User specified type decay

Document #:	P3398R0
Date:	2024-09-17
Project:	Programming Language C++
Audience:	EWGI

Reply-to:	Bengt Gustafsson
	<bengt.gustafsson@beamways.com>

Contents

1 Abstract

This proposal adds a class specifier `decays_to(T)` which can be placed where `final` is currently placed. A type with such a specifier decays to an instance of the *target type* T whenever a placeholder type or template parameter is deduced from it. This is useful to prevent dangling references.

The proposal also includes a way to disable the decay which is useful in some scenarios. This proposal uses the existing keyword `explicit` to disable decay.

A couple of type-traits to inspect types that may or may not have `decays_to` specifiers are also included.

2 Motivating examples

2.1 String interpolation

The motivating example for this proposal is the upcoming *string interpolation* proposal (aka. f-literals) where a naive specification would require the result of a f-literal to be a call to `std::format`. This has the same detrimental performance effect as sending the result of `std::format` to `std::cout`. The `std::print` functions were created to avoid this performance hit so we don't want to re-introduce it if string interpolation is standardized. Instead the string interpolation proposal will include a `formatted_string` type which encapsulates the `format_args` and `format_string` objects needed to call `vformat` but which can also be accessed individually.

```
// Type constructed by f-literals.
template<...> struct formatted_string decays_to(std::string) {
    ...
};

std::string x = "x";
auto s = f"value {x + "y"}";        // s is a std::string containing "xy". No dangling.

// A new std::print overload is introduced
extern void std::print(const formatted_string& s);

std::print(f"value {x + "y"}");     // No performance degradation
```

Note that without this proposal `s` would be a `formatted_string` containing a dangling reference to the intermediate result of `x + "y"` which is only alive during the initialization of s. And even if the initializer did not contain temporaries the delayed evaluation would cause unexpected results in cases like this:

```
std::string x = "x";
auto s = f"value {x}";

x = "y";
std::string s2 = s;         // s2 contains "value y" without this proposal.
```

This does not dangle as x is still in scope, but as the conversion from `formatted_string` occurs when `s2` is initialized x has already been reassigned at this point.

2.2 Expression templates

When implementing for instance a matrix arithmetics library it has distinct performance advantages to not let each operator return a matrix but instead an object *representing* the operator, with data members referring to its operands.

Users of such libraries may be unaware of this implementation detail and expect a result declared as an `auto` variable to be a matrix. While the library can be designed so that the objects representing operators fulfill

the *concept* of the matrix type accessing elements will re-calculate the result each time, with possibly large performance degradation as result.

Introducing a matrix arithmetic library into the C++ standard is not far fetched after the introduction of `mdspan` and `mdarray` so this proposal is a useful building block for not only user defined libraries but also for future standardization.

Here is an example using a non-template Matrix type. A similar implementation for mdspan/mdarray would be too long to show here.

```
template<typename L, typename R, typename Op>
class Binop<L, R, Op> decays_to(Matrix) {
public:
    Binop(explicit const& L lhs, explicit const& R rhs) :
        lhs(lhs), rhs(rhs) {
    }

    operator Matrix() {
        Matrix ret;
        for (int r = 0; r < lhs.height(); r++)
            for (int c = 0; c < lhs.width(); c++)
                ret[r, c] = operator[](r, c);

        return ret;
    }

    double operator[](size_t r, size_t c) const { return op(lhs[r, c], rhs[r, c]); }
    size_t width() const { return lhs.width(); }
    size_t height() const { return lhs.height(); }

private:
    L lhs;
    R rhs;
    OP op;
};

template<typename LHS, typename RHS>
explicit auto operator+(LHS&& lhs, RHS&& rhs)
{
    return Binop<LHS, RHS, std::plus<>>(lhs, rhs);
}

Matrix a, b, c;

Matrix d = a + b + c;               // Do both additions element by element.

auto e = a + b + c;                 // e is also a Matrix

explicit auto p = a + b;       // Avoid decay

auto f = p + c;              // f is also a Matrix with same performance as d and e.
```

Note that the return type of operator+() is annotated as `explicit`, otherwise a Matrix object would have been returned, defeating the purpose of the design.

