Structured bindings with polymorphic lambas

Aaryaman Sagar (rmn100@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

```
auto map = std::map<int, int>{{1, 2}, {3, 4}};
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

```
auto map = std::map<int, int>{{1, 2}, {3, 4}};
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

```
auto result = std::count_if(map, [](const auto& key_value_pair) {
    const auto& key = key_value_pair.first;
    const auto& value = key_value_pair.second;

    return examine(key, process_key(key), value);
});
```

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

```
auto result = std::count_if(map, [](const auto& [key, value]) {
    return examine(key, process_key(key), value);
});
```

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`) 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 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 inspecting 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 its implementation entirely in user 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] = instance;
        ...
    }
};
```

Note that although this expansion has almost the same semantics as the actual lambda. The above is just for illustration purposes. It has some differences with the way such a lambda would translate into real code - for example, the introduced bindings are lvalue references to the referenced type of the bindings. In the actual code they would conditionally be lvalue references or rvalue references depending on the value category of the corresponding initializer. The referenced type of the bindings however, would remain the same in both cases; so the change in the actual reference type of the bindings can be considered a mere implementation detail.

Given the above expansion, a polymorphic lambda behaves 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);
}
```

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>`

6 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 polyorphic 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] = 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] = 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

7 Exceptions

Exceptions during "unwrapping" (if, for example a member or free ADL defined `get<>` throws) will propagate from the call site, just as with regular polymorphic lambdas

8 Changes to the current C++17 standard

8.1 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 at the call site. 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`.

8.2 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

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

where `T` is `decltype(o)`

8.3 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 `[expr.prim.lambda.closure]`p3)