Computed Deduction

Document #: D1107R0
Date: 2019-10-28
Project: Programming Language C++
Evolution Incubator Working Group
Evolution Working Group
Reply-to: Gašper Ažman
<gasper.azman@gmail.com>
Simon Brand
<simon@codeplay.com>
Andrew Bennieston
<a.j.bennieston@gmail.com>
Thomas Russel

Contents

1 Abstract

The inability to constrain function template argument deduction causes un-necessary template instantiations, as well as frequent use of SFINAE to disable specific undesirable overloads. This paper proposes a way to intercept the deduction of template arguments and compute them.

2 Problem Statement

Template argument deduction is a basic feature of C++ templates. There are several quite distinct uses of it:

- to instantiate a generic algorithm for a given type (`template <typename T> auto min(T, T)`)
- handle various combinations of cv-ref qualifiers (`template <typename T> auto foo(T&&)`)
- disable a function instantiation using SFINAE or the new `requires` machinery (`template <typename T> auto foo(T) requires predicate<T>`)
- combinations of above

For an illustration, humor a contrived example. Consider a `get` for a simple value container, which is intended to be inherited from:

```
struct box {
  std::vector<int> value;
};

template <typename Box>
decltype(auto) get(Box&& x)
    requires std::derived_from<B, box> {
  return std::forward<Box>(x).box::value; /* access the *box* value */
}
```

The intention is to forward the value-category of the box to the accessed value. There are only 8 possible useful instantiations of the `get` function:

- `get<box&>`
- `get<box&&>`
- `get<box const&>`
- `get<box const&&>`
- and possibly their `volatile` versions, though those are unlikely to be useful.

Recall that `box` is intended to be inherited from. For any `struct derived : box {};`, `get` will get an instantiation of `get<derived[(const|volatile)?(&|&&)>`, leading to code bloat.

While the example *is* contrived for clarity, should `get` be a more complex function, such code bloat becomes a problem for library implementers. As templates become more useful, the problem gets worse, because there are more reasons for function template instantiations that result in identical (or functionally equivalent) assembly:

1. C++11: forwarding references add an orthogonal reason to overload,
2. C++20: concepts encourage more function templates by much easier constraints,
3. c++20: consteval functions called in the function template instantiation process inflate compile times,
4. C++2b: When [[p0847r1]] (or spiritual successor) allows multiple concrete overloads to become a single function template,
5. C++2c: Reflection and subsequent procedural code-generation driven from function template instantiations inflates compile times even more.

This is great for code clarity and power. We *want* the above. Instantiations also come with rising costs, which broadly fall into two classes:

Compilation / Executable costs: For every function template instantiation, the compiler has to do an increasing amount of work. The frontend has to produce an instantiation, the the IR has to be optimized, and

assembly emitted; along with it, exception tables and debug symbols. This costs compile time, link time, and space; it also has adverse effects on code cache-friendliness since we have many copies of the same code.

There are compiler (and linker) optimizations that help with the space usage (see "COMDAT folding / icf"), but they cannot reduce the number of exported symbols, and they again cost compile time.

Code Clarity due to relaxed assumptions: We cannot assume the base-class when only matching on qualifiers is desired.

We have to write this:

```
template <typename Box>
decltype(auto) get(Box&& x)
  require std::derived_from<
    std::remove_cvref_t<Box>, box>>
{
  // Need to to qualify every access
  // in case of shadowing
  return std::forward<Box>(x).box::value;
}
```

When we would like to write this:

```
template <typename Box
          /* but coerce to
             box[const?(&|&&)] */>
decltype(auto) get(Box&& x)
{
  // we can assume x is a reference
  // to a box, and not a derived
  return std::forward<Box>(x).value;
}
```

When writing templates that are really supposed to operate on a particular base-class type, one has to qualify every member access with the type's name, because derived classes can shadow those member functions and members.

3 Use-Cases

This paper proposes a mechanism to compute the final deduction of a function template parameter before template instantiation.

3.1 Reduce the number of necessary template instantiations

This is largely the first example above. TODO write it out.