The variable `p` is also marked `explicit` which prevents the decay to Matrix. Decay instead occurs after adding

the c matrix to produce f. This idiom can be used to avoid having to write very long expressions just to retain the performance of the library.

3 Proposal

This proposal consists of a new class specifier using the *special identifier* `decays_to` which can be placed after a class name but before any base classes. Thus it is a new *class-virt-specifier*. `decays_to` is followed by a parenthesis containing a *type-id*.

decays-to-specifier:
`decays_to` `(` *type-id* `)`

class-virt-specifier:
`final`
decays-to-specifier

The type denoted by the *type-id* of the `decays_to` specifier is called the *target type* of the user specified type decay.

When an expression with a `decays_to` specifier on its cvref-free type is used to deduce a placeholder type or template type-parameter the deduction occurs as if the expression had the target type.

A `decays_to` specifier does not indicate *how* the decay occurs, a valid conversion sequence from the type with the `decays_to` specifier to its target type must exist. Not being able to perform the conversion is *ill-formed no diagnostic required*.

When the decay is performed this is considered an *explicit* constructor call, so explicit constructors and conversion functions are considered. The motivation of this is that the `decays_to` specifier is akin to a static_cast which does consider explicit constructors and conversion functions.

Decay occurs in all contexts where type deduction is performed, including local and global variables, function parameters, return values, non-type template parameters and structured bindings.

In the case of structured bindings the decay applies to the entire object before destructuring into separately named elements occurs. Thus the number of elements of the structured binding must match the number of elements in the target type.

For the placeholder type `decltype(auto)` the deduced type is exactly the target type. In all other cases the cvref qualifiers of the placeholder type or template type-parameter are applied to the target type to arrive at the resulting type just as if the initializer expression had the target type.

3.1 Transitive application

If the cvref-free target type of a `decays_to` specifier also has a `decays_to` specifier it is applied in sequence until a type without `decays_to` specifier is reached. If the target type of the final `decays_to` specifier denotes an array or function type the implicit decay of the type to pointer type is applied. However, only the latter can actually happen (the case of a function reference converting to a function pointer), as arrays can't be constructed from any user defined type and thus are not a valid target type for a `decays_to` specifier.

Decaying to itself or in a loop is an error and must be detected by the compiler to avoid endless loops during compilation.

3.2 Forward declarations

When declaring a class with a `decays_to` specifier in its definition the same specifier must be present. The *type-id* of the two `decays_to` specifiers must refer to the same target type.

As enforcing this rule across TUs is not possible it must be *ill-formed, no diagnostic required*. However, within the compilation of one TU compilers should issue a diagnostic when mismatched or missing `decays_to` specifiers on declarations/definitions of the same class type are found.

The change to allow `decays_to` specifiers on class declarations is somewhat complicated to do. The change must be in *elaborated-type-specifier* but only in the case referred to as class declaration, which is described in clause 2 of its description shown here. One way of describing this grammar change would be to only change clause 2's grammar snippet to:

`class-key` *attribute-specifier-seq* $_{opt}$ *identifier decays-to-specifier* $_{opt}$;
`class-key` *attribute-specifier-seq* $_{opt}$ *simple-template-id decays-to-specifier* $_{opt}$;

This could be considered too obscure as the *decays-to-specifier* is not mentioned in the grammar production for *elaborated-type-specifier*. Then again, neither is the *attribute-specifier-seq* of the second alternative, which offers precedent of allowing additional grammar elements if the condition at the top of clause 2 is fulfilled.

An alternative would be to add the *decays-to-specifier* directly in the grammar and then add a clause to explain that it is only allowed if the condition of clause 2 is fulfilled.

