { int get_area() const override { return pi * r * r; } };
struct Rectangle : Shape { int get_area() const override { return w * h; } };

// With P2392
int get_area(const Shape& shape) {
    return shape.get_area();
}

// With P2688
int get_area(const Shape& shape) {
    return shape.get_area();
}
Circle: let [r] in pi * r * r;
Rectangle: let [w, h] in w * h;
}
```



C++ pattern matching proposals



C++ pattern matching proposals

There is no pattern matching in the language yet...



C++ pattern matching proposals

There is no pattern matching in the language yet...

...right?



The pattern matching we already have



The pattern matching we already have

```
bool foo(bool x) {  
    return not x;  
}
```



The pattern matching we already have

```
bool foo(bool x) {
    return not x;
}
```

```
template <class T>
requires std::floating_point<T>
void foo(T x) {
    std::cout << x;
}
```



The pattern matching we already have

```
bool foo(bool x) {
    return not x;
}
```

```
template <class T>
requires std::floating_point<T>
void foo(T x) {
    std::cout << x;
}
```

```
int foo(std::unique_ptr<int> x) {
    return x ? *x : 0;
}
```



The pattern matching we already have

```
foo(+[]{});
```

```
bool foo(bool x) {
    return not x;
}
```

```
template <class T>
requires std::floating_point<T>
void foo(T x) {
    std::cout << x;
}
```

```
int foo(std::unique_ptr<int> x) {
    return x ? *x : 0;
}
```



The pattern matching we already have

```
bool foo(bool x) {
    return not x;
}
```

```
template <class T>
requires std::floating_point<T>
void foo(T x) {
    std::cout << x;
}
```

```
int foo(std::unique_ptr<int> x) {
    return x ? *x : 0;
}
```

```
foo(+[]{});
```

```
foo(123);
```



The pattern matching we already have

```
bool foo(bool x) {
    return not x;
}
```

```
foo(+[]{});
```

```
template <class T>
requires std::floating_point<T>
void foo(T x) {
    std::cout << x;
}
```

```
foo(123);
```

```
int foo(std::unique_ptr<int> x) {
    return x ? *x : 0;
}
```

```
foo("abc");
```



The pattern matching we already have

```
bool foo(bool x) {
    return not x;
}
```

```
foo(+[]{});
```

```
template <class T>
requires std::floating_point<T>
void foo(T x) {
    std::cout << x;
}
```

```
// foo(bool)
```

```
int foo(std::unique_ptr<int> x) {
    return x ? *x : 0;
}
```

```
foo(123);
```

```
// foo(bool)
```

```
foo("abc");
```

```
// foo(bool)
```



The pattern matching we already have



The pattern matching we already have

```
template <class T, class U>
struct is_same {
    static constexpr bool value = false;
};
```



The pattern matching we already have

```
template <class T, class U>
struct is_same {
    static constexpr bool value = false;
};
```

```
template <class T>
struct is_same<T, T> {
    static constexpr bool value = true;
};
```



The pattern matching we already have

```
template <class T, class U>
struct is_same {
    static constexpr bool value = false;
};

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



```
template <class T>
struct is_same<T, T> {
    static constexpr bool value = true;
};
```



The pattern matching we already have

```
template <class T, class U>
struct is_same {
    static constexpr bool value = false;
};
```

```
template <class T>
struct is_same<T, T> {
    static constexpr bool value = true;
};
```

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

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



The pattern matching we already have



The pattern matching we already have

- Function overload resolution



The pattern matching we already have

- Function overload resolution
- Template specializations



The pattern matching we already have

- Function overload resolution
- Template specializations
- Template argument deduction



The pattern matching we already have

- Function overload resolution
- Template specializations
- Template argument deduction
- Class template argument deduction



Goal of this talk



Goal of this talk

- Understand basic mechanism of these facilities



Goal of this talk

- Understand basic mechanism of these facilities
- Analogue reasoning about your code



Goal of this talk

- Understand basic mechanism of these facilities
- Analogue reasoning about your code
- Give insight into the logical reasoning in C++



In this talk



In this talk

- Scratching the surface



In this talk

- Scratching the surface
- Minute details are glossed over



In this talk

- Scratching the surface
- Minute details are glossed over
- Newly invented syntax for demonstration purposes



In this talk

- Scratching the surface
- Minute details are glossed over
- Newly invented syntax for demonstration purposes
- Questions at any time!



About me



About me

- Canadian



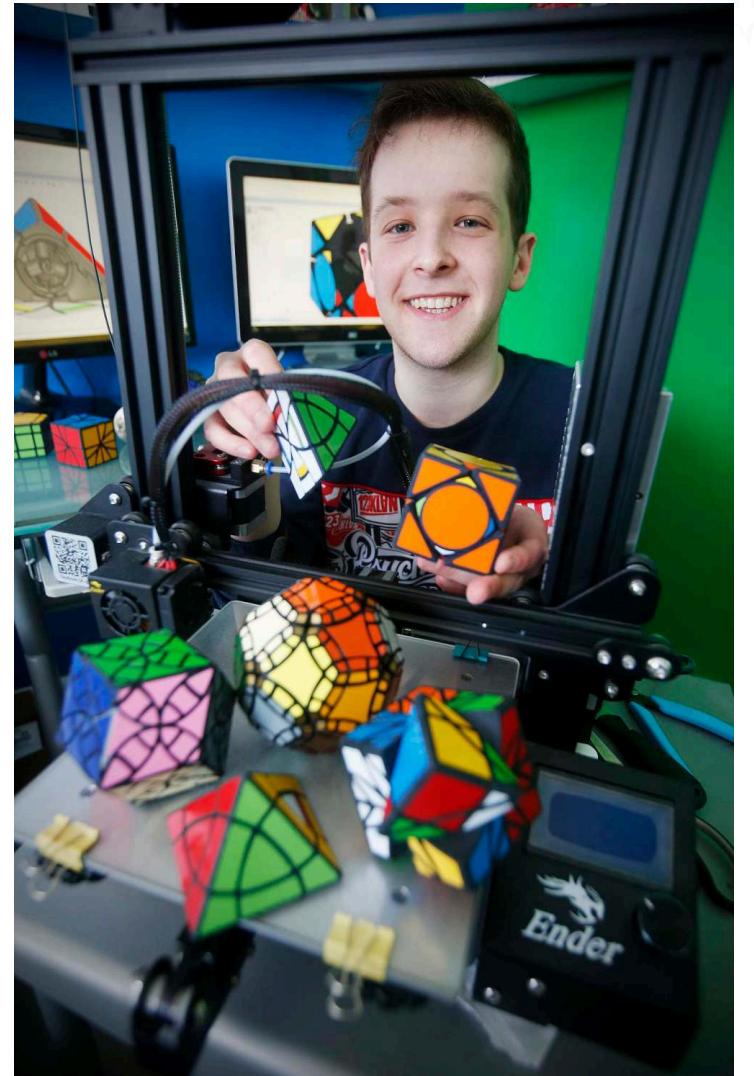
About me

- Canadian
- Licensed professional engineer



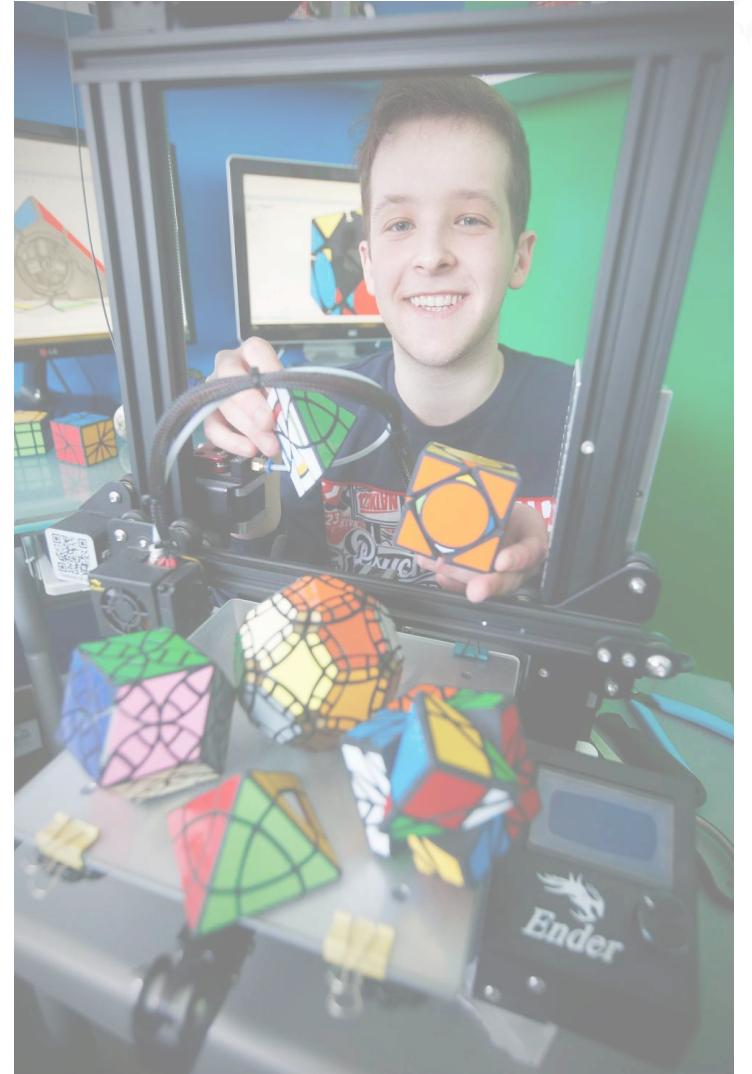
About me

- Canadian
- Licensed professional engineer
- I like twisty puzzles



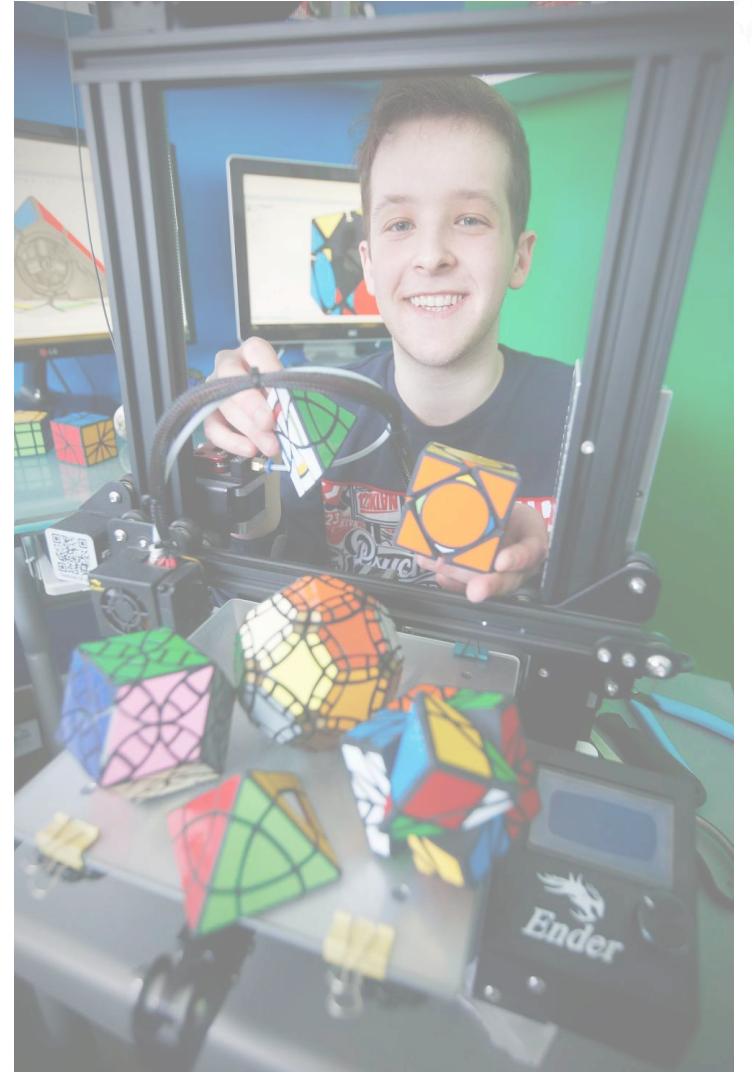
About me

- Canadian
- Licensed professional engineer
- I like twisty puzzles
- Working in C++ for 4 years



