Structured bindings with polymorphic lambas

Aaryaman Sagar (aary800@gmail.com)

August 14, 2017

1 Introduction

This paper proposes usage of structured bindings with polymorphic lambdas, adding them to another place where `auto` can be used as a declarator

```
std::for_each(map, [](auto [key, value]) {
    cout << key << " " << value << endl;
});
```

This would make for nice syntactic sugar to situations such as the above without having to decompose the tuple-like object manually, similar to how structured bindings are used in range based for loops

```
for (auto [key, value] : map) {
    cout << key << " " << value << endl;
}
```

2 Motivation

2.1 Simplicity and uniformity

Structured binding initialization can be used almost anywhere `auto` is used to initialize a variable (not considering `auto` deduced return types), and allowing this to happen in polymorphic lambdas would make code simpler, easier to read and generalize better

```
std::find_if(range, [](const auto& [key, value]) {
    return examine(key, value);
});
```

2.2 Programmer demand

There is some programmer demand and uniform agreement on this feature

1. Stack Overflow: Can the structured bindings syntax be used in polymorphic lambdas
2. ISO C++ : Structured bindings and polymorphic lambdas

2.3 Prevalence

It is not uncommon to execute algorithms on containers that contain a value type that is either a tuple or a tuple-like decomposable class. And in such cases code usually deteriorates to manually unpacking the instance of the decomposable class for maximum reaadability, for example

```
return std::when_all(one, two).then([](auto futures) {
    auto& one = std::get<0>(futures);
    auto& two = std::get<1>(futures);

    return one.get() * two.get() * three.get();
});
```

The first three lines in the lambda are just noise and can nicely be replaced with structured bindings in the function parameter

```
return std::when_all(one, two).then([](auto [one, two]) {
    return one.get() * two.get();
});
```

3 Impact on the standard

The proposal describes a pure language extension which is a non-breaking change - code that was previously ill-formed now would have well-defined semantics.

4 Interaction with concepts and traits

Definition (`x-decomposable`) For a given domain of types that are decomposable, if a type can be decomposed into a structured binding expression with `x` bindings, then it is said to be `x-decomposable` (reference `[dcl.struct.bind]` for the exact requirements). More specifically, if the following expression is well formed in the least restrictive member access scope (where privates are accessible) for a type `T`

```
auto&& [one, two, three, ... , x] = o;
```

Where `decltype(o)` is `T`, then the type `T` is said to be `x-decomposable`

This can be made available to the compiler as both a concept and a trait. The presence of such a concept makes it easy to define templates in terms of a type that is `x-decomposable`. A trait allows for the same thing but can be considered more versatile as it also fits well with existing code that employs value driven template specialization mechanisms and other more complicated specialization workflows.

It is possible to make a concept or trait that enables us to check if a type is decomposable into `x` bindings by virtue of it's interface. In particular the presence of an ADL defined or member `get<>()` function and the existence of specialized `std::tuple_element<>` and `std::tuple_size<>` traits qualifies something to be `x-decomposable`. However, a type can be `x-decomposable` even when these are not present (see `[dcl.struct.bind]`p4)

`[dcl.struct.bind]`p2 and `[dcl.struct.bind]`p3 define decomposability that can be checked by the programmer at compile time (described above) via a concept or trait. However `[dcl.struct.bind]`p4 describes a method of unpacking that cannot be enforced purely by the language constructs available as of C++17. As such a compiler intrinsic, say `__is_decomposable<T, x>` is required. Given such an intrinsic, defining a trait and concept that check if a type is `x-decomposable` on top of that is trivial. The trait itself can be used as a backend for the concept, leaving the implementation of the concept entirely in portable code without the help of compiler intrinsics.

The concept, say `std::decomposable<x>` accepts a non type template parameter of type `std::size_t` that determines the cardinality of the structured bindings decomposition. This concept holds if a type is `x-decomposable` (and this will take into consideration the requirements set forth by `[dcl.struct.bind]` paragraphs 2, 3 and 4.

