Call side return type deduction

Thomas Mejstrik

June 17, 2025

Contents

1 Motivation

For some functions it would be nice, if there were an easy way to have access to the expected return type. In Section 2 we list some use cases where this may be handy.

Deducing the return type is already partly possible for class types, by adding a user-defined conversion function

```
struct S {
    template< typename Out >
    operator Out() { return Out{}; }
};
```

For all other types, this is currently not possible. E.g. the numbers in `std::numbers` are variable templates, and thus, one probably cannot add such a facility to these without ABI break.

- We propose a language extension to get access to return type, similar as it is done in conversion constructors.
- Some cases are eliminated where we need to write out type names multiple times. Thus, we have less possibilities for bugs.

1.1 Notation

I propose to use a keyword, to opt in to deduction at call site. *Note: In the following code, the keyword is `deduce`.*

2 Examples

2.1 Instead of `auto` for non-static data members

```
struct S {
    //auto i = std::numeric_limits< int >::max();
    // not allowed

    int i = std::numeric_limits<>::max< deduce >();
};
```

2.2 Instead of `auto` for defaulted function arguments

```
//void func( auto i = std::numeric_limits< int >::max() );
// means something different

void func( int i = std::numeric_limits<>::max< deduce >() );
```

2.3 std::numeric_limits, std::numbers

For <numbers> this is straightforward.

```
long double _ = std::numbers::pi< deduce >;
long double _ = std::numbers::pi_d;  // could be added, similar to ::pi_v
                                     // and in the mood of ::pi
```

2.4 Static auto cast

Opt-in to implicit casts

```
// Double/Float strong types for double/float
Double func( Double );
Float f = 1;
func( f );  // compile time error
func( static_cast< deduce >( f ) );
// opt-in to implicit cast
```

On SO this was a solution for the following code (https://stackoverflow.com/questions/5693432)

```
int *ptr = static_cast< deduce >( malloc( sizeof(int) ) );
```

2.5 Random objects

```
TEST( suitename, testname ) {
    EXPECT_TRUE( func( random_T() ) );
}
```

3 Wanted behaviour

- Return type deduction should be visible in source code. This is important, since return type deduction is not something common in C++, and thus users may miss it when reading source code.
- No behaviour change for existing code
- No ABI break if we add the facility to existing functions

```
template< typename T > T foo() {}

func();                       // compilation error
double _ = foo< int >();      // deduction: int
double _ = foo();             // compilation error
double _ = foo< deduce >();   // deduction: double
```

```
(void) foo< deduce >();       // deduction: void
foo< deduce >();              // compilation error
auto _ = foo< deduce >();     // compilation error
/** ---------------- **/
template< typename T > T bar( T ) {}

bar( 1.f );                         // deduction: float
double _ = bar< int >( 1.f );       // deduction: int
double _ = bar( 1.f );              // deduction: float
double _ = bar< deduce >( 1.f );    // compilation error
```

3.1 Extensions using this idea

This notation opens up nice new possibilities. `deduce` can have additional meaning, and we can allow other keywords here too.

Property (1) `deduce` tells the compiler that this argument shall be deduced by incorporating the return type, but a later one is user provided.

```
template< typename T, typename U >
U func( T t, U u );

int f = func< deduce, float >( 1., 2. );
// deduction: T = double, U = float
```

Property (2) `default` tells the compiler that a default argument shall be used, but a latter one is passed explicitly.

```
template< typename T , typename U = int, typename V > void gon( T, V ) {}

gon( 1., 0 );  // deduction: T = double, U = int, V = int

gon< deduce, default, float >( 1., 0 );
// deduction: T = double, U = int, V = float

void func( int i = 0, double d = 1. );  // declaration
func( default, 2. );
// func( 0, 2. );
```

Property (3) By providing `deduce` at the side of the definition, one could also unconditionally enable return type deduction for certain functions

```
template< typename deduce T = double > T func();
int i = func();
// int i = func< deduce >();
// deduction: int

func();
// func< deduce >();
// deduction: double

template< typename deduce T > T bar();
bar();
// func< deduce >();
// compilation error
```

4 Additional examples

If other parts of this proposal are accepted, here are some examples for those.

4.1 Wrappers (Property 2)

Removes the need to write some wrapper functions.

```
void func( Arg1 a1, Arg2 a2 = x2, Arg3 a3 = x3 );
void func_wrapper( Arg1 a1, Arg3 a3 ) {
    Arg2 a2 = a2;
    return func( a1, a2, a3 );
}

func_wrapper( a1, a3 );
func( a1, default, a3 );
```

4.2 Default arguments not at the end

```
template< typename T = int, typename U >
struct S {};

S< default, int > s{};

void func( int a1 = 1, int a2 );
func( default, 2 );
```

5 Open questions

- This only makes sense for template functions/template variables. Is there any sense in this code

  ```
  template< typename T >
  struct S {
     T x;
  };

  S< deduce > s{1};
  ```

  Note: The paper does not propose anything to make this an allowed syntax

- List of possible keywords instead of `deduce`
  - `std::deduce`: Since we cannot use a positional keyword, `deduce` cannot be used. We could add it to namespace `std` and use `std::deduce`. It is strange, to have a keyword, in the namespace `std` (although we have this de-facto already with `std::declval`).
  - `auto`: This keyword is not used currently in this context. I fear, that there may be a better usage for the keyword `auto` in this context.
  - `return`: Strange keyword, but carries the meaning of "something from the return stuff". But, does not make sense for **Property (1)**.
- List of possible keywords instead of `default`
  - `_`: (Underscore)
- Syntax for **Property (3)** questionable.

6 Alternative solutions

6.1 Change of meaning of existing code 1

```
template< typename T > T func() {}

func();                     // compilation error
double d = func< int >();   // deduction: int
double d = func();          // deduction: double (*)
/** ---------------- **/
template< typename T > T bar( T ) {}

bar( 1.f );                       // deduction: double
double d = bar< int >( 1.f );     // deduction: int
double d = bar( 1.f );  //  deduction: float  (should be a compilation error) (**)
```

Description

- When the return type is templated, and not provided at call site, then it gets deduced.

Assessment

- Either changes the meaning of existing code (Example *), or leads to inconsistencies (Example **)

6.2 Additional positional keyword and trailing return type

```
template< typename T > deduce foo() -> T {}
template< typename T > deduce bar( T ) -> T {}

foo();                     // compilation error
double d = foo< int >();   // deduction: int
double d = foo();          // deduction: double

bar( 1.f );                       // deduction: double
double d = bar< int >( 1.f );     // deduction: int
double d = bar( 1.f );            //  compilation error
```

This resembles the syntax of trailing return type. If a function has return type deduce and a trailing templated return type, then the return type is deduced.

Assessment

- I would guess, this is an ABI break.