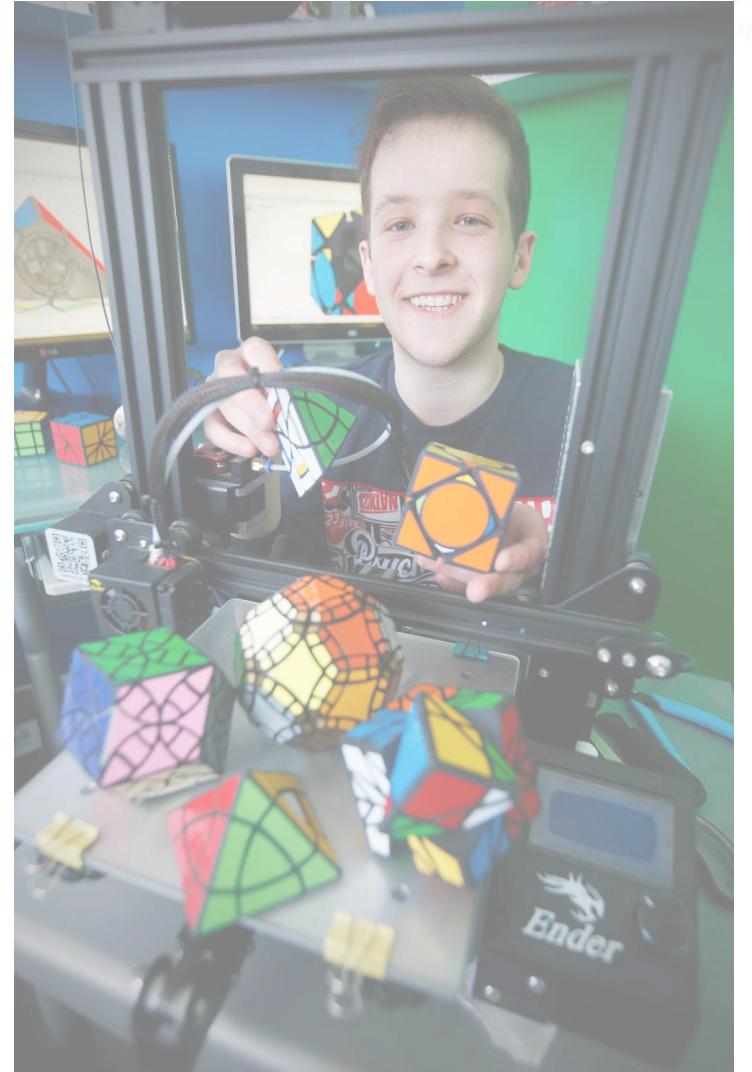
About me

- Canadian
- Licensed professional engineer
- I like twisty puzzles
- Working in C++ for 4 years
- On WG21 since 2023



About me

- Canadian
- Licensed professional engineer
- I like twisty puzzles
- Working in C++ for 4 years
- On WG21 since 2023
- Run Winnipeg C++ Developers



Function overload resolution

Function overload resolution



Function overload resolution

- Since the beginning



Function overload resolution

- Since the beginning
- In the C++98 standard



Function overload resolution

- Since the beginning
- In the C++98 standard
- A defining feature of C++ compared to C



Function overload resolution

- Since the beginning
- In the C++98 standard
- A defining feature of C++ compared to C
- Non-template pattern matching



```
#define init(p, ...) \  
    ((void)0,init(p,__VA_ARGS__+0))  
extern void (init)(Stack*, size_t);
```

C

C++

```
void init(Stack*);  
void init(Stack*, size_t);
```

(code adapted from <https://redd.it/r02cy4>)

C++

Look what they need to
mimic a fraction of our power



Function overloading



Function overloading

- Examples of function overloading



Function overloading

- Examples of function overloading
- How does overload resolution work?



Examples of function overloading

```
1 // From C++ standard [basic.scope.scope]
2 using Int = int;
3 void f(int);
4 void f(Int);
```

redeclaration, overload, or ill-formed?



Examples of function overloading

```
1 // From C++ standard [basic.scope.scope]
2 using Int = int;
3 void f(int);
4 void f(Int);
```

redeclaration



Examples of function overloading

```
1 // From C++ standard [basic.scope.scope]
2 enum E : int {};
3 void f(int);
4 void f(E);
```

redeclaration, overload, or ill-formed?



Examples of function overloading

```
1 // From C++ standard [basic.scope.scope]
2 enum E : int {};
3 void f(int);
4 void f(E);
```

overload



Examples of function overloading

```
1 // From C++98 standard [over.load]
2 void f(int);
3 void f(const int);
```

redeclaration, overload, or ill-formed?



Examples of function overloading

```
1 // From C++98 standard [over.load]
2 void f(int);
3 void f(const int);
```

redeclaration



Examples of function overloading

```
1 void f(int*);  
2 void f(const int*);
```

redeclaration, overload, or ill-formed?



Examples of function overloading

```
1 void f(int*);  
2 void f(const int*);
```

overload



Examples of function overloading

```
1 // From C++98 standard [over.load]
2 int f(char*);  
3 int f(char[]);  
4 int f(char[7]);
```

redeclaration, overload, or ill-formed?



Examples of function overloading

```
1 // From C++98 standard [over.load]
2 int f(char*);  
3 int f(char[]);  
4 int f(char[7]);
```

redeclaration



Examples of function overloading

```
1 // From C++98 standard [over.load]
2 void f(int i, int j);
3 void f(int i, int j = 99); // OK: redeclaration of f(int, int)
4 void f(int i = 88, int j); // OK: redeclaration of f(int, int)
5
6 void f(); // OK: overloaded declaration of f
7
8 void prog ()
9 {
10    f(1, 2); // OK: call f(int, int)
11    f(1); // OK: call f(int, int)
12    f(); // Error: f(int, int) or f()?
13 }
```



Examples of function overloading

```
1 // From C++98 standard [over.load]
2 void f(int i, int j);
3 void f(int i, int j = 99); // OK: redeclaration of f(int, int)
4 void f(int i = 88, int j); // OK: redeclaration of f(int, int)
5
6 void f(); // OK: overloaded declaration of f
7
8 void prog ()
9 {
10    f(1, 2); // OK: call f(int, int)
11    f(1); // OK: call f(int, int)
12    f(); // Error: f(int, int) or f()?
13 }
```



Examples of function overloading

```
1 // From C++98 standard [over.load]
2 void f(int i, int j);
3 void f(int i, int j = 99); // OK: redeclaration of f(int, int)
4 void f(int i = 88, int j); // OK: redeclaration of f(int, int)
5
6 void f(); // OK: overloaded declaration of f
7
8 void prog ()
9 {
10    f(1, 2); // OK: call f(int, int)
11    f(1); // OK: call f(int, int)
12    f(); // Error: f(int, int) or f()?
13 }
```



Examples of function overloading

```
1 // From C++98 standard [over.load]
2 void f(int i, int j);
3 void f(int i, int j = 99); // OK: redeclaration of f(int, int)
4 void f(int i = 88, int j); // OK: redeclaration of f(int, int)
5
6 void f(); // OK: overloaded declaration of f
7
8 void prog ()
9 {
10    f(1, 2); // OK: call f(int, int)
11    f(1); // OK: call f(int, int)
12    f(); // Error: f(int, int) or f()?
13 }
```



Examples of function overloading

```
1 // From C++98 standard [over.load]
2 void f(int i, int j);
3 void f(int i, int j = 99); // OK: redeclaration of f(int, int)
4 void f(int i = 88, int j); // OK: redeclaration of f(int, int)
5
6 void f(); // OK: overloaded declaration of f
7
8 void prog ()
9 {
10    f(1, 2); // OK: call f(int, int)
11    f(1); // OK: call f(int, int)
12    f(); // Error: f(int, int) or f()?
13 }
```



Examples of function overloading

```
1 // From C++98 standard [over.load]
2 void f(int i, int j);
3 void f(int i, int j = 99); // OK: redeclaration of f(int, int)
4 void f(int i = 88, int j); // OK: redeclaration of f(int, int)
5
6 void f(); // OK: overloaded declaration of f
7
8 void prog ()
9 {
10    f(1, 2); // OK: call f(int, int)
11    f(1); // OK: call f(int, int)
12    f(); // Error: f(int, int) or f()?
13 }
```



Examples of function overloading

```
1 // From C++98 standard [over.load]
2 void f(int i, int j);
3 void f(int i, int j = 99); // OK: redeclaration of f(int, int)
4 void f(int i = 88, int j); // OK: redeclaration of f(int, int)
5
6 void f(); // OK: overloaded declaration of f
7
8 void prog ()
9 {
10    f(1, 2); // OK: call f(int, int)
11    f(1); // OK: call f(int, int)
12    f(); // Error: f(int, int) or f()?
13 }
```



Examples of function overloading

```
1 // From C++ standard [basic.scope.scope]
2 struct X {
3     static void f();
4     void f();
5 };
```

redeclaration, overload, or ill-formed?



Examples of function overloading

```
1 // From C++ standard [basic.scope.scope]
2 struct X {
3     static void f();
4     void f();
5 };
```

ill-formed



Examples of function overloading

```
1 // From C++ standard [basic.scope.scope]
2 struct X {
3     void g();
4     void g() const;
5 };
```

redeclaration, overload, or ill-formed?



Examples of function overloading

```
1 // From C++ standard [basic.scope.scope]
2 struct X {
3     void g();
4     void g() const;
5 };
```

overload



Examples of function overloading

```
1 // From C++ standard [basic.scope.scope]
2 struct X {
3     void g();
4     void g() &;
5 };
```

redeclaration, overload, or ill-formed?



Examples of function overloading

```
1 // From C++ standard [basic.scope.scope]
2 struct X {
3     void g();
4     void g() &;
5 };
```

ill-formed



Examples of function overloading

```
1 struct X {  
2     void g() &;  
3     void g() &&;  
4     void g() const &;  
5     void g() const &&;  
6     void g() volatile &;  
7     void g() volatile &&;  
8     void g() const volatile &;  
9     void g() const volatile &&;  
10};
```

redeclaration, overload, or ill-formed?



