

P3747R1

Deducing target types for function templates

Document No.: P3747R1
Date: 2026-09-30
Project: Programming Language C++
Audience: EWGI
Author: Thomas Mejstrik
Email: tommsch@gmx.at

Contents

1. Revision history	2
2. Motivation	2
2.1. Notation	2
2.2. Design guidelines	2
3. How type deduction works	2
3.1. General rule	2
3.2. cvref qualifiers	3
3.3. Partial deduction lists	3
3.4. Unconditional target type deduction	3
3.5. Default arguments	4
3.6. Variable templates: Same as for functions	4
3.7. Class templates: Not applicable	4
4. Usage examples	4
4.1. Less duplication of type names	4
4.2. Remedy <code>std::accumulate</code> and <code>consorts</code> bug	5
4.3. Generic programming	5
4.4. Conversions and <code>auto</code>	6
4.5. Deducing overloads for <code>std::numeric_limits</code> , <code>std::numbers</code>	7
5. Name bike-shedding	8
A. Why writing a proxy is not a good solution	8
B. Existing practice in other languages	8

We propose a facility to incorporate the target type at call site into the type deduction machinery for template functions and template variables. No new type deduction rules are proposed. No ABI break happens.

1. Revision history

R1

- Changed deduction method to follow the standard procedure of type deduction as done in templated user-defined conversion functions
- Added a lot of examples

R0

- Initial version, titled *Call side return type deduction*,
- Presented in Sofia 2025.
- The paper received an informal encouragement poll that received extensive discouragement (2 encourage, 16 discourage).

2. Motivation

For some situations it would be nice,

- if there were an easy way to have access to the requested target type, or
- not to be forced to duplicate type names (“always-auto-movement”).

This is already partly possible for class types, by adding a user-defined conversion function, see Listing 1. But, relying on this needs boiler plate code, plus gives the user foot-guns.

Listing 1:

```
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 the target type, similarly as it is done in templated user defined conversion functions. Apart from syntactic sugar, this removes some easy to happen bugs, and enables new functionality, see Section 4.

2.1. Notation

Note: In the following listings, the word *Deduce* implies that target type deduction shall occur. Name bikeshedding comes later.

2.2. Design guidelines

- Target type deduction should be visible in source code. This is important, since target type deduction is not common in C++, and thus users may miss it when reading source code.
- No behaviour change for existing code
- No ABI break
- No new type deduction rules

3. How type deduction works

3.1. General rule

This proposal introduces **no new deduction rules**. All deduction is performed by existing, unmodified language machinery:

1. The target type is deduced as for a templated user-defined conversion functions [`temp.deduct.conv`], only if *Deduce* is supplied at the call site.
2. Ordinary function arguments are deduced by the standard rules for function template argument deduction from a function call [`temp.deduct.call`].

3. If deduction fails or is ambiguous the compilation shall fail.

The design intentionally reuses the conversion-function model so that the answer to every “*what happens if...?*” question is: “*Whatever happens today for a class with a templated conversion function.*”

```
template< typename T > T foo() {}
foo(); // compilation error
double _ = foo< int >(); // T = int
double _ = foo(); // compilation error
double _ = foo< Deduce >(); // T = double
(void) foo< Deduce >(); // T = void
foo< Deduce >(); // compilation error
auto _ = foo< Deduce >(); // compilation error

template< typename T > T bar( T ) {}
bar( 1.f ); // T = float
double _ = bar< int >( 1.f ); // T = int
double _ = bar( 1.f ); // T = float
double _ = bar< Deduce >( 1.f ); // compilation error
auto _ = bar< Deduce >( 1 ); // compilation error

// binary operators
struct S {
    template< typename T > operator T() { return {}; }
} s;
struct A{} a;
void operator+( A, int ) {}
foo< Deduce >() + 1; // same behaviour as: s + 1
a + foo< Deduce >(); // same behaviour as: a + s
foo< Deduce >() + foo< Deduce >; // same behaviour as: s + s
```

3.2. cvref qualifiers

It is possible to put cvref qualifiers in front of *Deduce*. In this case, deduction works (again) as for templated user defined conversion functions.