3.3 Disabling of decay

Decay can be disabled by prefixing the placeholder type or template type-parameter with the existing keyword `explicit`. This applies to all contexts where a placeholder type or template type-parameter can be deduced. This specifier makes type deduction ignore any `decays_to` specifier of the argument type, but does nothing if there is no such specifier.

When the `explicit` keyword is combined with `decltype(auto)` the cvref qualification of the expression type is retained just as if none of `explicit` and `decays_to` existed.

The grammar addition to allow `explicit` where appropriate is made in *type-specifier*. While this makes `explicit` grammatically allowed in all declarations it is semantically only allowed when type deduction with decay can occur, i. e. when other type-specifiers refer to a placeholder type or template type-parameter with optional cvref-qualification. *Note that this excludes deduction of template type-parameters from pointer types, array types or function pointer types.*

type-specifier:
  simple-type-specifier
  elaborated-type-specifier
  typename-specifier
  cv-qualifier
  `explicit`

If the `explicit` specifier is placed on a function parameter from which a template type-parameter can be deduced but the template type-parameter is explicitly given at the call site the `explicit` specifier has no effect at that call site. Example:

```
template<typename T> void f(explicit const T& x);

f<std::string>(f"Not deduced {"and not decayed"}";  // Call f<std::string>
```

3.4 Type traits

The existing trait `std::decay` should return the the cvref-free target type if applied to a type with a `decays_to` specifier. A new trait which gives the exact target type as suggested in [N4035] is also proposed. A reasonable name would be `std::decays_to_type` and as this should return the incoming type for types without `decays_to` specifier. It may be of interest to add a bool predicate `std::has_decays_to` indicating whether a type has the specifier. This predicate can be defined as:

```
template<typename T> bool has_decays_to_v = is_same_v<T, decays_to_type<T>>;
```

As it probably doesn't have many uses it may be left out of the standard.

4 History

This proposal is an elaboration of the [N4035] proposal *Implicit Evaluation of "auto" Variables and Arguments*, dated 2014, with the authors Joël Falcou, Peter Gottschling and Herb Sutter. This proposal suggests *user specified type decay* to identify this mechanism instead of *implicit evaluation* as evaluation is such a broad term. The expression template example is elaborated from an N4035 example.

4.1 Language development

During the ten years since N4035 several changes to the C++ language that are relevant for this proposal have occured:

— decltype(auto) of C++14
— CTAD of C++17
— placeholder types for function parameters of C++17
— Structured bindings of C++17
— `auto` NTTPs of C++17.

Of these changes CTAD has the most profound effect, and makes the straw poll "See a new proposal based on this one, but with `using auto` only applying to variable and data member initializers." (which got a weak consensus) less feasible. CTAD also makes the idea of only applying the decay for by-value types but not for references infeasible.

`decltype(auto)` could possibly be used as a way to disable the decay but this comes with the drawback that it can't be combined with cvref qualification to control which may be of interest when disabling decay. This proposal therefore retains the N4035 syntax of an `explicit` specifier to disable decay, even though it is something of a misnomer. An alternative would be to invent a new keyword which has its own challenges. `protected` could be an alternative, but it is very vague.

Placeholder types for function parameters, structured bindings and NTTPs do not introduce any new problems for this proposal, just new contexts in which it is applicable, and thereby where the `explicit` specifier can be applied.

5 Design decisions

The introduction of CTAD forces the design to differ from N4035 in some ways. The issue of incomplete classes was furthermore not covered in N4035. The reasons for not limiting decay to initializers of variables are also detailed below, as that would also create unreasonable results with CTAD and to some extent in any function using the decayed placeholder type or template type-parameter.

These design decision discussions provide rationale for the details of the proposal.