The corresponding trait, say `std::is_decomposable<T, x>` has value `true` if and only if type `T` is `x-decomposable`. The usual variable template `std::is_decomposable_v<T, x>` should also be defined.

5 Impact on overloading and function resolution

Lambdas do not natively support function overloading, however one can lay out lambdas in a way that they are overloaded, for example let's assume the following definition of `make_overload()` for the rest of the paper

```
template <typename... Types>
struct Overload : public Types... {
    template <typename... T>
    Overload(T&&... types) : Types{std::forward<T>(types)}... {}

    using Types::operator()...;
```

```
};
template <typename... Types>
auto make_overload(Types&&... instances) {
    return Overload<std::decay_t<Types>...>{std::forward<Types>(instances)...};
}
```

Now this can be used like so to generate a functor with overloaded `operator()` methods from anonymous lambdas

```
namespace {
    auto one = [](int) {};
    auto two = [](char) {};
    auto overloaded = make_overload(one, two);
} // namespace <anonymous>
```

In such a situation the consequences of this proposal must be considered. The easiest way to understand this proposal is to consider the rough syntactic sugar that this provides. A polymorphic lambda with a structured binding declaration translates to a simple functor with a templated `operator()` method with the structured binding decomposition happening inside the function

```
auto lambda = [](const auto [key, value]) { ... };

/**
 * Expansion of the above lambda
 */
struct ANONYMOUS_LAMBDA {
    template <std::decomposable<2> __Type>
    auto operator()(const __Type __instance) const {
        auto&& [key, value] = std::move(__instance);
        ...
    }
};
```

The `std::move()` is added to force a conversion to xvalue type because the expression `e` (see `[dcl.struct.bind]p1`) is not an lvalue in the structured binding declaration, and when `e` is not an lvalue, the introduced bindings are decomposed as if `e` was an xvalue. If `e` was an lvalue, (i.e. if `&` was used as the `ref-qualifier` or if `&&` was used and an lvalue was passed in to the lambda) then the `std::move()` will be omitted (see the next expansion for an example where `std::move()` is omitted)

Similarly a lambda that has two seperate groups of structured binding declarations will translate with the decompositions happening serially within the function body in order of binding declarations from left to right

```
auto lambda = [](const auto [key, value], auto& [one, two, three]) { ... };

/**
 * Expansion of the above lambda
 */
struct ANONYMOUS_LAMBDA {
    template <std::decomposable<2> __One, std::decomposable<3> __Two>
    auto operator()(const One __one, Two& __two) {
        auto&& [key, value] = std::move(__one);
        auto& [one, two, three] = __two;
        ...
    }
};
```

Given the above expansions, a polymorphic lambda behaves almost identically to a lambda with a auto parameter type with the difference that these are constrained to work only with parameters that are `x-decomposable`. And nothing special happens when overloading

```
namespace {
    auto one = [](int) {};
    auto two = [](auto [key, value]) {};
    auto overloaded = make_overload(one, two);
} // namespace <anonymous>
```

```
int main() {
    auto integer = int{1};
    auto pair = std::make_pair(1, 2);
    auto error = double{1};

    // calls the lambda named "one"
    overloaded(integer);
    // calls the lamdba named "two"
    overloaded(pair);
    // error
    overloaded(error);
}
```

5.1 Viable orthogonal overloads

One key point to consider here is that the concept based constraints on such lambdas allows for the following two orthogonal overloads to work nicely with each other

```
namespace {
    auto lambda_one = [](auto [one, two]) {};
    auto lambda_two = [](auto [one, two, three]) {};
    auto overloaded = make_overload(lambda_one, lambda_two);
} // namespace <anonymous>

int main() {
    auto tup_one = std::make_tuple(1, 2);
    auto tup_two = std::make_tuple(1, 2, 2);
    overloaded(tup_one);
    overloaded(tup_two);

    return 0;
}
```