Examples of function overloading

```
1 struct X {  
2     void g() &;  
3     void g() &&;  
4     void g() const &;  
5     void g() const &&;  
6     void g() volatile &;  
7     void g() volatile &&;  
8     void g() const volatile &;  
9     void g() const volatile &&;  
10 };
```

overload



General mechanism



General mechanism

- Context of overload resolution, possible candidate functions



General mechanism

- Context of overload resolution, possible candidate functions
- List of viable candidate functions



General mechanism

- Context of overload resolution, possible candidate functions
- List of viable candidate functions
- From the viable list, find the "best" one



Distinct contexts of overload resolution



Distinct contexts of overload resolution

- Function calls



Distinct contexts of overload resolution

- Function calls
- Object operator()



Distinct contexts of overload resolution

- Function calls
- Object operator()
- Operator overloading



Distinct contexts of overload resolution

- Function calls
- Object operator()
- Operator overloading
- Constructors



Distinct contexts of overload resolution

- Function calls
- Object `operator()`
- Operator overloading
- Constructors
- Conversion functions



List of viable candidate functions



List of viable candidate functions

- Grab all candidate functions



List of viable candidate functions

- Grab all candidate functions
- Grab the list of arguments



List of viable candidate functions

- Grab all candidate functions
- Grab the list of arguments
- Remove candidates with wrong number of parameters



List of viable candidate functions

- **Grab all candidate functions**
- **Grab the list of arguments**
- **Remove candidates with wrong number of parameters**
- **Remove candidates with unsatisfied constraints**



List of viable candidate functions

- **Grab all candidate functions**
- **Grab the list of arguments**
- **Remove candidates with wrong number of parameters**
- **Remove candidates with unsatisfied constraints**
- **Remove candidates with incompatible arguments**



List of viable candidate functions

```
1 int foo();
2 int foo(double);
3 int foo(int, double = 4.0);
4 int foo(int, bool);
5 int foo(...);
6 int foo(bool);
7 int foo(std::unique_ptr<int>);
8 template <class T>
9     requires (sizeof(T) == 3)
10 int foo(T);
```



List of viable candidate functions

```
1 int foo();
2 int foo(double);
3 int foo(int, double = 4.0);
4 int foo(int, bool);
5 int foo(...);
6 int foo(bool);
7 int foo(std::unique_ptr<int>);
8 template <class T>
9     requires (sizeof(T) == 3)
10 int foo(T);
```



List of viable candidate functions

```
1 int foo();
2 int foo(double);
3 int foo(int, double = 4.0);
4 int foo(int, bool);
5 int foo(...);
6 int foo(bool);
7 int foo(std::unique_ptr<int>);
8 template <class T>
9     requires (sizeof(T) == 3)
10 int foo(T);
```



List of viable candidate functions

```
1 int foo();
2 int foo(double);
3 int foo(int, double = 4.0);
4 int foo(int, bool);
5 int foo(...);
6 int foo(bool);
7 int foo(std::unique_ptr<int>);
8 template <class T>
9     requires (sizeof(T) == 3)
10 int foo(T);
```



List of viable candidate functions

```
1 int foo();
2 int foo(double);
3 int foo(int, double = 4.0);
4 int foo(int, bool);
5 int foo(...);
6 int foo(bool);
7 int foo(std::unique_ptr<int>);
8 template <class T>
9     requires (sizeof(T) == 3)
10 int foo(T);
```



List of viable candidate functions

```
1 int foo();
2 int foo(double);
3 int foo(int, double = 4.0);
4 int foo(int, bool);
5 int foo(...);
6 int foo(bool);
7 int foo(std::unique_ptr<int>);
8 template <class T>
9     requires (sizeof(T) == 3)
10 int foo(T);
```

```
void bar() {
    foo(nullptr);
}
```



List of viable candidate functions

```
1 int foo();
2 int foo(double);
3 int foo(int, double = 4.0);
4 int foo(int, bool);
5 int foo(...);
6 int foo(bool);
7 int foo(std::unique_ptr<int>);
8 template <class T>
9     requires (sizeof(T) == 3)
10 int foo(T);

void bar() {
    foo(nullptr);
}
```

- int foo();
- int foo(double);
- int foo(int, double = 4.0);
- int foo(int, bool);
- int foo(...);
- int foo(bool);
- int foo(std::unique_ptr<int>);
- int foo<T>(T);



List of viable candidate functions

```
1 int foo();
2 int foo(double);
3 int foo(int, double = 4.0);
4 int foo(int, bool);
5 int foo(...);
6 int foo(bool);
7 int foo(std::unique_ptr<int>);
8 template <class T>
9     requires (sizeof(T) == 3)
10 int foo(T);
```

```
void bar() {
    foo(nullptr);
}
```

- int foo();
- int foo(double);
- int foo(int, double = 4.0);
- int foo(int, bool);
- int foo(...);
- int foo(bool);
- int foo(std::unique_ptr<int>);
- int foo<T>(T);



List of viable candidate functions

```
1 int foo();
2 int foo(double);
3 int foo(int, double = 4.0);
4 int foo(int, bool);
5 int foo(...);
6 int foo(bool);
7 int foo(std::unique_ptr<int>);
8 template <class T>
9     requires (sizeof(T) == 3)
10 int foo(T);

void bar() {
    foo(nullptr);
}
```

- int foo();
- int foo(double);
- int foo(int, double = 4.0);
- int foo(int, bool);
- int foo(...);
- int foo(bool);
- int foo(std::unique_ptr<int>);
- int foo<T>(T);



List of viable candidate functions

```
1 int foo();
2 int foo(double);
3 int foo(int, double = 4.0);
4 int foo(int, bool);
5 int foo(...);
6 int foo(bool);
7 int foo(std::unique_ptr<int>);
8 template <class T>
9     requires (sizeof(T) == 3)
10 int foo(T);
```

```
void bar() {
    foo(nullptr);
}
```

- int foo();
- int foo(double);
- int foo(int, double = 4.0);
- int foo(int, bool);
- int foo(...);
- int foo(bool);
- int foo(std::unique_ptr<int>);
- int foo<T>(T);



List of viable candidate functions

```
1 int foo();
2 int foo(double);
3 int foo(int, double = 4.0);
4 int foo(int, bool);
5 int foo(...);
6 int foo(bool);
7 int foo(std::unique_ptr<int>);
8 template <class T>
9     requires (sizeof(T) == 3)
10 int foo(T);
```

```
void bar() {
    foo(nullptr);
}
```

- int foo();
- int foo(double);
- int foo(int, double = 4.0);
- int foo(int, bool);
- int foo(...);
- int foo(bool);
- int foo(std::unique_ptr<int>);
- int foo<T>(T);



Find the best candidate function



Find the best candidate function

- "Implicit conversion sequence" for each parameter-argument pair



Find the best candidate function

- "Implicit conversion sequence" for each parameter-argument pair
- Rank the viable candidates based on conversions



Find the best candidate function

```
1 int foo(...);  
2 int foo(bool);  
3 int foo(std::unique_ptr<int>);
```

```
void bar() {  
    foo(nullptr);  
}
```

`int foo(...);`

`int foo(bool);`

`int foo(std::unique_ptr<int>);`



Find the best candidate function

```
1 int foo(...);  
2 int foo(bool);  
3 int foo(std::unique_ptr<int>);
```

```
void bar() {  
    foo(nullptr);  
}
```

`int foo(...);`

`nullptr_t` to variadic

"ellipsis conversion sequence"

`int foo(bool);`

`int foo(std::unique_ptr<int>);`



Find the best candidate function

```
1 int foo(...);  
2 int foo(bool);  
3 int foo(std::unique_ptr<int>);  
  
void bar() {  
    foo(nullptr);  
}
```

`int foo(...);`
`nullptr_t` to variadic
"ellipsis conversion sequence"

`int foo(bool);`
`nullptr_t` to `bool`
"standard conversion sequence"

`int foo(std::unique_ptr<int>);`



Find the best candidate function

```
1 int foo(...);  
2 int foo(bool);  
3 int foo(std::unique_ptr<int>);  
  
void bar() {  
    foo(nullptr);  
}
```

`int foo(...);`
`nullptr_t` to variadic
"ellipsis conversion sequence"

`int foo(bool);`
`nullptr_t` to `bool`
"standard conversion sequence"

`int foo(std::unique_ptr<int>);`
`nullptr_t` to `std::unique_ptr<int>`
"user-defined conversion sequence"



Find the best candidate function

```
1 int foo(...);  
2 int foo(bool);  
3 int foo(std::unique_ptr<int>);
```

```
void bar() {  
    foo(nullptr);  
}
```



Find the best candidate function

```
1 int foo(...);  
2 int foo(bool);  
3 int foo(std::unique_ptr<int>);
```

```
void bar() {  
    foo(nullptr);  
}
```

1. "standard conversion sequence"



Find the best candidate function

```
1 int foo(...);  
2 int foo(bool);  
3 int foo(std::unique_ptr<int>);
```

```
void bar() {  
    foo(nullptr);  
}
```

1. "standard conversion sequence"
2. "user-defined conversion sequence"



Find the best candidate function

```
1 int foo(...);
2 int foo(bool);
3 int foo(std::unique_ptr<int>);

void bar() {
    foo(nullptr);
}
```

1. "standard conversion sequence"
2. "user-defined conversion sequence"
3. "ellipsis conversion sequence"



Find the best candidate function

```
1 int foo(...);  
2 int foo(bool);  
3 int foo(std::unique_ptr<int>);
```

```
void bar() {  
    foo(nullptr);  
}
```

1. "standard conversion sequence"
2. "user-defined conversion sequence"
3. "ellipsis conversion sequence"

Calls `int foo(bool);`



Template specializations

Template specializations



Template specializations

- Since the beginning



Template specializations

- Since the beginning
- In the C++98 standard



Template specializations

- Since the beginning
- In the C++98 standard
- Facilitate type traits in C++11



Template specializations

- Since the beginning
- In the C++98 standard
- Facilitate type traits in C++11
- "14.5.4.1 Matching of class template partial specializations"



One could imagine...

1
10
01 01
01 01 11
10 11 11 0
00 11 00 00
01 10 10 00 10
11 10 10 00 00
10 11 00 01 01 11 1



One could imagine...

```
// Syntax for demonstration purposes only
(T, U) match {
    (T, T) => true;
    _ => false;
};
```



One could imagine...

```
// Syntax for demonstration purposes only
(T, U) match {
    (T, T) => true;
    _ => false;
};
```

```
// "Primary template"
template <class T, class U>
struct is_same {
    enum {
        value = (int)false
    };
};
```



One could imagine...

```
// Syntax for demonstration purposes only
(T, U) match {
    (T, T) => true;
    _ => false;
};
```

```
// "Primary template"
template <class T, class U>
struct is_same {
    enum {
        value = (int)false
    };
};
```

```
// "Partial specialization"
template <class T>
struct is_same<T, T> {
    enum {
        value = (int)true
    };
};
```



One could imagine...

```
// Syntax for demonstration purposes only
(T, U) match {
    (T, T) => true;
    _ => false;
};
```

```
// "Primary template"
template <class T, class U>
struct is_same {
    enum {
        value = (int)false
    };
};
```

```
// "Partial specialization"
template <class T>
struct is_same<T, T> {
    enum {
        value = (int)true
    };
};
```



One could imagine...

```
// Syntax for demonstration purposes only
(T, U) match {
    (T, T) => true;
    _ => false;
};
```

```
// "Primary template"
template <class T, class U>
struct is_same {
    enum {
        value = (int)false
    };
};
```

```
// "Partial specialization"
template <class T>
struct is_same<T, T> {
    enum {
        value = (int)true
    };
};
```



In C++11

1
10
01 01
01 01 11
10 11 11 0
00 11 00 00
01 10 10 00 10
11 10 10 00 00
10 11 00 01 01 11 1



In C++11

```
// "Primary template"
template <class T, class U>
struct is_same {
    static constexpr bool value = false;
};
```



In C++11

```
// "Primary template"
template <class T, class U>
struct is_same {
    static constexpr bool value = false;
};
```

```
// "Partial specialization"
template <class T>
struct is_same<T, T> {
    static constexpr bool value = true;
};
```



In C++11

```
// "Primary template"
template <class T, class U>
struct is_same {
    static constexpr bool value = false;
};
```

```
// "Partial specialization"
template <class T>
struct is_same<T, T> {
    static constexpr bool value = true;
};
```