3.2 Make code clearer by tightening preconditions

Deducing to a base type is impossible to do with concepts. TODO flesh this out.

We have to write this:

```
template <typename Box>
decltype(auto) get(Box&& x)
  require std::derived_from<
    std::remove_cvref_t<Box>, box>>
{
  // Need to to qualify every access
  // in case of shadowing
  return std::forward<Box>(x).box::value;
}
```

When we would like to write this:

```
template <typename Box
          /* but coerce to
             box[const?(&|&&)] */>
decltype(auto) get(Box&& x)
{
  // we can assume x is a reference
  // to a box, and not a derived
  return std::forward<Box>(x).value;
}
```

3.3 Allow deduction to vocabulary type

When putting together two libraries based on expression templates that both understand a small set of concrete vocabulary types at the boundary, computed deduction would solve the `operator auto()` problem.

TODO flesh this out

3.4 Allow deduction to appropriate type-erased wrapper

Say that we have a library with the interface of

```
/* for buffer sizes of 8, 16, 32, 64, 128, 256, 512 */
void register_callback(in_place_function<SmallBufferSize, void()> f);
```

Unfortunately, these are highly ambiguous when one considers something like `register_callback([&]{ notify(); });`.

One would like to offer a frontend for this, to auto-select the overload:

```
template <std::invocable<> F>
void register_callback(F&& f) {
  using function_t = in_place_function<SmallBufferSize_, void()>;
  register_callback(function_t(std::forward<F>(f));
}
```

but this generates a separate template instantiation for every F, leading to code bloat.

It also *captures too much*! Because of the way we capture `F`, every other template in the overload set will be less specialized. We only wanted to select the buffer size - not hijack the entire overload set.

What we need is something more along the lines of

```
template <std::invocable<> F
          /*deduce-to*/ in_place_function<std::ceil2(sizeof(F)), void()>>>
void register_callback(F&& f) {
  /* F is an in_place_function inside the body. */
}
```

This allows us to compute the specialization without hijacking the entire overload set.

4 Proposed Solution

We propose a mechanism to allow a metafunction to compute the final deduction from the first-pass deduction that occurs in C++17.

The syntax is higly preliminary, but the semantics do not have a whole lot of wiggle room.

4.1 Example

```
template <typename Box : std::copy_cvref_t<Box, box>>
decltype(auto) get(Box&& x) {
  return std::forward<Box>(x).value;
}
```

— `copy_cvref_t` copies (and overwrites) any cv-ref qualifiers on its second parameter with the ones on its first
— There is no need to use `x.box::value` anymore, as `copy_cvref_t<Box, box>` always results in a cv-qualified `box`

— the `requires` clause is no longer (strictly) necessary, since a reference to a `box` will always only bind to to `box`es and their derived classes.

5 Syntax

From section 13.1 [temp.param]:

template-parameter:
    type-parameter
    parameter-declaration

type-parameter:
    type-parameter-key...$_{opt}$ *identifier*$_{opt}$ *deduction-expression*$_{opt}$...$_{opt}$
    type-parameter-key identifier$_{opt}$ = *type-id deduction_expression*
    type-constraint...$_{opt}$ *identifier*$_{opt}$ *deduction-expression*$_{opt}$...$_{opt}$
    type-constraint identifier$_{opt}$ = *type-id deduction_expression*
    template-head type-parameter-key...$_{opt}$ *identifier*$_{opt}$
    template-head type-parameter-key identifier$_{opt}$ = *id-expression*

type-parameter-key:
    `class`
    `typename`

deduction-expression:
    `:` id-expression

6 Proposed Semantics

This section describes the feature using a few "as if rewritten as" sections, each describing a part of the proposed mechanism.