```
template< typename T > cvref(A) T baz();
cvref(C) _ = baz< cvref(B) Deduce >();

// deduces like

struct S {
    template< typename T > operator cvref(B) T();
};
cvref(A) S s;
cvref(C) _ = s;
```

3.3. Partial deduction lists

If a function takes multiple template arguments, *Deduce* applies to only those where it is supplied.

```
template< typename T, typename U > T foo( U ) {}
double _ = foo< Deduce >( 1 ) // T = double, U = int
```

3.4. Unconditional target type deduction

By providing *Deduce* in front of *typename* (or *class*) at the side of the declaration, one unconditionally enables target type deduction. Note: A *Deduce* in a declaration cannot be prefixed with cvref qualifiers.

```
template< Deduce typename T > T jia( T ) {}
jia( 1.f ); // compilation error
auto _ = jia( 1.f ); // compilation error
double _ = jia< int >( 1.f ); // T = int
```

```
double _ = jia< Deduce >( 1. ); // T = double
double _ = jia( 1.f ); // compilation error
double _ = jia< Deduce >( 1.f ); // compilation error
```

3.5. Default arguments

If a template argument is *Deduce* (at call side) any default template argument for that parameter is ignored. (“Deduce is an argument, not the absence of one.”).

```
template< typename T = double > T baz();
int i = baz(); // T = double
int i = baz< Deduce >(); // T = int
baz< Deduce >(); // compilation error
auto _ = baz< Deduce >(); // compilation error
```

3.6. Variable templates: Same as for functions

```
template< Deduce typename T >
constexpr T pi_a = T(3.1415926535897932385L); // always deducing variable template

float a = pi_a;
```

3.7. Class templates: Not applicable

Deduce is not allowed in class template instantiations and declarations since it makes no sense; A constructor has no return type in C++.

```
template< typename T >
struct S {
    T x;
};
// S< Deduce > s{1}; // Not allowed
```

4. Usage examples

4.1. Less duplication of type names

Instead of “auto” for non-static data members In structs, *auto* is only allowed for static member variables.

```
struct S {
    //auto i = std::numbers::pi_v< float >; // not allowed
    float i = std::numbers::pi_v< Deduce >;
};
```

Constructor member-initializer lists Less potential for bugs when the type definition and its usage are located in different places.

```
// Arc.h
class Arc {
    float alpha_, r_;
public:
    Arc();
};

// Before: Arc.cpp
Arc::Arc() :
    : alpha_( std::numbers::pi_v< float > ) // type name is spelled out in two files
    , r_( 0 )
{}

// After: Arc.cpp
```

```
Arc::Arc() :
    : alpha_( std::numbers::pi_v< Deduce > ) // type name is spelled out in two files
    , r_( 0 )
    {}
```

Refactoring function signatures Changing function signatures is safer. Note: `std::numeric_limits<>::max< Deduce >()` in line (*) is defined in a later example.

```
void set_watermark( std::uint16_t level );

// Before: The reader must verify uint16_t matches
set_watermark( std::numeric_limits< std::uint16_t >::max() );

// After: Types are deduced from the parameter and cannot mismatch
// If set_watermark is refactored to use another type, user code is still correct
set_watermark( std::numeric_limits<>::max< Deduce >() ); // (*)
```

Static auto cast Opt-in to implicit casts:

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

Prevent implicit casts, or “`static_cast< Deduce >` cast is safer than `static_cast`”.