In C++11

```
// "Primary template"
template <class T, class U>
struct is_same {
    static constexpr bool value = false;
};
```

```
// "Partial specialization"
template <class T>
struct is_same<T, T> {
    static constexpr bool value = true;
};
```



Backbone of the type traits

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



Backbone of the type traits

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

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



Backbone of the type traits

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

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



Backbone of the type traits

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

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



Broad strokes



Broad strokes

- Primary template



Broad strokes

- Primary template
- Partial specializations



Broad strokes

- Primary template
- Partial specializations
- Explicit specializations



Primary template

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



Primary template

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



Primary template

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



Primary template

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



Partial specialization

```
template <class T, class U>
struct my_trait<std::pair<T, U>> {
    using type = U*;
};
```



Partial specialization

```
template <class T, class U>
struct my_trait<std::pair<T, U>> {
    using type = U*;
};
```



Partial specialization

```
template <class T, class U>
struct my_trait<std::pair<T, U>> {
    using type = U*;
};
```



Partial specialization

```
template <class T, class U>
struct my_trait<std::pair<T, U>> {
    using type = U*;
};
```



Explicit specialization

```
template <>
struct my_trait<int> {
    using type = void;
};
```



Explicit specialization

```
template <>
struct my_trait<int> {
    using type = void;
};
```



Explicit specialization

```
template <>
struct my_trait<int> {
    using type = void;
};
```



Explicit specialization

```
template <>
struct my_trait<int> {
    using type = void;
};
```



Specializations

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



Specializations

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

```
template <>
struct my_other_trait<int*> {
    int foo() {
        return 123;
    }
};
```



Specializations

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

```
template <>
struct my_other_trait<int*> {
    int foo() {
        return 123;
    }
};
```

```
template <class T>
struct my_other_trait<T*> {
    static constexpr bool value = true;
};
```



[temp.expl.spec]/8

The placement of explicit specialization declarations for [...], and the placement of partial specialization declarations of [...] can affect whether a program is well-formed according to the relative positioning of the explicit specialization [...]. When writing a specialization, be careful about its location; or to make it compile will be such a trial as to kindle its self-immolation.



[temp.expl.spec]/8

The placement of explicit specialization declarations for [...], and the placement of partial specialization declarations of [...] can affect whether a program is well-formed according to the relative positioning of the explicit specialization [...]. When writing a specialization, be careful about its location; or to make it compile will be such a trial as to kindle its self-immolation.



Specialization matching



Specialization matching

- Use explicit specialization if available



Specialization matching

- Use explicit specialization if available
- Otherwise, use template argument deduction* to match partial specializations



Specialization matching

- Use explicit specialization if available
- Otherwise, use template argument deduction* to match partial specializations
- If 0 matching partial specializations, use the primary



Specialization matching

- Use explicit specialization if available
- Otherwise, use template argument deduction* to match partial specializations
- If 0 matching partial specializations, use the primary
- If 1 matching partial specialization, use it



Specialization matching

- Use explicit specialization if available
- Otherwise, use template argument deduction* to match partial specializations
- If 0 matching partial specializations, use the primary
- If 1 matching partial specialization, use it
- If multiple matching partial specializations, choose the "best"



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```

A<int, int>

1, 2, or 3?



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```

A<int, int>

1



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```

A<int*, int>

1, 2, or 3?



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```

A<int*, int>

2



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```

A<int***, int>

1, 2, or 3?



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```

A<int***, int>

2



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```

A<int, int*>

1, 2, or 3?



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```

A<int, int*>

3



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```

A<int*, int*>

1, 2, or 3?



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```

A<int*, int*>



Matching against partial specialization



Matching against partial specialization

- Rewrite class templates as function templates



Matching against partial specialization

- Rewrite class templates as function templates
- Use template argument deduction*, and overload resolution



Matching against partial specialization

- Rewrite class templates as function templates
- Use template argument deduction*, and overload resolution
- **Ultimately use the specialization corresponding to the resolved function**



Rewrite class templates as function templates

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```



Rewrite class templates as function templates

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};

// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };

// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
```



Rewrite class templates as function templates

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};

// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };

// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
```



Rewrite class templates as function templates

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};

// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };

// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
```



Rewrite class templates as function templates

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};

// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };

// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
```



Rewrite class templates as function templates

```
// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };
// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
```



Rewrite class templates as function templates

```
// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };
// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
```

```
__imaginary_function(A<int*, int>{});  
// Calls 2 - __imaginary_function<int, int>(A<int*, int>)
```



Rewrite class templates as function templates

```
// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };
// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
```

```
__imaginary_function(A<int*, int>{});  
// Calls 2 - __imaginary_function<int, int>(A<int*, int>)
```

```
__imaginary_function(A<int, int*>{});  
// Calls 3 - __imaginary_function<int, int>(A<int, int*>)
```



Rewrite class templates as function templates

```
// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };
// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
```

```
__imaginary_function(A<int*, int>{});  
// Calls 2 - __imaginary_function<int, int>(A<int*, int>)
```

```
__imaginary_function(A<int, int*>{});  
// Calls 3 - __imaginary_function<int, int>(A<int, int*>)
```

```
__imaginary_function(A<int*, int*>{}); // error: ambiguous
// Could be 2 - __imaginary_function<int, int*>(A<int*, int*>)
// Could be 3 - __imaginary_function<int*, int>(A<int*, int*>)
```



Rewrite class templates as function templates

```
// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };
// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
```

```
__imaginary_function(A<int*, int>{});  
// Calls 2 - __imaginary_function<int, int>(A<int*, int>)
```

```
__imaginary_function(A<int, int*>{});  
// Calls 3 - __imaginary_function<int, int>(A<int, int*>)
```

```
__imaginary_function(A<int*, int*>{}); // error: ambiguous
// Could be 2 - __imaginary_function<int, int*>(A<int*, int*>)
// Could be 3 - __imaginary_function<int*, int>(A<int*, int*>)
```



Rewrite class templates as function templates

```
// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };
// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
```

```
__imaginary_function(A<int*, int>{});  
// Calls 2 - __imaginary_function<int, int>(A<int*, int>)
```

```
__imaginary_function(A<int, int*>{});  
// Calls 3 - __imaginary_function<int, int>(A<int, int*>)
```

```
__imaginary_function(A<int*, int*>{}); // error: ambiguous
// Could be 2 - __imaginary_function<int, int*>(A<int*, int*>)
// Could be 3 - __imaginary_function<int*, int>(A<int*, int*>)
```



Rewrite class templates as function templates

```
// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };
// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
```

```
__imaginary_function(A<int*, int>{});  
// Calls 2 - __imaginary_function<int, int>(A<int*, int>)
```

```
__imaginary_function(A<int, int*>{});  
// Calls 3 - __imaginary_function<int, int>(A<int, int*>)
```

```
__imaginary_function(A<int*, int*>{}); // error: ambiguous
// Could be 2 - __imaginary_function<int, int*>(A<int*, int*>)
// Could be 3 - __imaginary_function<int*, int>(A<int*, int*>)
```



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
```



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};
4 template <class T1, class T2> struct A<T1*, T2*> {};
```



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};

4 template <class T1, class T2> struct A<T1*, T2*> {};

// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };

// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };

// 4
template <class T1, class T2>
void __imaginary_function(A<T1*, T2*>) { };
```



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};

4 template <class T1, class T2> struct A<T1*, T2*> {};

// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };

// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };

// 4
template <class T1, class T2>
void __imaginary_function(A<T1*, T2*>) { };
```



Examples of specialization matching

```
1 template <class T1, class T2> struct A          {};
2 template <class T1, class T2> struct A<T1*, T2> {};
3 template <class T1, class T2> struct A<T1, T2*> {};

4 template <class T1, class T2> struct A<T1*, T2*> {};

// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };

// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };

// 4
template <class T1, class T2>
void __imaginary_function(A<T1*, T2*>) { };
```



Examples of specialization matching

```
// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };
// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
// 4
template <class T1, class T2>
void __imaginary_function(A<T1*, T2*>) { };
```



Examples of specialization matching

```
// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };
// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
// 4
template <class T1, class T2>
void __imaginary_function(A<T1*, T2*>) { };
```

```
__imaginary_function(A<int*, int*>{});  
// Could be 2 - __imaginary_function<int, int*>(A<int*, int*>)  
// Could be 3 - __imaginary_function<int*, int>(A<int*, int*>)  
// Tie breaker - __imaginary_function<int, int>(A<int*, int*>)
```



Examples of specialization matching

```
// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };
// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
// 4
template <class T1, class T2>
void __imaginary_function(A<T1*, T2*>) { };
```

```
__imaginary_function(A<int*, int*>{});  
// Could be 2 - __imaginary_function<int, int*>(A<int*, int*>)  
// Could be 3 - __imaginary_function<int*, int>(A<int*, int*>)  
// Tie breaker - __imaginary_function<int, int>(A<int*, int*>)
```



Examples of specialization matching

```
// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };
// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
// 4
template <class T1, class T2>
void __imaginary_function(A<T1*, T2*>) { };
```

```
__imaginary_function(A<int*, int*>{});  
// Could be 2 - __imaginary_function<int, int*>(A<int*, int*>)  
// Could be 3 - __imaginary_function<int*, int>(A<int*, int*>)  
// Tie breaker - __imaginary_function<int, int>(A<int*, int*>)
```



Examples of specialization matching

```
// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };
// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
// 4
template <class T1, class T2>
void __imaginary_function(A<T1*, T2*>) { };
```

```
__imaginary_function(A<int*, int*>{});  
// Could be 2 - __imaginary_function<int, int*>(A<int*, int*>)  
// Could be 3 - __imaginary_function<int*, int>(A<int*, int*>)  
// Tie breaker - __imaginary_function<int, int>(A<int*, int*>)
```



Examples of specialization matching

```
// 2
template <class T1, class T2>
void __imaginary_function(A<T1*, T2>) { };
// 3
template <class T1, class T2>
void __imaginary_function(A<T1, T2*>) { };
// 4
template <class T1, class T2>
void __imaginary_function(A<T1*, T2*>) { };
```

```
__imaginary_function(A<int*, int*>{});  
// Could be 2 - __imaginary_function<int, int*>(A<int*, int*>)  
// Could be 3 - __imaginary_function<int*, int>(A<int*, int*>)  
// Tie breaker - __imaginary_function<int, int>(A<int*, int*>)
```



Template argument deduction

Template argument deduction



Template argument deduction

```
template <class T>
void foo(std::vector<T>);
```



Template argument deduction

```
template <class T>
void foo(std::vector<T>);
```

```
int main() {
    std::vector<int> vec;
    foo(vec);
    foo<int>(vec);
}
```



Template argument deduction

```
template <class T>
void foo(std::vector<T>);
```

```
int main() {
    std::vector<int> vec;
    foo(vec);
    foo<int>(vec);
}
```



Template argument deduction

```
template <class T>
void foo(std::vector<T>);
```

```
int main() {
    std::vector<int> vec;
    foo(vec);
    foo<int>(vec);
}
```



Template argument deduction

```
template <class T>
void foo(std::vector<T>);
```

```
int main() {
    std::vector<int> vec;
    foo(vec);
    foo<int>(vec);
}
```



Template argument deduction



Template argument deduction

- Since the beginning



Template argument deduction

- Since the beginning
- In the C++98 standard



Template argument deduction

- Since the beginning
- In the C++98 standard
- Deduce template arguments of function templates



Template argument deduction

- Since the beginning
- In the C++98 standard
- Deduce template arguments of function templates
- Can specify some template arguments explicitly



