Class Template Argument Deduction

A New Abstraction

Zhihao Yuan zhihao.yuan@depaul.edu

Before	C++17
`std::make_move_iterator(it)`	`std::move_iterator(it)`

Before	C++17
`std::make_move_iterator(it)`	`std::move_iterator(it)`
`std::make_reverse_iterator(it)`	`std::reverse_iterator(it)` ??

```
std::vector{ 1, 2, 3 }

std::set{ "std::string"s }
```

```
std::vector{ 1, 2, 3 }

std::set{ "std::string"s }

map{ { "key1"s, 1 }, { "key2"s, 2 } }  ??
```

```
std::vector{ 1, 2, 3 }

std::set{ "std::string"s }

map{ pair{ "key1"s, 1 }, pair{ "key2"s, 2 } }  ???
```

Language details

Part. 1

Automatic deduction

Special Members

compiler implicitly declares

user declares

	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Automatic deduction

- *template-name* is a new *simple-type-specifier*

```
vector v = { 1, 2, 3 };
```

- may come with *nested-name-specifier* – `std::vector`

Automatic deduction

```
template <class T, class Alloc = allocator<T>>
struct vector
{

    vector(size_type, const T&, const Alloc& = Alloc());
};
vector v(10, 'a');
```

Automatic deduction

```
struct vector
{
      template <class T, class Alloc = allocator<T>>
      vector(size_type, const T&, const Alloc& = Alloc());
};
vector v(10, 'a');
```

Automatic deduction

```
struct vector
{
      template <class T, class Alloc = allocator<T>>
      vector(size_type, const T&, const Alloc& = Alloc());
};
vector v(10, 'a');  // T = char, Alloc = allocator<char>
```

Deduction candidates

1. Constructors with compiler synthesized template parameter lists

Automatic deduction: Problems

```
template <typename T>
struct A
{
     template <typename F>
     A(T&&, F&&);
};
A x(3, []{});
```

Automatic deduction: Problems

```
struct A
{
     template <typename T, typename F>
     A(T&&, F&&);
};
A x(3, []{});
```

Automatic deduction: Problems

```
struct A
{
     template <typename T, typename F>
     A(T&&, F&&);  // not a forwarding reference
};
A x(3, []{});
```

Where this feature can be used?

Wherever “`auto`” can be used, plus minus exceptions, notably

- `new ClassTemplate{ a, b }`, but not “`new auto{ a, b }`”
- `ClassTemplate const v =` ..., but no &, &&, *
-
```
std::vector f()  // not allowed (yet?)
{
    return { 1, 2, 3 };
}
```
- `[](std::vector v) { … }  // no either`

Where this feature can be used?

`std::vector v = { 1, 2 }, w(start, end);` is also allowed,

- and works in the “`auto`” way
- but `CT v = {…}` and `CT v{…}` are different in a different way which differs from `auto v = {…}` and `auto v{…}`

auto

```
optional opt = 42;

auto v(opt);

new auto(opt)

auto(opt)   ?
```

template name

```
optional opt = 42;

optional v(opt);

new optional(opt)

optional(opt)
```

Same

auto	template name
`auto v{opt};`	`optional v{opt}`
`new auto{opt}`	`new optional{opt}`
`auto{opt}` ?	`optional{opt}`

Differences

auto	template name
`auto v = { e1, e2 };`	`optional v = { e1, e2 };`
~~`auto v{ e1, e2 };`~~	`optional v{ e1, e2 };`

Deduction candidates

1. Constructors with synthesized template parameter lists
2. The $\textit{copy deduction candidate}$

Deduction candidates

1. Constructors with synthesized template parameter lists
2. The *copy deduction candidate*
3. Default constructor if no constructor declared