```
int i = 1234;
short s = static_cast< char >( i ); // s == -46, not 1234!
short s = static_cast< Deduce >( i ); // s == 1234
```

On SO this “operator” is discussed at <https://stackoverflow.com/q/5693432>

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

Instead of auto for defaulted function arguments

```
void func( auto i = std::numeric_limits<>::max< int >() );
// means something different
void func( float i = std::numbers::pi_v< float > ); // type name duplication
void func( float i = std::numbers::pi_v< Deduce > );
```

In particular, this is not solvable by reflection. Indeed, at the point where `Deduce` is used we cannot reflect on `func` yet because its declaration is not finished.

4.2. Remedy `std::accumulate` and consorts bug

The 0-literal `std::accumulate` bug is well-know. A function with a slightly different signature prevents this bug. We use `Deduce`, and move the accumulator-type decision from a position where it is easy to get wrong, to the target type which is typically correct.

```
template< Deduce typename T, typename InputIt >
T accu( InputIt first, InputIt last, T init );

double y = accu( vec.begin(), vec.end(), 0 ); // compilation error
```

Note: We do not propose to change the signature of `std::accumulate`.

4.3. Generic programming

Less decltype

```
// Before: Precision must be spelled out, or decltype:
float circumference(auto d) const { return d * std::numbers::pi_v< decltype(d) >; }

// After:
float circumference(auto d) const { return d * std::numbers::pi_v< Deduce >; }
```

Argument parsing: More more generic lambdas

```
struct Config {
    int          threads;
    double       timeout;
    std::size_t  maxmem;
    std::string  logfile;
};
```

```
// Before: Classic typed parser
template< typename T >
T parse( std::string_view raw );

void load( Config& cfg, const ArgMap& args ) {
    // The target type is repeated on every line and can silently mismatch:
    cfg.threads = parse<int>      ( args["--threads"] );
    cfg.timeout = parse<double>   ( args["--timeout"] );
    cfg.maxmem  = parse<int>      ( args["--maxmem"] );  // BUG: int, not size_t
    cfg.logfile = parse<std::string>( args["--logfile"] );
}

// After: Framework knows nothing about the target types
void load( Config& cfg, const ArgMap& args, auto parse ) {
    cfg.threads = parse( args["--threads"] );  // T = int
    cfg.timeout = parse( args["--timeout"] );  // T = double
    cfg.maxmem  = parse( args["--maxmem"] );    // T = std::size_t
    cfg.logfile = parse( args["--logfile"] );  // T = std::string
}

int main( int argc, char** argv ) {
    Config cfg;
    load( cfg,
        make_arg_map(argc, argv),
        []( std::string_view raw ) {
            return parse< Deduce >( raw );
        } );
}
```

More parsing In the next example, the target type is almost unknown, since it only exists inside an instantiation of a lambda.

```
class Settings;
template< typename T > T parse( const Settings &, std::string option );

Settings s{};

auto callback = []( auto width ) { /* ... */ };
apply_layout( callback,
    [s&]( std::string option ){ return parse< Deduce >( s, option ); }
);
```

4.4. Conversions and auto

Prevent unwanted conversions

```
template< Deduce typename T, typename U > T no_conversion( U ) {
    static_assert( std::is_same_v< T, U > )
}
int i = no_conversion( 1. ); // compilation error
```

Prevent return reference to temporary

```
struct Box {
    int value = 42;
    template< Deduce typename Out, typename Self >
        requires ( !std::is_reference_v<Out> || // value target: always fine
                  std::is_lvalue_reference_v<Self> // ref. target: lvalue only
                )
    Out get( this Self&& self ) {
        return self.value;
    }
};

Box b;
int _ = b.get(); // Out = int, Self = box& -> OK: copy
int& _ = b.get(); // Out = int&, Self = box& -> OK: reference
int _ = Box{}.get(); // Out = int, Self = box&& -> OK: copy out
int& d = Box{}.get(); // Out = int&, Self = box&&, -> compilation error
```

Forbid assigning to auto Some objects, e.g. proxies, should not be assigned to variables.

```
template< Deduce typename T > T foo();

auto _ = foo(); // compilation error
```

No auto on the lhs If one wants to avoid `auto` (e.g. in order to write all type names on the left side), it can often be replaced by `Deduce`.