Template argument deduction

- Since the beginning
- In the C++98 standard
- Deduce template arguments of function templates
- Can specify some template arguments explicitly
- New semantics in C++11 with forwarding



Template argument deduction



Template argument deduction

- General mechanism



Template argument deduction

- General mechanism
- Non-deduced context



Template argument deduction

- General mechanism
- Non-deduced context
- Forwarding



General mechanism



General mechanism

- For each function parameter-argument pair:



General mechanism

- For each function parameter-argument pair:
- Ignore top-level cv qualifiers



General mechanism

- For each function parameter-argument pair:
- Ignore top-level cv qualifiers
- Process the parameter's and argument's "special forms"



General mechanism

- **For each function parameter-argument pair:**
- **Ignore top-level cv qualifiers**
- **Process the parameter's and argument's "special forms"**
- **Match the template parameters to the corresponding template arguments**



"Special forms"

1
10
01 01
01 01 11
10 11 11 0
00 11 00 00
01 10 10 00 10
11 00 10 10 11 0
10 11 00 01 01 11 1



"Special forms"

- **cvopt T**



"Special forms"

- $\mathbf{cv_{opt} \ T}$
- $\mathbf{T^*}$



"Special forms"

- **cvopt T**
- **T***
- **T&**



"Special forms"

- **cvopt T**
- **T***
- **T&**
- **T&&**



"Special forms"

- **cv_{opt} T**
- **T***
- **T&**
- **T&&**
- **T [i_{opt}]**



"Special forms"

- **cv_{opt} T**
- **T***
- **T&**
- **T&&**
- **T [i_{opt}]**
- **T (U...) noexcept(i_{opt})**



"Special forms"

- **cv_{opt} T**
- **T***
- **T&**
- **T&&**
- **T [i_{opt}]**
- **T (U...) noexcept(i_{opt})**
- **T U::***



"Special forms"

- **cv_{opt} T**
- **T***
- **T&**
- **T&&**
- **T [i_{opt}]**
- **T (U...) noexcept(i_{opt})**
- **T U::***
- **TT<...>**



"Special forms"

- **cv_{opt} T**
- **T***
- **T&**
- **T&&**
- **T [i_{opt}]**
- **T (U...) noexcept(i_{opt})**
- **T U::***
- **TT<...>**
- **TT<>**



General mechanism examples

```
template <class T>
void foo(T*);
```



General mechanism examples

```
template <class T>
void foo(T*);
```

```
std::vector<int> vec;
foo(&vec);
```



General mechanism examples

```
template <class T>
void foo(T*);
```

```
std::vector<int> vec;
foo(&vec);
```

- Parameter **T***, argument **std::vector<int>***



General mechanism examples

```
template <class T>
void foo(T*);
```

```
std::vector<int> vec;
foo(&vec);
```

- Parameter **T***, argument **std::vector<int>***
- Match the form **T***



General mechanism examples

```
template <class T>
void foo(T*);
```

```
std::vector<int> vec;
foo(&vec);
```

- Parameter `T*`, argument `std::vector<int>*`
- Match the form `T*`
- Therefore `T` is `std::vector<int>`



General mechanism examples

```
template <class T>
void foo(std::vector<T>*);
```



General mechanism examples

```
template <class T>
void foo(std::vector<T>*);
```

```
std::vector<int> vec;
foo(&vec);
```



General mechanism examples

```
template <class T>
void foo(std::vector<T>*);
```

```
std::vector<int> vec;
foo(&vec);
```

- Parameter `std::vector<T>*`, argument `std::vector<int>*`



General mechanism examples

```
template <class T>
void foo(std::vector<T>*);
```

```
std::vector<int> vec;
foo(&vec);
```

- Parameter `std::vector<T>*`, argument `std::vector<int>*`
- Match the form `T*`



General mechanism examples

```
template <class T>
void foo(std::vector<T>*);
```

```
std::vector<int> vec;
foo(&vec);
```

- Parameter `std::vector<T>*`, argument `std::vector<int>*`
- Match the form `T*`
- Therefore `std::vector<T>` is `std::vector<int>`



General mechanism examples

```
template <class T>
void foo(std::vector<T>*);
```

```
std::vector<int> vec;
foo(&vec);
```

- Parameter `std::vector<T>*`, argument `std::vector<int>*`
- Match the form `T*`
- Therefore `std::vector<T>` is `std::vector<int>`
- Match the form `TT<T>`, where `TT` is known to be `std::vector`



General mechanism examples

```
template <class T>
void foo(std::vector<T>*);
```

```
std::vector<int> vec;
foo(&vec);
```

- Parameter `std::vector<T>*`, argument `std::vector<int>*`
- Match the form `T*`
- Therefore `std::vector<T>` is `std::vector<int>`
- Match the form `TT<T>`, where `TT` is known to be `std::vector`
- Therefore `T` is `int`



General mechanism examples

```
template <template <class...> class Vec, class T>
void foo(Vec<T>*);
```



General mechanism examples

```
template <template <class...> class Vec, class T>
void foo(Vec<T>*);
```

```
std::vector<int> vec;
foo(&vec);
```



General mechanism examples

```
template <template <class...> class Vec, class T>
void foo(Vec<T>*);
```

```
std::vector<int> vec;
foo(&vec);
```

- Parameter `Vec<T>*`, argument `std::vector<int>*`



General mechanism examples

```
template <template <class...> class Vec, class T>
void foo(Vec<T>*);
```

```
std::vector<int> vec;
foo(&vec);
```

- Parameter `Vec<T>*`, argument `std::vector<int>*`
- Match the form `T*`



General mechanism examples

```
template <template <class...> class Vec, class T>
void foo(Vec<T>*);
```

```
std::vector<int> vec;
foo(&vec);
```

- Parameter `Vec<T>*`, argument `std::vector<int>*`
- Match the form `T*`
- Therefore `Vec<T>` is `std::vector<int>`



General mechanism examples

```
template <template <class...> class Vec, class T>
void foo(Vec<T>*);
```

```
std::vector<int> vec;
foo(&vec);
```

- Parameter `Vec<T>*`, argument `std::vector<int>*`
- Match the form `T*`
- Therefore `Vec<T>` is `std::vector<int>`
- Match the form `TT<T>`



General mechanism examples

```
template <template <class...> class Vec, class T>
void foo(Vec<T>*);
```

```
std::vector<int> vec;
foo(&vec);
```

- Parameter `Vec<T>*`, argument `std::vector<int>*`
- Match the form `T*`
- Therefore `Vec<T>` is `std::vector<int>`
- Match the form `TT<T>`
- Therefore `Vec` is `std::vector` and `T` is `int`



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);

std::vector<int> vec;
std::pair<int, float> pair;
foo(vec, pair);
```



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```

```
std::vector<int> vec;
std::pair<int, float> pair;
foo(vec, pair);
```

- Parameter `std::vector<T>`, argument `std::vector<int>`



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```

```
std::vector<int> vec;
std::pair<int, float> pair;
foo(vec, pair);
```

- Parameter `std::vector<T>`, argument `std::vector<int>`
- Match the form `TT<T>`, where `TT` is known to be `std::vector`



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```

```
std::vector<int> vec;
std::pair<int, float> pair;
foo(vec, pair);
```

- Parameter `std::vector<T>`, argument `std::vector<int>`
- Match the form `TT<T>`, where `TT` is known to be `std::vector`
- Therefore `T` is `int`



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```

```
std::vector<int> vec;
std::pair<int, float> pair;
foo(vec, pair);
```

- Parameter `std::vector<T>`, argument `std::vector<int>`
- Match the form `TT<T>`, where `TT` is known to be `std::vector`
- Therefore `T` is `int`
- Parameter `std::pair<T,U>`, argument `std::pair<int,float>`



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```

```
std::vector<int> vec;
std::pair<int, float> pair;
foo(vec, pair);
```

- Parameter `std::vector<T>`, argument `std::vector<int>`
- Match the form `TT<T>`, where `TT` is known to be `std::vector`
- Therefore `T` is `int`

- Parameter `std::pair<T,U>`, argument `std::pair<int,float>`
- Match the form `TT<T1,T2>`, where `TT` is known to be `std::pair`



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```

```
std::vector<int> vec;
std::pair<int, float> pair;
foo(vec, pair);
```

- Parameter `std::vector<T>`, argument `std::vector<int>`
- Match the form `TT<T>`, where `TT` is known to be `std::vector`
- Therefore `T` is `int`

- Parameter `std::pair<T,U>`, argument `std::pair<int,float>`
- Match the form `TT<T1,T2>`, where `TT` is known to be `std::pair`
- Therefore `T` is `int` and `U` is `float`



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);

std::vector<int> vec;
std::pair<double, float> pair;
foo(vec, pair);
```



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);

std::vector<int> vec;
std::pair<double, float> pair;
foo(vec, pair);
```



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```

```
std::vector<int> vec;
std::pair<double, float> pair;
foo(vec, pair);
```



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);

std::vector<int> vec;
std::pair<double, float> pair;
foo(vec, pair);
```



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```

```
std::vector<int> vec;
std::pair<double, float> pair;
foo(vec, pair);
```

- Parameter `std::vector<T>`, argument `std::vector<int>`



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```

```
std::vector<int> vec;
std::pair<double, float> pair;
foo(vec, pair);
```

- Parameter `std::vector<T>`, argument `std::vector<int>`
- Match the form `TT<T>`, where `TT` is known to be `std::vector`



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```

```
std::vector<int> vec;
std::pair<double, float> pair;
foo(vec, pair);
```

- Parameter `std::vector<T>`, argument `std::vector<int>`
- Match the form `TT<T>`, where `TT` is known to be `std::vector`
- Therefore `T` is `int`



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```

```
std::vector<int> vec;
std::pair<double, float> pair;
foo(vec, pair);
```

- Parameter `std::vector<T>`, argument `std::vector<int>`
- Match the form `TT<T>`, where `TT` is known to be `std::vector`
- Therefore `T` is `int`
- Parameter `std::pair<T,U>`, argument `std::pair<double,float>`



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```

```
std::vector<int> vec;
std::pair<double, float> pair;
foo(vec, pair);
```

- Parameter `std::vector<T>`, argument `std::vector<int>`
- Match the form `TT<T>`, where `TT` is known to be `std::vector`
- Therefore `T` is `int`

- Parameter `std::pair<T,U>`, argument `std::pair<double,float>`
- Match the form `TT<T1,T2>`, where `TT` is known to be `std::pair`



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);
```

```
std::vector<int> vec;
std::pair<double, float> pair;
foo(vec, pair);
```

- Parameter `std::vector<T>`, argument `std::vector<int>`
- Match the form `TT<T>`, where `TT` is known to be `std::vector`
- Therefore `T` is `int`

- Parameter `std::pair<T,U>`, argument `std::pair<double,float>`
- Match the form `TT<T1,T2>`, where `TT` is known to be `std::pair`
- Therefore `T` is `double` and `U` is `float`



General mechanism examples

```
template <class T, class U>
void foo(std::vector<T>, std::pair<T, U>);

std::vector<int> vec;
std::pair<double, float>
foo(vec, pair);
```

- Parameter `std::vector`, argument `std::vector<int>`
- Match the form `TT<T1, T2>`, where `TT` is known to be `std::vector`
- Therefore `T` is `int`
- Parameter `std::pair<T,U>`, argument `std::pair<double,float>`
- Match the form `TT<T1,T2>`, where `TT` is known to be `std::pair`
- Therefore `T` is `double` and `U` is `float`



General mechanism examples

```
// From [temp.over]
template <class T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```



General mechanism examples

```
// From [temp.over]
template <class T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

max(1, 2); // max<int>(int, int)
```