C++14

```
std::sort(bg, ed, std::greater<>());
```

C++14

```
std::sort(bg, ed, std::greater<>());
```

C++17

```
std::sort(bg, ed, std::greater());
```

```
template <class T = void>
struct greater
{
        // …
};
template <>
struct greater<void> { /* … */ };
```

```
struct greater
{
	template <class T = void> greater();
};
template <>
struct greater<void> { /* … */ };
```

```
template <class T = void>
struct greater;
```

```
template <>
struct greater<void> { /* … */ };
```

```
template <class T, class Alloc = allocator<T>>
struct vector
{
    template <class InputIt>
    vector(InputIt, InputIt);
};
vector v(begin(buf), end(buf));
```

```
struct vector
{
    template <class T, class = allocator<T>, class InputIt>
    vector(InputIt, InputIt);
};
vector v(begin(buf), end(buf));
```

```
struct vector
{
        template <class T, class = allocator<T>, class InputIt>
        vector(InputIt, InputIt);
};
vector v(begin(buf), end(buf));
```

Deduction guide

```
struct vector
{
    template <class T, class InputIt>
    vector(InputIt, InputIt) -> vector<ValueType<InputIt>>;
};
vector v(begin(buf), end(buf));
```

Deduction guide

```
struct vector

{ /* … */ };


template <class T, class InputIt>

vector(InputIt, InputIt) -> vector<ValueType<InputIt>>;

vector v(begin(buf), end(buf));
```

Deduction guide

deduction-guide:

`explicit`$_{opt}$ *template-name* `(` *parameter-declaration-clause* `) ->` *simple-template-id* `;`

- a declaration
- appears in the scope where class template X is declared
- *template-name* is X
- *simple-template-id* is specialization for X

Deduction guide: Limitations

deduction-guide:

`explicit`$_{opt}$ *template-name* `(` *parameter-declaration-clause* `) ->` *simple-template-id* `;`

```
template <typename T>
Param(T const&) -> param_type_t<T>;  ??
```

Deduction candidates

1. Constructors with synthesized template parameter lists
2. The *copy deduction candidate*
3. Default constructor if no constructor declared
4. *deduction-guide*

}

for automatic deduction

Deduction guide: Limitations

The automatic deduction candidates are never “replaced”.

```
template <typename T>
struct Value { Value(T); };


template <Integral T>
Value(T) -> Value<T>;
Value(3.14)  ??
```

A deduction guide toolbox

Part. 2

Good for…

```
template <class T, class Comp = less<T>, class Alloc = allocator<T>>
struct set
{
      set(initializer_list<T>, Comp const& = {}, Alloc const& = {});
      set(initializer_list<T>, Alloc const&);
};



set v({ 1, 3, 4 }, a);  ?
```

Disambiguation

```
template <class T, class Comp = less<T>, class Alloc = allocator<T>>
struct set
{
      set(initializer_list<T>, Comp const& = {}, Alloc const& = {});
      set(initializer_list<T>, Alloc const&);
};
template <Allocator A>
set(initializer_list<T>, A const&) -> set<T, A>;
set v({ 1, 3, 4 }, a);  ✓
```

```
template <class T>
struct optional
{
    template <class U>  // constrained
    optional(U&&);
};
optional opt = 42;
```

```
template <class T>
struct optional
{ /* … */ };

template <class T>
optional<decay_t<T>> make_optional(T&&);
optional opt = 42;
```

```
template <class T>
struct optional
{ /* … */ };


template <class T>
optional(T&&) -> optional<decay_t<T>>;  ??
optional opt = 42;
```

Take by value

```
template <class T>
struct optional
{ /* … */ };


template <class T>
optional(T) -> optional<T>;
optional opt = 42;
```

Special Members

compiler implicitly declares

user declares

	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Take by value

```
template <class T>
optional(T) -> optional<T>;
```

Essence of deduction guide

1. Dissociates overload resolution from application

Benefits: Replacing all these

`C(T&)`

`C(T const&)`

`C(T&&)`

`C(T const&&)`

…

The *copy deduction candidate* is generated from `C(C)`.