```
auto i = cast_to< int >( "1" );
int i = cast_to< Deduce >( "1" );
```

4.5. Deducing overloads for `std::numeric_limits`, `std::numbers`

If this paper is accepted, we will propose that:

numeric_limits The following specialization is added:

```
template<>
struct std::numeric_limits< void > {
    template< Deduce typename T > max() {
        return std::numeric_limits< T >::max();
    }
    /* etc... */
};

//Usage:
float f = std::numeric_limits<>::max< Deduce >();
```

numbers The following constants are added:

```
template< Deduce typename T > T pi_a = std::numbers::pi_v< T >
    // _a for "auto", similar to ::pi_v, and in the mood of ::pi
    /* etc... */

//Usage:
float alpha = std::number::pi_a;
```

5. Name bike-shedding

List of possible keywords (since *Deduce* was just a placeholder):

- **auto**: This keyword is not currently used in this context, and I think it is the correct word to use for this feature. In particular, I propose using **auto**.
- **std::deduce**: Since we cannot use a new keyword, nor 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 already have this de facto with **std::declval**.
- **return**: Strange keyword, but it carries the meaning of “something from the return stuff”. We do not propose using that. In particular, because if that keyword is given, the target type is *additionally* considered, not *only* as the word **return** would suggest.
- **_** (Underscore): *Deduce* is not a pattern-matching construct, for which **_** is often used. Furthermore, it is not not very visible or searchable in the source code.

A. Why writing a proxy is not a good solution

One could solve this problem by letting the function return a proxy object, which stores the input parameter and has a templated user defined conversion function.

```
template< typename T >
struct proxy_midpoint {
    T a, b;
    template< typename Out > operator Out() {
        static_assert( std::is_same_v< T, Out > );
        return (a+b)/2;
    }
};

template< typename T >
proxy_midpoint< T > midpoint( T a, T b ) {
    return {a,b};
}
```

This has several downsides:

- This is a lot of boilerplate code.
- It does not work well for third party code, e.g. **std::numbers**.
- It is an ABI and API break if added later.
- It allows that the user stores this proxy object, which is itself a source of bugs.

```
auto x = midpoint( 2, 3 ); // User expects `x` to be a number, but it is a callable.
```

B. Existing practice in other languages

Surprisingly many languages have this feature. This list is not exhaustive.

F#

```
let parse<'T> (text: string) : 'T = failwith "..."  
  
let x: int = parse "42"  
let y: float = parse "3.14"  
//let z = parse "3.14" // does not compile
```

Haskell

```
read :: Read a => String -> a
read = undefined

n :: Int
n = read @_ "42"

x :: Double
x = read @_ "3.14"

y :: Int
y = read "42"
```

Java

```
static List<String> xs = new ArrayList<>();
static <T> T make() { ... }
static Integer x = make();
```

Kotlin

```
fun < T > parse( text: String ): T = TODO()
val x: Int = parse<_>( "42" )
val y = parse<Int>( "3.14" )
// val z = parse<_>( "3.14" ) // Does not compile
```

Lean

```
def myDefault {a : Type} [Inhabited a] : a := default
def x : Nat := myDefault -- a := Nat, from the target type
def s : String := myDefault -- a := String
```

Rust

```
fn parse< T: std::str::FromStr >( s: &str ) -> T {
    s.parse().unwrap()
}

let x: f64 = parse::<_>( "3.14" );
let y: i32 = parse( "666" );

fn parse<T: FromStr>(s: &str) -> T { ... }
let x: i32 = "42".parse().unwrap(); // T inferred from target
let v = iter.collect::<Vec<_>>(); // explicit turbofish with a hole!
```

Swift

```
func decode< T >( _ text: String ) -> T {
    fatalError()
}

let n: Int = decode( "42" )
let x: Double = decode( "3.14" )
// let u = decode( "3.14" ) // does not compile
```

Ceterum censeo operatorem implicationis esse necessarium.