General mechanism examples

```
// From [temp.over]
template <class T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

max(1, 2); // max<int>(int, int)

max(1.0, 2.0); // max<double>(double, double)
```



General mechanism examples

```
// From [temp.over]
template <class T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

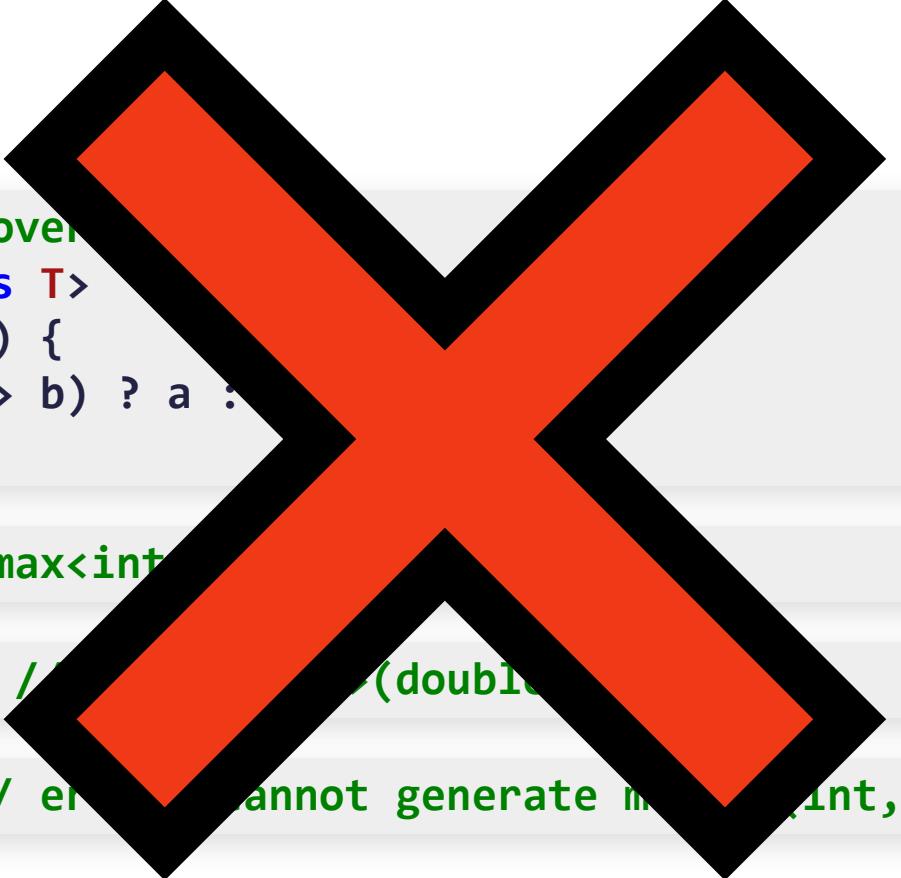
max(1, 2); // max<int>(int, int)

max(1.0, 2.0); // max<double>(double, double)

max(1, 2.0); // error: cannot generate max<T>(int, double)
```



General mechanism examples



```
// From [temp.over.1]
template <class T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

max(1, 2); // max<int>(int, int)

max(1.0, 2.0); // max<double>(double, double)

max(1, 2.0); // error: cannot generate max<int, double>
```



Non-deduced context

```
template <class T>
T max(T a, typename std::type_identity<T>::type b) {
    return (a > b) ? a : b;
}
```



Non-deduced context

```
template <class T>
T max(T a, typename std::type_identity<T>::type b) {
    return (a > b) ? a : b;
}
```



Non-deduced context

```
template <class T>
T max(T a, typename std::type_identity<T>::type b) {
    return (a > b) ? a : b;
}
```



Non-deduced context

```
template <class T>
T max(T a, typename std::type_identity<T>::type b) {
    return (a > b) ? a : b;
}

max(1, 2); // max<int>(int, int)
```



Non-deduced context

```
template <class T>
T max(T a, typename std::type_identity<T>::type b) {
    return (a > b) ? a : b;
}
```

```
max(1, 2); // max<int>(int, int)
```

```
max(1.0, 2.0); // max<double>(double, double)
```



Non-deduced context

```
template <class T>
T max(T a, typename std::type_identity<T>::type b) {
    return (a > b) ? a : b;
}
```

```
max(1, 2); // max<int>(int, int)
```

```
max(1.0, 2.0); // max<double>(double, double)
```

```
max(1, 2.0); // max<int>(int, int)
```



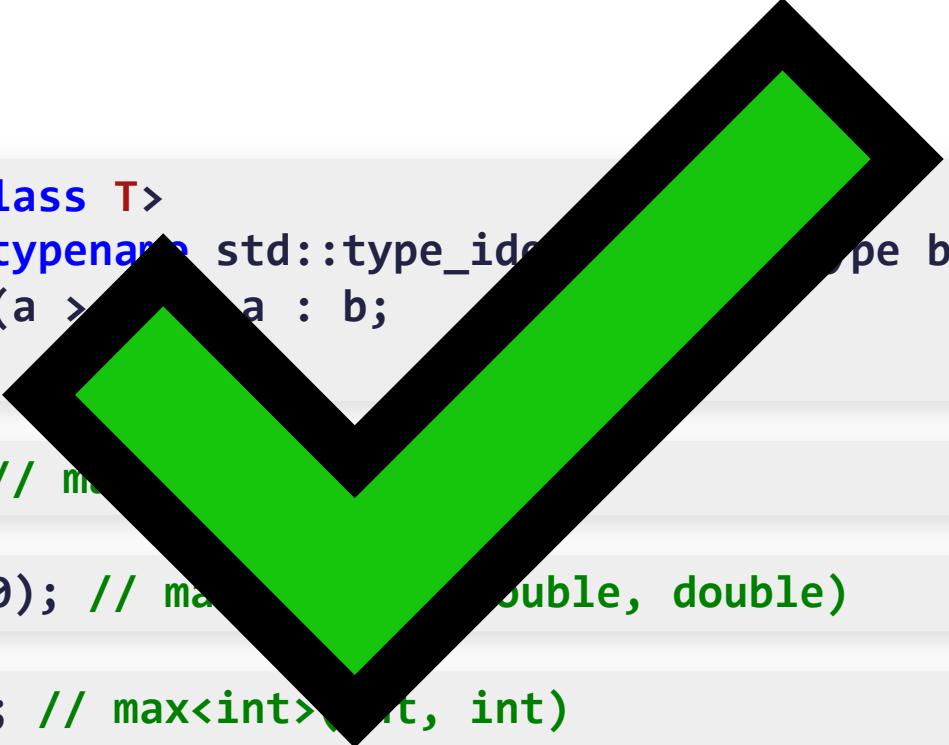
Non-deduced context

```
template <class T>
T max(T a, typename std::type_id<T>::type b) {
    return (a > b) ? a : b;
}

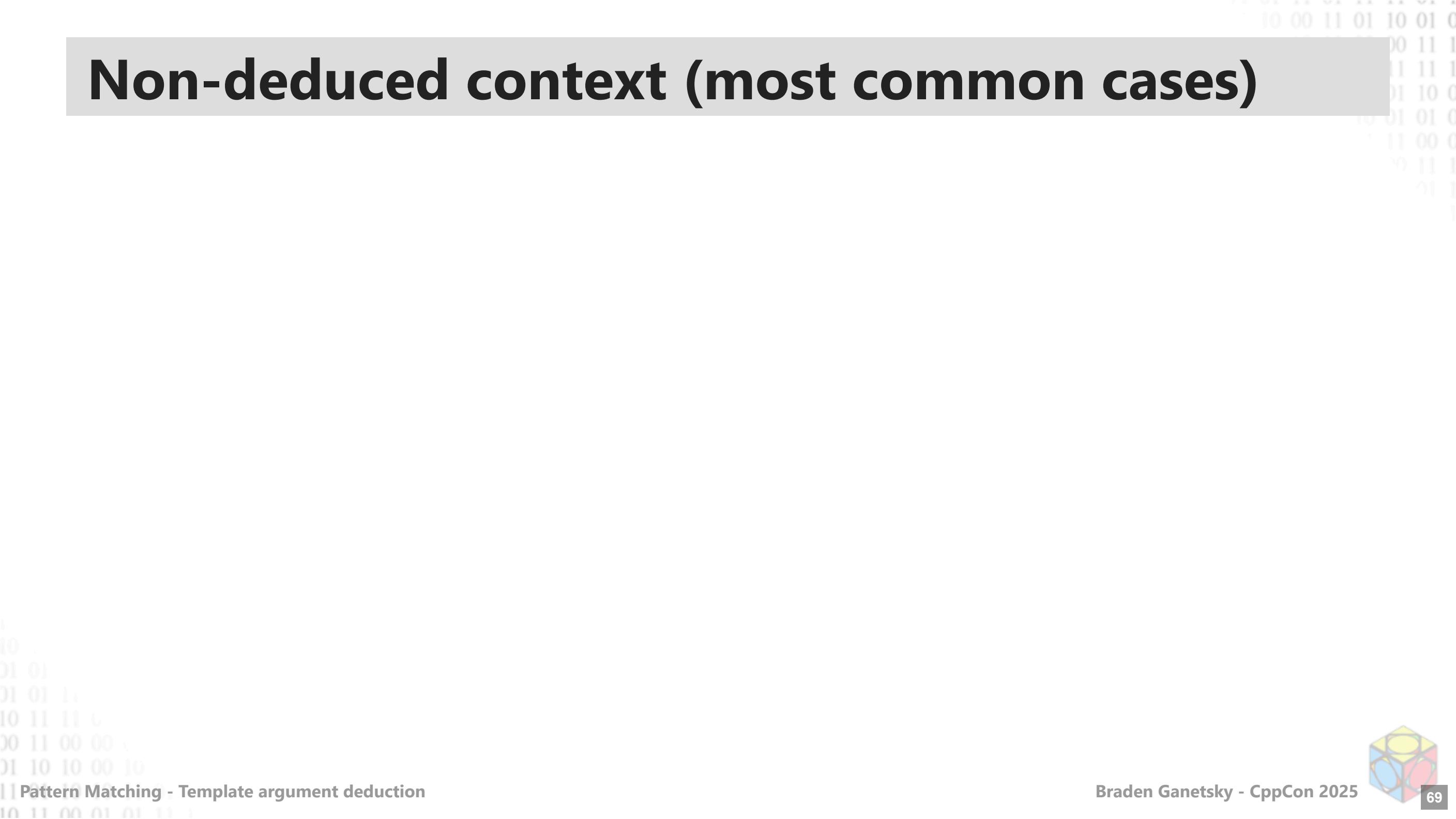
max(1, 2); // max<int>(int, int)

max(1.0, 2.0); // max<double>(double, double)

max(1, 2.0); // max<int>(int, double)
```



Non-deduced context (most common cases)



Non-deduced context (most common cases)

- The "nested-name-specifier" of a qualified type name



Non-deduced context (most common cases)

- The "nested-name-specifier" of a qualified type name
- `decltype` expression



Non-deduced context (most common cases)

- The "nested-name-specifier" of a qualified type name
- `decltype` expression
- Parameter is not `std::initializer_list` but the argument is a braced init list



Non-deduced context

```
template <class T>
struct S {
    using type = int;
};
template <class T>
void foo(typename S<T>::type);
```



Non-deduced context

```
template <class T>
struct S {
    using type = int;
};
template <class T>
void foo(typename S<T>::type);
```



Non-deduced context

```
template <class T>
struct S {
    using type = int;
};
template <class T>
void foo(typename S<T>::type);
```



Non-deduced context

```
template <class T>
struct S {
    using type = int;
};
template <class T>
void foo(typename S<T>::type);
```