Since here either one lambda can be called or both, in no case can both satisfy the requirements set forth by the compiler concept `std::decomposable<x>`

5.2 Access control and decompositions

Another key point to consider is acess control within the expansion of the lambda. Decompositions will share the access control powers of the code in the surrounding scope where the lambda is defined. So if the decomposition was in the body of the lambda and was valid (for example, even if the type being decomposed has private `get<>()` methods) the lambda would be able to decompose it successfully. For example the following code is valid

```
class Something {
public:
    static auto make_decomposer();
private:
    std::tuple<int, int> tup{1, 2};
    template <std::size_t Index>
    int get();
};

namespace std {
template <>
class tuple_size<Something> : public std::integral_constant<std::size_t, 2> {};
template <std::size_t Index>
class tuple_element<Index, Something> {
public:
```

```
    using type = int;
};
} // namespace std

template <std::size_t Index>
int Something::get() {
    return std::get<Index>(this->tup);
}

auto Something::make_decomposer() {
    // decomposition of Something instances is allowed access to the privates
    // of Something since it's defined in a context where private members are
    // visible
    return [](auto [one, two]) {
        assert(one == 1);
        assert(two == 2);
    };
}

void foo() {
    auto something = Something{};
    auto decomposer = Something::make_decomposer();

    // the decomposition here happens in the scope of the lambda so is valid
    decomposer(something);
}
```

6 Compatibility with ODR

A typical problem with traits classes comes from the ODR rule. We have the same potential problem with the trait defined earlier - `std::is_deomposable<>`. A type may be defined such that it is unable to be decomposed at one point in a program and might be decomposable at other points.

This problem comes from the nature of the structured bindings feature. To maintain backwards compatibility with types that were defined before the feature, structured bindings use traits types to detect whether types are decomposable. In particular one trait that is always used and needs to be defined for non array and non class types with only public members is `std::tuple_size`. By virtue of being a trait, it needs to be defined or specialized after a class's definition. This leads to possible discrepencies in whether it is complete or not at different points in the program. And if we instantiate the `std::is_decomposable` trait at these incompatible points we risk violating ODR.

There are three possible solutions to this problem

6.1 Disabled overloading based on decomposability

Disabling overloading based on decomposability is another solution to the problem of possibly violating ODR. With this we can treat all lambdas with structured bindings parameters as the normal equivalent lambda with a single non structured binding parameter with no decomposing and the same cv-ref qualifications. For example

```
auto one = [](auto one, auto&& [two, three], auto four, const auto& [five]) {};
```

Would be equivalent to

```
auto one = [](auto one, auto&& two, auto three, const auto& four)
```

This seems to be the simplest way forward. As a corrolary `std::is_decomposable` can still safely exist and be used in situations where programmers are guaranteed to get a decomposable type using the approach described in the next subsection

6.2 Poisoning the trait

One possible approach is to poison the `std::is_decomposable` trait to produce ill defined code when instantiated with a type that is not decomposable (i.e. a non-array, non-class type with only public data members and a type for which the required structured bindings specializations are not defined). This would work in concept something like this

```
template <typename T, typename = void_t<>>
class IsDecomposableImpl {
    /**
     * This would render the program ill formed when instantied with a non
     * decomposable type - including but not limited to types which do not
     * have std::tuple_size defined
     */
    static_assert(always_false<T>);
};

/**
 * Then the other specializations
 */
template <typename T>
class IsDecomposableImpl<T, ....> { ... };
...
```

This would limit the overloadability of lambdas which accept structured bindings parameters to work only with other lambdas accepting structured bindings parameters.