Take by value

```
template <class T1, class T2>
  pair(T1, T2) -> pair<T1, T2>;
template <class... T>
  tuple(T) -> tuple<T>;
template <class... T>
  array(T...) -> array<same_type_t<T...>, sizeof...(T)>;
auto ls = array{ 1, 3, n };   // similar to std::vector
```

Suppress an automatic candidate

```
template <typename T>
struct Value { Value(T); };

template <Integral T>
  Value(T) -> Value<T>;
template <template T> requires not Integral<T>
  Value(T) = delete;   ?
```

Simulate `= delete;`

```
template <typename T>
struct Value { Value(T); };


template <Integral T>
  Value(T) -> Value<T>;
template <template T> requires not Integral<T>
  Value(T) -> Value<nonesuch>;
```

Suppress… all automatic candidates

```
template <typename T>

struct X

{

     X(size_t, T const&);

     X(std::initializer_list<T>);

};
```

Step 1. rename T to T_

```
template <typename T_>
struct X
{

    X(size_t, T const&);
    X(std::initializer_list<T>);
};
```

Step 2. typedef T to make it non-deducible

```
template <typename T_>

struct X

{

      using T = identity_t<T_>;

      X(size_t, T const&);

      X(std::initializer_list<T>);

};
```

```
template <typename T>
struct identity
{
      using type = T;
};
template <typename T>
using identity_t = typename identity<T>::type;
```

Benefits: Position-based overriding

```
template <typename T_>
struct X
{
        using T = identity_t<T_>;
        X(size_t, T const&);
        X(std::initializer_list<T>);
};
template <typename T, typename U>
X(T, U) -> X<U const>;   // overrides more specialized size_t
```

Benefits: Enabling all prioritization tricks

- priority tags
- `...`, literally
- and more

Dictate

```
template <typename T>
struct ostream_joiner
{
      ostream_joiner(ostream&, T&& delimiter);
};
char buf[] = " + ";
ostream_joiner j(cout, buf);  // want string not char const*
```

Dictate

```
template <typename T>

struct ostream_joiner

{

     ostream_joiner(ostream&, T&& delimiter);

};

ostream_joiner(ostream, char const*) ->
ostream_joiner<string>;

ostream_joiner j(cout, buf);  ✓
```

Deduction guide

deduction-guide:

`explicit`$_{opt}$ *template-name* `(` *parameter-declaration-clause* `) ->` *simple-template-id* `;`

Dictate

```
template <typename T>
struct ostream_joiner
{
      ostream_joiner(ostream&, T&&);
};
ostream_joiner(ostream, char const*) -> ostream_joiner<string>;
// ostream_joiner(ostream&, string&&)
ostream_joiner j(cout, buf);   // invites an implicit conversion
```

Essence of deduction guide

1. Dissociates overload resolution from application
2. Uses overload resolution to select specializations

Design by having CTAD in mind

Part. 3

Design

How people started with

```
template <class T>
optional(T&&) -> optional<decay_t<T>>;
```

Is that designed?

How would I design that deduction guide

User: `optional opt = 42;` doesn’t work!

Me: OK, which specialization of `optional` you expected when you wrote that statement?

A: `optional<int>`

Q: No problem, does the following deduction guide

```
optional(int) -> optional<int>;
```

solve your problem?

A: Huh, that doesn’t even deduce `optional('a')`.

Continued

Q: Sure… I now have

```
optional(int) -> optional<int>;

optional(char) -> optional<char>;
```