Non-deduced context

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

template <class T>
void foo(typename S<T>::type);

foo(123); // error, cannot deduce T
```



Non-deduced context

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

template <class T>
void foo(typename S<T>::type);

foo(123); // error, cannot deduce T
```

- Parameter `typename S<T>::type`, a qualified name



Non-deduced context

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

template <class T>
void foo(typename S<T>::type);

foo(123); // error, cannot deduce T
```

- Parameter `typename S<T>::type`, a qualified name
- "nested-name-specifier" `S<T>`, which is a non-deduced context



Non-deduced context

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

template <class T>
void foo(typename S<T>::type);

foo(123); // error, cannot deduce T
```

- Parameter `typename S<T>::type`, a qualified name
- "nested-name-specifier" `S<T>`, which is a non-deduced context
- Therefore cannot deduce `T`



Non-deduced context

```
template <class T>
struct S {
    using type = int;
};
template <class T>
void foo(typename S<T>::type, T);
```



Non-deduced context

```
template <class T>
struct S {
    using type = int;
};
template <class T>
void foo(typename S<T>::type, T);
```



Non-deduced context

```
template <class T>
struct S {
    using type = int;
};
template <class T>
void foo(typename S<T>::type, T);
```



Non-deduced context

```
template <class T>
struct S {
    using type = int;
};
template <class T>
void foo(typename S<T>::type, T);
```



Non-deduced context

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

template <class T>
void foo(typename S<T>::type, T);

foo(123, 456);
```



Non-deduced context

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

template <class T>
void foo(typename S<T>::type, T);

foo(123, 456);
```

- Parameter `typename S<T>::type`, a qualified name



Non-deduced context

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

template <class T>
void foo(typename S<T>::type, T);

foo(123, 456);
```

- Parameter `typename S<T>::type`, a **qualified name**
- "nested-name-specifier" `S<T>`, which is a **non-deduced context**



Non-deduced context

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

template <class T>
void foo(typename S<T>::type, T);

foo(123, 456);
```

- Parameter `typename S<T>::type`, a **qualified name**
- "nested-name-specifier" `S<T>`, which is a **non-deduced context**
- Therefore cannot deduce `T`



Non-deduced context

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

template <class T>
void foo(typename S<T>::type, T);

foo(123, 456);
```

- Parameter `typename S<T>::type`, a **qualified name**
- "nested-name-specifier" `S<T>`, which is a **non-deduced context**
- Therefore cannot deduce `T`
- Parameter `T`, argument `int`



Non-deduced context

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

template <class T>
void foo(typename S<T>::type, T);

foo(123, 456);
```

- Parameter `typename S<T>::type`, a qualified name
- "nested-name-specifier" `S<T>`, which is a non-deduced context
- Therefore cannot deduce `T`
- Parameter `T`, argument `int`
- Therefore `T` is `int`



Non-deduced context

```
template <class T>
struct S {
    using type = int;
};
template <class T>
void foo decltype(S<T>{});
```



Non-deduced context

```
template <class T>
struct S {
    using type = int;
};
template <class T>
void foo decltype(S<T>{});
```



Non-deduced context

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

template <class T>
void foo decltype(S<T>{});
```



Non-deduced context

```
template <class T>
struct S {
    using type = int;
};
template <class T>
void foo decltype(S<T>{});
```



Non-deduced context

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

template <class T>
void foo decltype(S<T>{});

S<double> s;
foo(s); // error, cannot deduce T
```



Non-deduced context

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

template <class T>
void foo decltype(S<T>{});

S<double> s;
foo(s); // error, cannot deduce T
```

- Parameter `decltype(S<T>{})`



Non-deduced context

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

template <class T>
void foo decltype(S<T>{});

S<double> s;
foo(s); // error, cannot deduce T
```

- Parameter `decltype(S<T>{})`
- A `decltype` expression, which is a **non-deduced context**



Non-deduced context

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

template <class T>
void foo decltype(S<T>{});

S<double> s;
foo(s); // error, cannot deduce T
```

- Parameter `decltype(S<T>{})`
- A `decltype` expression, which is a **non-deduced context**
- Therefore cannot deduce `T`



Non-deduced context

```
template <class T>
void foo(std::vector<T>);
```



Non-deduced context

```
template <class T>
void foo(std::vector<T>);

foo({1, 2, 3}); // error, cannot deduce T
```



Non-deduced context

```
template <class T>
void foo(std::vector<T>);

foo({1, 2, 3}); // error, cannot deduce T
```

- Parameter `std::vector<T>`, argument is a braced init list



Non-deduced context

```
template <class T>
void foo(std::vector<T>);

foo({1, 2, 3}); // error, cannot deduce T
```

- Parameter `std::vector<T>`, argument is a braced init list
- Parameter is not a `std::initializer_list`



Non-deduced context

```
template <class T>
void foo(std::vector<T>);

foo({1, 2, 3}); // error, cannot deduce T
```

- Parameter `std::vector<T>`, argument is a braced init list
- Parameter is not a `std::initializer_list`
- Braced init list argument causes a non-deduced context



Non-deduced context

```
template <class T>
void foo(std::vector<T>);

foo({1, 2, 3}); // error, cannot deduce T
```

- Parameter `std::vector<T>`, argument is a braced init list
- Parameter is not a `std::initializer_list`
- Braced init list argument causes a non-deduced context
- Therefore cannot deduce `T`



Forwarding



Forwarding

- A special mention in the deduction rules



Forwarding

- A special mention in the deduction rules
- "An rvalue reference to a cv-unqualified template parameter"



Forwarding

- A special mention in the deduction rules
- "An rvalue reference to a cv-unqualified template parameter"
- i.e. `T&&`



Forwarding

- A special mention in the deduction rules
- "An rvalue reference to a cv-unqualified template parameter"
- i.e. `T&&`
- Must be a template parameter of the function itself



Forwarding

```
// [temp.deduct.call]/3
template <class T>
int f(T&& heisenreference);
```



Forwarding

```
// [temp.deduct.call]/3
template <class T>
int f(T&& heisenreference);
```



Forwarding

```
// [temp.deduct.call]/3
template <class T>
int f(T&& heisenreference);
```



Forwarding

```
// [temp.deduct.call]/3
template <class T>
int f(T&& heisenreference);

int i = 0;
f(i); // calls f<int&>(int&)
```



Forwarding

```
// [temp.deduct.call]/3
template <class T>
int f(T&& heisenreference);

int i = 0;
f(i); // calls f<int&>(int&)
```



Forwarding

```
// [temp.deduct.call]/3
template <class T>
int f(T&& heisenreference);

int i = 0;
f(i); // calls f<int&>(int&)
```



Forwarding

```
// [temp.deduct.call]/3
template <class T>
int f(T&& heisenreference);

int i = 0;
f(i); // calls f<int&>(int&)
```



Forwarding

```
// [temp.deduct.call]/3
template <class T>
int f(T&& heisenreference);

int i = 0;
f(i); // calls f<int&>(int&)

f(0); // calls f<int>(int&&)
```



Reference collapsing



Reference collapsing

- `T&` & is `T&`



Reference collapsing

- `T& & is T&`
- `T& && is T&`



Reference collapsing

- T& & is T&
- T& && is T&
- T&& & is T&



Reference collapsing

- T& & is T&
- T& && is T&
- T&& & is T&
- T&& && is T&&



Reference collapsing

```
using R1 = int&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```

```
using R1 = int&;
using R2 = R1&&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```

```
using R1 = int&;
using R2 = R1&&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```

```
using R1 = int&;
using R2 = R1&&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```

```
using R1 = int&;
using R2 = R1&&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```

```
using R1 = int&;
using R2 = R1&&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```



Reference collapsing

```
using R1 = int&&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```

```
using R1 = int&&;
using R2 = R1&&;
static_assert(std::same_as<R2, int&&>);
```



Reference collapsing

```
using R1 = int&&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```

```
using R1 = int&&;
using R2 = R1&&;
static_assert(std::same_as<R2, int&&>);
```



Reference collapsing

```
using R1 = int&&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```

```
using R1 = int&&;
using R2 = R1&&;
static_assert(std::same_as<R2, int&&>);
```



Reference collapsing

```
using R1 = int&&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```

```
using R1 = int&&;
using R2 = R1&&;
static_assert(std::same_as<R2, int&&>);
```



Reference collapsing

```
using R1 = int&&;
using R2 = R1&;
static_assert(std::same_as<R2, int&>);
```

```
using R1 = int&&;
using R2 = R1&&;
static_assert(std::same_as<R2, int&&>);
```



Forwarding

```
template <class T>
void foo(T&&);
```



Forwarding

```
template <class T>
void foo(T&&);
```

```
int i = 0;
foo(i);
```



Forwarding

```
template <class T>
void foo(T&&);
```

```
int i = 0;
foo(i);
```

- Parameter `T&&`, argument `int&`



Forwarding

```
template <class T>
void foo(T&&);
```

```
int i = 0;
foo(i);
```

- Parameter `T&&`, argument `int&`
- Match the form `T&&`



Forwarding

```
template <class T>
void foo(T&&);
```

```
int i = 0;
foo(i);
```

- Parameter `T&&`, argument `int&`
- Match the form `T&&`
- Remember, `int&` is `int& &&`



Forwarding

```
template <class T>
void foo(T&&);
```

```
int i = 0;
foo(i);
```

- Parameter `T&&`, argument `int&`
- Match the form `T&&`
- Remember, `int&` is `int& &&`
- Therefore `T&&` is `int& &&`



Forwarding

```
template <class T>
void foo(T&&);
```

```
int i = 0;
foo(i);
```

- Parameter `T&&`, argument `int&`
- Match the form `T&&`
- Remember, `int&` is `int& &&`
- Therefore `T&&` is `int& &&`
- Therefore `T` is `int&`



Forwarding

```
template <class T>
void foo(T&&);
```

```
int i = 0;
foo(i);
```

- Parameter `T&&`, argument `int&`
- Match the form `T&&`
- Remember, `int&` is `int& &&`
- Therefore `T&&` is `int& &&`
- Therefore `T` is `int&`
- `foo<int&>(int&)`



Forwarding

```
template <class T>
void foo(T&&);
```



Forwarding

```
template <class T>
void foo(T&&);

foo(0);
```



Forwarding

```
template <class T>
void foo(T&&);

foo(0);
```

- Parameter **T&&**, argument **int&&**



Forwarding

```
template <class T>
void foo(T&&);

foo(0);
```

- Parameter **T&&**, argument **int&&**
- Match the form **T&&**



Forwarding

```
template <class T>
void foo(T&&);

foo(0);
```

- Parameter **T&&**, argument **int&&**
- Match the form **T&&**
- Therefore **T** is **int**



Forwarding

```
template <class T>
void foo(T&&);

foo(0);
```

- Parameter **T&&**, argument **int&&**
- Match the form **T&&**
- Therefore **T** is **int**
- **foo<int>(int&&)**



Class template argument deduction

Class template argument deduction



Class template argument deduction

- Since C++17



Class template argument deduction

- Since C++17
- "CTAD" ("see-tad")



Class template argument deduction

- Since C++17
- "CTAD" ("see-tad")
- Same as template argument deduction



Class template argument deduction

- Since C++17
- "CTAD" ("see-tad")
- Same as template argument deduction
- ...but for classes (shocking)



Class template argument deduction

- Since C++17
- "CTAD" ("see-tad")
- Same as template argument deduction
- ...but for classes (shocking)
- "resolving a placeholder for a deduced class type"



Class template argument deduction



Class template argument deduction