6.3 Argument Introspection

Like function arguments, programers should be able to detect the number of bindings to the structured bindings parameters in a lambda. So they can employ metaprogramming strategies. To enable this, there can be traits that list the number of function arguments allowed, which ones are structured bindings parameters, the types and cv qualifications of each function parameter (template types should be treated specially)

7 Conversions to function pointers

A capture-less polymorphic lambda with structured binding parameters can also be converted to a plain function pointer. Just like a regular polymorphic lambda

```
using FPtr_t = void (*) (std::tuple<int, int>);
auto f_ptr = static_cast<FPtr_t>([](auto [a, b]){});
```

So another conversion operator needs to be added to the expansion of the polymorphic lambda with structured bindings above

```
auto lambda = [](const auto [one, two]) { ... };

/**
 * Expansion of the above lambda in C++17 form with respect to overloading
 */
struct ANONYMOUS_LAMBDA {
    template <std::decomposable<2> __Type>
    auto operator()(const __Type __instance) const {
        auto&& [one, two] = std::move(__instance);
        ...
    }

private:
    /**
     * Cannot use operator() because that will either cause moves or copies,
```

```
     * elision isn't guaranteed to happen to function parameters (even in
     * return values)
     */
    template <std::decomposable<2> __Type>
    static auto __invoke(const __Type __instance) {
        auto&& [key, value] = std::move(__instance);
        ...
    }

public:
    /**
     * Enforce the decomposable requirement on the argument of the function
     */
    template <typename Return, std::decomposable<2> Arg>
    operator Return(*)(Arg)() const {
        return &__invoke;
    }
};
```

And like regular polymorphic lambdas, returning the address of the static function invokes an instantiation of the function with the types used in the conversion operator

8 Exceptions

Any exceptions during copy/move construction of the instance which is to be decomposed being will be thrown from the call site, just as with regular polymorphic lambdas. However, if there is an exception thrown during the decomposition process, for example if `get<>()` throws, that will propagate from within the lambda. So if function level try catch blocks were allowed for lambdas, those would catch any exceptions generated during the decomposition process, whereas exceptions from the copy/move construction for the creation of the entity `e` (see `[dcl.struct.bind]p1`) would not be caught by the imaginary function level try-catch block.

9 Acknowledgements

Thanks to Eric Niebler and Nicol Bolas for the suggestions and the help with this paper!

10 Changes to the current C++17 standard

10.1 Section 8.1.5.1 (`[expr.prim.lambda.closure]`) paragraph 3

For a generic lambda, the closure type has a public inline function call operator member template (17.5.2) whose `template-parameter-list` consists of one invented type `template-parameter` for each occurrence of `auto` in the lambdas `parameter-declaration-clause`, in order of appearance. For each occurrence of a structured binding with cardinality `x`, the `template-parameter-list` consists of an invented type `template-parameter` with the constraint that it has to be decomposable into `x` structured bindings (see `[dcl.struct.bind]`). And as such the function template only participates in overloading when all the structured bindings are appropriately decomposable. The invented type `template-parameter` is a parameter pack if the corresponding `parameter-declaration` declares a function parameter pack (11.3.5). A structured binding `parameter-declaration` cannot be used to invent a parameter pack. The return type and function parameters of the function call operator template are derived from the `lambda-expressions` `trailing-return-type` and `parameter-declaration-clause` by replacing each occurrence of auto in the decl-specifiers of the `parameter-declaration-clause` with the name of the corresponding invented `template-parameter`.

10.2 Section 11.5 ([dcl.struct.bind])

5. If the number of structured bindings introduced by a structured binding declaration is `x`, then a type `T` is called `x-decomposable` if the following is well formed in the least restrictive member access scope (where privates are accessible)

```
auto&& [one, two, three, ... , x] = o;
```

where `T` is `decltype(o)`

10.3 Section 23.15.4.3 Type properties ([meta.unary.prop])

```
template <typename T, std::size_t X>
struct is_decomposable;
```

Condition The type `T` has to be `x-decomposable` (see `[dcl.struct.bind]`p5)