6.1 Deduction

6.1.1 For a type template parameter

```
// template <
CONCEPT T = DEFAULT_EXPR : DEDUCTION_EXPR
// > void
```

1. The deduction of `T` proceeds normally until `T` is deduced as per C++17 rules, with any default initializer expressions executing if necessary. Let us name this result the *initial deduction*.
2. Immediately after the initial deduction is known, but before executing any `requires` constraints, execute `DEDUCTION_EXPR` in the with the same set name bindings available as the `DEFAULT_EXPR` would have (or has) been run with, with the addition of `T` being bound to the *initial deduction*. Let the value of `DEDUCTION_EXPR` be the *final deduction*. If `DEDUCTION_EXPR` does not evaluate to a type, this results in a substitution failure (SFINAE).
3. Any `requires` expressions that would be run in C++17 are run now, with the name `T` being bound to the *final deduction*.

Deduction of following parameters is done with the name `T` being bound to the constrained deduction.

6.1.2 For a value template parameter

The algorithm is exactly the same, but the the expression after the colon has to result in a a value. Basically, `DEDUCTION_EXPR` has to result in something that can be bound to the way the template parameter is declared.

6.1.3 For a template-template parameter

See values. Same rules - if it binds, it works, if it doesn't, SFINAE.

6.1.4 FAQ: Why should default template arguments execute before deduction exressions?

6.2 Function signature construction

Same as now - the deduced parameters are substituted back into the function signature (and the body of the template), with *deduced parameters* now meaning *final deduced parameters*. This may result in an invalid signature, which is a SFINAE condition.

6.3 Overload set construction

The construction of the overload set is unchanged, once one takes into account that candidates are generated differently than before. Compared to C++17, the overload set consists of functions instantiated from the very same candidate templates as before, though their signatures may be different. If two templates generate the same function signature, the result is ambiguous, and therefore results in an invalid program (diagnostic required).

7 Examples

8 FAQ

8.1 Can I use a previously deduced parameter in a `DEDUCTION_EXPR`?

Yes! This should work:

```
template <
  typename T : like_t<T, box>
  typename U : decltype(declval<T>().value)
> foo(T&&, U) {}
```

`T` always deduces to some cv-qualified version of `box` or `box&`, and `U` is coerced to the declval of the box's value. Note that `T` is the already fully deduced `box` in `U`'s `deduction-expr`.

8.2 Can I use the *initial deduction* in other template parameters?

In other words, given

```
template <
  typename T : long /* T will *always* be long */,
  typename U = T
>
void foo(T) {}
```

is it possible to have `U` deduce to `int` instead of `long` in the call `foo(1)`?

The answer is *no*. There is no way to access the *initial deduction* outside of the `deduction-expr` (though I'm sure clever metaprogrammers can find a way to export it somehow).

8.3 What if the final signature doesn't bind to the given parameters?

The scenario is the following:

```
template <typename T : int>
void foo(T) {}

foo(nullptr);
```

The initial deduction for `T` is `nullptr_t`, but the `deduction-expr` for `T` forces it to be `int`. The resulting signature is `foo(int)`, which does not match, and is removed from the overload set. In the absence of additional overloads for `foo` this fails with a compilation error because there were no matching functions to call.

8.4 What happens if two templates generate the same overload

Same as now - if the best match is ambigous, the program ill-formed (diagnostic required). Two templates resulting in the same best-match overload is a special case of this eventuality.

8.5 Could Concepts Solve This?

No. Concepts can only answer the question of whether to admit or remove an overload once it has already been enumerated as a candidate for the overload set, which is almost no better than `enable_if`, because it happens *after* template argument deduction has already occurred. In this case, we need to change the template argument deduction rules themselves, so that the template parameter itself is deduced in a programmable fashion, and *then* perhaps constrained by a concept.

9 Acknowledgements

The authors would like to thank Alisdair Meredith, especially, as he had both the original complaint about the deduction rules, and the first workable suggestion of how to fix it. This solution is far more general, but every proposal needs a spark.

The authors would additionally like to thank everyone (as I don't think there was anyone who remained silent) who attended the C++Now 2018 talk "My Little *this deduction: Friendship is Uniform", for their helpful comments, commentary, hints, tips, and ideas, whithout which this paper would not have gotten the idelological momentum to be born.