```
std::pair p{ 1, 2.0 };
```



Class template argument deduction

```
std::pair p{ 1, 2.0 };  
// Deduced as std::pair<int, double>
```



Class template argument deduction

```
std::pair p{ 1, 2.0 };
```

```
// Deduced as std::pair<int, double>
```

```
std::vector<int> foo() {
    // ...
}
```



Class template argument deduction

```
std::pair p{ 1, 2.0 };
```

```
// Deduced as std::pair<int, double>
```

```
std::vector<int> foo() {  
    // ...  
}
```

```
std::vector v = foo();
```



Class template argument deduction

```
std::pair p{ 1, 2.0 };

// Deduced as std::pair<int, double>
```

```
std::vector<int> foo() {
    // ...
}

std::vector v = foo();

// Deduced as std::vector<int>
```



Class template argument deduction



Class template argument deduction

- General mechanism



Class template argument deduction

- General mechanism
- Deduction guides



General mechanism



General mechanism

- Create a theoretical set of function templates, the "guides"



General mechanism

- Create a theoretical set of function templates, the "guides"
- One for each constructor



General mechanism

- Create a theoretical set of function templates, the "guides"
- One for each constructor
- One for each deduction guide*



General mechanism

- Create a theoretical set of function templates, the "guides"
- One for each constructor
- One for each deduction guide*
- Then do overload resolution and template argument deduction



General mechanism

- Create a theoretical set of function templates, the "guides"
- One for each constructor
- One for each deduction guide*
- Then do overload resolution and template argument deduction
- The winning function's return type is chosen



General mechanism



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};

template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};

template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};

template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
std::vector<int> vec;
MyStruct s1{ vec, 1.0 };
// Deduced as MyStruct<int, double>
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
std::vector<int> vec;
MyStruct s1{ vec, 1.0 };
// Deduced as MyStruct<int, double>
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
std::vector<int> vec;
MyStruct s1{ vec, 1.0 };
// Deduced as MyStruct<int, double>
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
std::vector<int> vec;
MyStruct s1{ vec, 1.0 };
// Deduced as MyStruct<int, double>
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
std::vector<int> vec;
MyStruct s1{ vec, 1.0 };
// Deduced as MyStruct<int, double>
```

```
std::pair<char, double> p;
MyStruct s2{ 123, p };
// Deduced as MyStruct<int, char>
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
std::vector<int> vec;
MyStruct s1{ vec, 1.0 };
// Deduced as MyStruct<int, double>
```

```
std::pair<char, double> p;
MyStruct s2{ 123, p };
// Deduced as MyStruct<int, char>
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
std::vector<int> vec;
MyStruct s1{ vec, 1.0 };
// Deduced as MyStruct<int, double>
```

```
std::pair<char, double> p;
MyStruct s2{ 123, p };
// Deduced as MyStruct<int, char>
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
std::vector<int> vec;
MyStruct s1{ vec, 1.0 };
// Deduced as MyStruct<int, double>
```

```
std::pair<char, double> p;
MyStruct s2{ 123, p };
// Deduced as MyStruct<int, char>
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// Deduced as MyStruct<int, std::pair<char, double>>
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// Deduced as MyStruct<int, std::pair<char, double>>
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// Deduced as MyStruct<int, std::pair<char, double>>
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// Deduced as MyStruct<int, std::pair<char, double>>
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// Deduced as MyStruct<int, std::pair<char, double>>
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    MyStruct(std::vector<T>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};
```

```
template <class T, class U>
MyStruct<T, U> __imaginary_function(std::vector<T>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// Deduced as MyStruct<int, std::pair<char, double>>
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    template <class A>
    MyStruct(std::vector<T, A>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);

template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    template <class A>
    MyStruct(std::vector<T, A>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);

template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    template <class A>
    MyStruct(std::vector<T, A>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);

template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    template <class A>
    MyStruct(std::vector<T, A>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);

template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    template <class A>
    MyStruct(std::vector<T, A>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);

template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    template <class A>
    MyStruct(std::vector<T, A>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);

template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    template <class A>
    MyStruct(std::vector<T, A>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);

template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);

std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// Ambiguous!
```



General mechanism

```
template <class T, class U>
struct MyStruct {
    template <class A>
    MyStruct(std::vector<T, A>, U);
    template <class V>
    MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);

template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);

std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// Ambiguous!
```



Deduction guides



Deduction guides

- More specificity with how CTAD behaves



Deduction guides

- More specificity with how CTAD behaves
- Defined after the class template



Deduction guides

- More specificity with how CTAD behaves
- Defined after the class template
- Must align with the available constructors



Deduction guides

```
template <class T, class U> struct MyStruct {  
    template <class A> MyStruct(std::vector<T, A>, U);  
    template <class V> MyStruct(T, std::pair<U, V>);  
};  
template <class T, class U, class A, class V>  
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, std::pair<U, V>>;
```



Deduction guides

```
template <class T, class U> struct MyStruct {  
    template <class A> MyStruct(std::vector<T, A>, U);  
    template <class V> MyStruct(T, std::pair<U, V>);  
};  
template <class T, class U, class A, class V>  
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, std::pair<U, V>>;
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, std::pair<U, V>>;
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, std::pair<U, V>>;
```



```
template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, std::pair<U, V>>;
```



```
template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```



```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, std::pair<U, V>>;
```



```
template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```



```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



```
template <class T, class U, class A, class V>
MyStruct<T, std::pair<U, V>> __imaginary_function(std::vector<T, A>, std::pair<U, V>);
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, std::pair<U, V>>;
```



```
template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```



```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



```
template <class T, class U, class A, class V>
MyStruct<T, std::pair<U, V>> __imaginary_function(std::vector<T, A>, std::pair<U, V>);
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, std::pair<U, V>>;
```



```
template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```



```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



```
template <class T, class U, class A, class V>
MyStruct<T, std::pair<U, V>> __imaginary_function(std::vector<T, A>, std::pair<U, V>);
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, std::pair<U, V>>;
```



```
template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```



```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



```
template <class T, class U, class A, class V>
MyStruct<T, std::pair<U, V>> __imaginary_function(std::vector<T, A>, std::pair<U, V>);
```



```
std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// Deduced as MyStruct<int, std::pair<char, double>>
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, std::pair<U, V>>;
```



```
template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```



```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



```
template <class T, class U, class A, class V>
MyStruct<T, std::pair<U, V>> __imaginary_function(std::vector<T, A>, std::pair<U, V>);
```



```
std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// Deduced as MyStruct<int, std::pair<char, double>>
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, std::pair<U, V>>;
```



```
template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```



```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



```
template <class T, class U, class A, class V>
MyStruct<T, std::pair<U, V>> __imaginary_function(std::vector<T, A>, std::pair<U, V>);
```



```
std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// Deduced as MyStruct<int, std::pair<char, double>>
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, std::pair<U, V>>;
```



```
template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```



```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



```
template <class T, class U, class A, class V>
MyStruct<T, std::pair<U, V>> __imaginary_function(std::vector<T, A>, std::pair<U, V>);
```



```
std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// Deduced as MyStruct<int, std::pair<char, double>>
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, U>;
```



```
template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```



```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



Deduction guides

```
template <class T, class U> struct MyStruct {  
    template <class A> MyStruct(std::vector<T, A>, U);  
    template <class V> MyStruct(T, std::pair<U, V>);  
};  
template <class T, class U, class A, class V>  
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, U>;
```

```
template <class T, class U, class A>  
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```

```
template <class T, class U, class V>  
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, U>;
```



```
template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```



```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



Deduction guides

```
template <class T, class U> struct MyStruct {  
    template <class A> MyStruct(std::vector<T, A>, U);  
    template <class V> MyStruct(T, std::pair<U, V>);  
};  
template <class T, class U, class A, class V>  
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, U>;
```

```
template <class T, class U, class A>  
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```

```
template <class T, class U, class V>  
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
template <class T, class U, class A, class V>  
MyStruct<T, U> __imaginary_function(std::vector<T, A>, std::pair<U, V>);
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, U>;
```



```
template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```



```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```



```
template <class T, class U, class A, class V>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, std::pair<U, V>);
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, U>;
```

```
template <class T, class U, class A>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);
```

```
template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);
```

```
template <class T, class U, class A, class V>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, std::pair<U, V>);
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, U>;
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);

template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);

template <class T, class U, class A, class V>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, std::pair<U, V>);

std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// No matches
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, U>;
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);

template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);

template <class T, class U, class A, class V>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, std::pair<U, V>);

std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// No matches
```



Deduction guides

```
template <class T, class U> struct MyStruct {
    template <class A> MyStruct(std::vector<T, A>, U);
    template <class V> MyStruct(T, std::pair<U, V>);
};

template <class T, class U, class A, class V>
MyStruct(std::vector<T, A>, std::pair<U, V>) -> MyStruct<T, U>;
MyStruct<T, U> __imaginary_function(std::vector<T, A>, U);

template <class T, class U, class V>
MyStruct<T, U> __imaginary_function(T, std::pair<U, V>);

template <class T, class U, class A, class V>
MyStruct<T, U> __imaginary_function(std::vector<T, A>, std::pair<U, V>);

std::vector<int> vec;
std::pair<char, double> p;
MyStruct s{ vec, p };
// No matches
```



Conclusion

C++ has pattern matching!

1
10
01 01
01 01 11
10 11 11 0
00 11 00 00
01 10 10 00 10
11 00 01 01 11 1



C++ has pattern matching!

- Function overload resolution

1
10
01 01
01 01 11
10 11 11 0
00 11 00 00
01 10 10 00 10
11 00 01 01 11 1



C++ has pattern matching!

- Function overload resolution
- Template specializations



C++ has pattern matching!

- Function overload resolution
- Template specializations
- Template argument deduction



C++ has pattern matching!

- Function overload resolution
- Template specializations
- Template argument deduction
- Class template argument deduction



I read the standard so you don't have to

1
0
01 01
01 01 11
10 11 11 0
00 11 00 00
01 10 10 00 10
11 00 01 01 11 1



I read the standard so you don't have to

- [over.match]



I read the standard so you don't have to

- [over.match]
- [temp.names], [temp.spec.partial], [temp.spec]



I read the standard so you don't have to

- [over.match]
- [temp.names], [temp.spec.partial], [temp.spec]
- [temp.deduct]



I read the standard so you don't have to

- [over.match]
- [temp.names], [temp.spec.partial], [temp.spec]
- [temp.deduct]
- [over.match.class.deduct], [dcl.type.class.deduct],
[temp.deduct.guide]



I read the standard so you don't have to

- [over.match]
- [temp.names], [temp.spec.partial], [temp.spec]
- [temp.deduct]
- [over.match.class.deduct], [dcl.type.class.deduct],
[temp.deduct.guide]
- **Cppreference is also an amazing resource**



Takeaways



Takeaways

- We already have pattern matching!



Takeaways

- We already have pattern matching!
- The general rules can be quite elegant



Takeaways

- We already have pattern matching!
- The general rules can be quite elegant
- ...but the edge cases can get gnarly



Takeaways

- We already have pattern matching!
- The general rules can be quite elegant
- ...but the edge cases can get gnarly
- **Analogue reasoning about your code is a muscle**



Takeaways

- We already have pattern matching!
- The general rules can be quite elegant
- ...but the edge cases can get gnarly
- Analogue reasoning about your code is a muscle
- Get curious, exercise that muscle



Takeaways

- We already have pattern matching!
- The general rules can be quite elegant
- ...but the edge cases can get gnarly
- Analogue reasoning about your code is a muscle
- Get curious, exercise that muscle
- Appreciate how far C++ has come



Thank you!

Braden Ganetsky

braden@ganets.ky
GitHub @k3DW