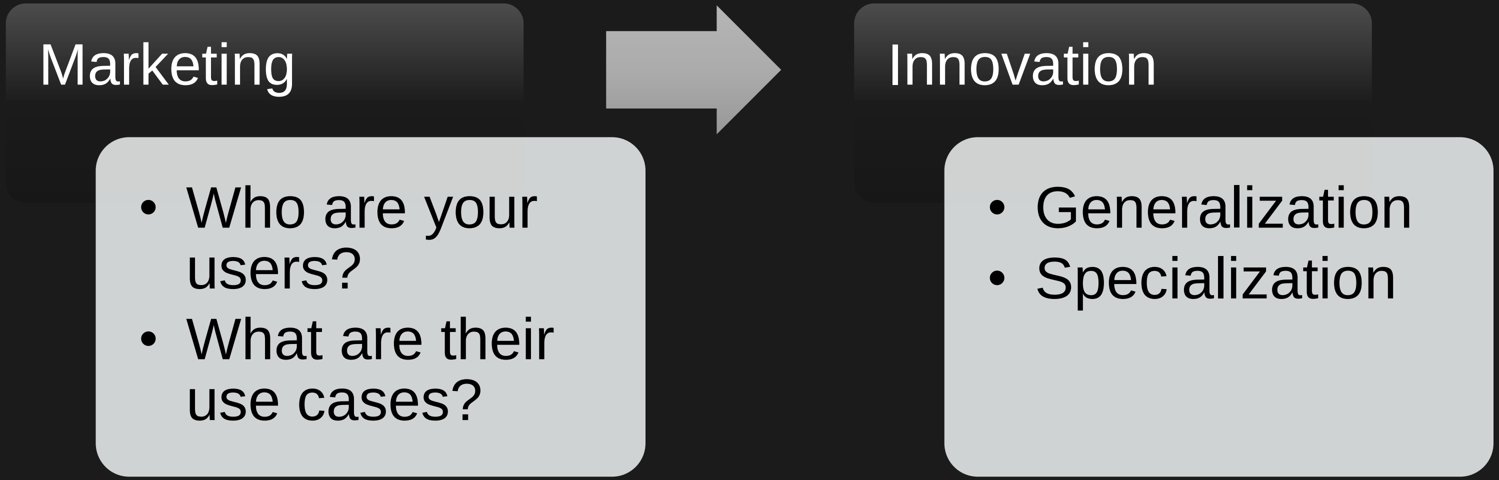
… Generalize those to

```
template <typename T>

optional(T) -> optional<T>;
```

How is that?

Writing a deduction guide



Marketing

- Who are your users?
- What are their use cases?

Innovation

- Generalization
- Specialization

Why you write a deduction guide?

?

- To fix a bug caused by automatic deduction

?

- To enable a non-deduced case

?

- To give the class template a new interface

Essentially,

- What class template argument deduction gives you is a new kind of interface for a class template; the basic form is *template-name*`(args…)`.

Example

```
template <typename T, typename Alloc = std::allocator<T>>
struct Vector
{
     Vector(initializer_list<T>, Alloc const& = {});
};


Vector({ 1, 2 }, mallocobj) ?? Vector({ 1, 2 }, stdalloc)
```

Example

```
template <typename T, typename Alloc = PolyAlloc<T>>
struct Vector
{
     Vector(initializer_list<T>, Alloc const& = {});
};


Vector({ 1, 2 }, pool) ??? Vector({ 1, 2 }, new_delete)
```

Again

Q: What’s your favorite specialization when seeing `Vector(ls, pool)` ?

A: `Vector<T, PolyAlloc>`.

Q: Let me suppress all the automatic candidates first, and then…

```
template <typename T>
Vector(T, PolyAlloc) -> Vector<T, PolyAlloc>
```

The new interface vs. the traditional interface

```
sort<vector<double>::iterator,
     greater<double>>(begin(v), end(v), {});  ??
```

Build specializations to implement the interfaces

```
Vector{ true, true, false }

Vector(initializer_list<bool>) -> Vector<__real_bool_tag>;


Vector<bool>()  ?

Vector(type_c<bool>)

Vector(type<bool>) -> Vector<__real_bool_tag>;
```

Conclusion

- Design your entirely class template as one overload set

Questions?

Resources

- Hinnant, Howard. *Everything You Ever Wanted To Know About Move Semantics (and then some).* https://howardhinnant.github.io/bloomberg_2016.pdf
- Yuan, Zhihao. *Disambiguation the Black Technology.* https://speakerdeck.com/lichray/disambiguation-the-black-technology