5.1 References must decay

Due to CTAD a specification where only values, not references, decay would create unreasonable results. Here is an example based on the string interpolation use case:

```
template<typename T> class THolder {
    THolder(const T& v) : member(v) {}

    T member;
};
```

```
std::string x;
THolder t(f"value {x + "y"}");      // CTAD!
```

If references don't decay T is deduced to `formatted_string` which means that `member` is a formatted_string and `t.member` will contain a dangling reference. Similar problems can occur with functions for instance if they declare static variables of type T.

5.2 Incomplete types

If `decays_to` can only be placed on a class definition deduction outcomes can be different depending on if the class is complete or not, and it is not possible to detect that this has happened (especially not between TUs).

Here is an example of the confusion that can arise if an incomplete type is used to deduce a function parameter type.

```
class formatted_string;

void f(const auto& c)
{
}

extern const formatted_string& get_fs();

f(get_fs());            // Calls f(const formatted_string&)

#include <format>       // Defines formatted_string with decays_to(std::string)

f(get_fs());            // Calls f(const std::string&)
```

It is not entirely clear to the author if this would cause problems in real-life situations as the possibilities of using an incomplete type are very limited. It is clear that it is *possible* to detect the difference between the above two f calls for instance using a static variable in the function and returning its address, but it is harder to come up with cases where it actually matters.

The syntax `using auto = T;` of N4035 is not amenable to a class *declaration* which is one reason to replace it with a specifier after the class name, which can be allowed when forward-declaring the class as well:

```
class formatted_string decays_to(std::string);
```

To play it safe this proposal requires a matching `decays_to` specifier on all declarations of class types which have a `decays_to` specifier in their definitions. Breaking this rule is an ODR violation which must be *ill-formed no diagnostic required* to be able to handle separate compilation of TUs where some may never see the class definition.

It may be possible to annotate the mangled class name to be able to diagnose mismatched `decays_to` specifiers between TUs by causing linker errors, but this can typically not catch errors in template classes and may reduce the possibilities of adding `decays_to` specifiers on existing classes in some environments where re-compilation is problematic.

5.3 Transitive application

The `noeval` idea to use a library feature to prevent decay presented in N4035 relies on `using auto = T;` declarations to not be transitive, that is, if T itself has a `using auto = U;` declaration it is *not* applied transitively. Or maybe it relies on references not being subject to decay, but then it would not work at all.

In this proposal decay is transitive: If one class decays to another class with a `decays_to` specifier this decay is also applied. This is consistent with how `operator->` works as well as the *implicit conversion functions* of

[P3298R0?].

5.4 Non-const references

As decay occurs regardless of cvref qualification of the placeholder type it is possible that after decay a non-const reference tries to bind to a temporary, which will fail as usual. This is logical as types with **decays_to** specifiers are designed to decay (except if the decay is disabled). This means that there are usually no variables that a non-const reference can refer to, these have already decayed when initialized.

Here is an example using `formatted_string` as before:

```
void modify_string(auto& v) { v += "z"; }

int x;
auto s = f"{x}";

modify_string(s);       // This works, s is a std::string.

modify_string(f"{x}");  // Does not compile, the formatted_string decays to std::string
```

On the last line it seems very natural that the call fails as there is no lvalue that can be modified.

It is possible to declare that a type decays to a T& for some T and this would compile in the above case.

5.5 Syntactical ambiguity

Just as for `final` there is a syntactical ambiguity when a *class-head-name* is followed by a `decays_to` and the solution is to disambiguate in favor of the specifier, just like for `final`. This introduces a small risk of code breakage, just as introducing `final` did.

Here is the standard's example for `final`:

```
struct A;
struct A final {};      // OK, definition of struct A,
                        // not value-initialization of variable final

struct X {
 struct C { constexpr operator int() { return 5; } };
 struct B final : C{};  // OK, definition of nested class B,
                        // not declaration of a bit-field member final
};
```

and this proposal adds a similar example for `decays_to` albeit somewhat different as there is a mandatory `(` after the specifier name.

```
struct A decays_to(int);
struct A decays_to(int) {};      // OK, definition of struct A,
                                 // not function called decays_to returning an A.

struct X {
 struct C { constexpr operator int() { return 5; } };
 struct B decays_to(int) {};     // OK, definition of nested struct B,
                                 // not member function called decays_to.
};
```

Well in contrast with `final` there is actually only one case, it doesn't matter if the declared struct is nested or not. So the standard example probably doesn't need to show both.

5.6 Decay at return

For consistency returning a type with a `decays_to` specifier from a function declared to return a placeholder type should decay the returned value. This seems sub-optimal as this prevents disabling the decay at the call site of the function, but here is an example of why it is necessary:

```
auto f()
{
    std::string x = "x";
    return f"value {x}";
}
```

Not decaying to a std::string would return a formatted_string that contains a dangling reference so *by default* decay must happen at return.

There are cases where we may want to disable decay in the return which was shown in the `operator+()` overload of the expression template example above. In this case the `explicit` specifier is placed on the return type of the function.

5.7 Function overload resolution

This proposal has no impact on function overload resolution, if a function overload which does not imply deduction of an argument type is selection no decay occurs. If a function overload where an argument type is deduced is selected the function specialization for the decayed type is called, unless the parameter is marked `explicit` to prevent the decay.

5.8 Invalid conversions

This proposal makes placing a `decays_to` specifier with a target type that can't be implicitly converted from the type containing the `decays_to` specifier *ill-formed, no diagnostic required* as it may be a bit complicated for the compiler to determine if conversion to the target type is possible, and once a conversion is attempted an error will be issued by current mechanisms anyway if no valid conversion sequence is found. This also opens up for specifying incomplete classes (by reference or pointer) as the target type.

An alternative is not making invalid conversions an error at all, leaving diagnostics to the first time that the decay is needed. The drawback is that diagnostics may be unnecessarily delayed and multiplied to each usage site.

5.9 decltype

The decltype of an expression of a type with a `decays_to` specifier could either be the expression's type or the target type. In this proposal decltype is always the undecayed type. A trait is instead provided which gives the decayed type from an original type. If decltype was defined to give the decayed type there would be no way to get the undecayed type and no way to make a trait for it as the trait would need the user to use decltype first.

5.10 Arrays, pointers and functions

Decay only applies when a type with a `decays_to` specifier is by value or by reference with possible cv qualification. Pointers to types with `decays_to` specifiers, arrays of such types or function pointers to functions returning such types or taking such types as parameters are not affected. This comes quite natural, as seen from these examples:

```
formatted_string f = f"Square root of 2: {std::sqrt(2.0)}";

auto p = &f;        // formatted_string*, not std::string*.

template<typename T, size_t N> int back(T (&arr)[N]) { return arr[N - 1]; }
```

```
formatted_string farr[] = { f"Square root of 2: {std::sqrt(2.0)}" };

back(farr);          // Calls with T == formatted_string, not std::string.

extern void std::print(const formatted_string& s);

template<typename R, typename A> void fpf(R(*fp)(A&&));

fpf(std::print);   // A is deduced to const formatted_string&, not const std::string&
```

This behavior is created naturally by the proposal's specification that the cvref-free type must have the `decays_to` specifier, as none of pointer, array or function pointer types can have `decays_to` specifiers.

6 Future design direction

If this proposal is accepted it is feasible to widen the new meaning of `explicit` to include array types of fixed bound (or templated bound), thus allowing by value passing and returning of arrays. For consistency this would also be a way to initialize an array from another array, but this purpose seems to be better served by just allowing both initialization and assignment of arrays. To allow initialization but not assignment seems very odd.

7 Implementation experience

None

8 Wording

No

9 Acknowedgements

Thanks to my employer, ContextVision AB for sponsoring my attendance at C++ standardization meetings.

10 References

[N4035] P. Gottschling, J. Falcou, H. Sutter. 2014-05-23. Implicit Evaluation of "auto" Variables and Arguments.
https://wg21.link/n4035