Structured bindings with polymorphic lambas

Aaryaman Sagar (rmn100@gmail.com)

August 14, 2017

1 Introduction

This paper proposes usage of structured 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

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). Any compiler diagnostics provided are the same as that for regular structured bindings initializations - a compiler concept that determines if a type is `x-decomposable`.

Part of such a concept can be made using the tools available to the programmer. In particular, it is possible to make a concept or SFINAE trait that enables us to check if a type is decomposable via a member or ADL defined free `get<>()` method/function and the existence of a specialized `std::tuple_size<>` and `std::tuple_size<>` traits

From the current working draft of the C++17 standard `[dcl.struct.bind]`p2 and `[dcl.struct.bind]`p3 define decomposability that can be checked by the programmer at compile time via a concept or SFINAE 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 concept provided by the compiler is required.

This concept `std::decomposable<x>` shall be a concept that 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.

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

```
template <typename... Types> struct Overload { using Types::operator()...; };
template <typename... Types> Overload(Types...) -> Overload<Types...>;

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

And 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 in C++17 form with respect to overloading
 */
struct ANONYMOUS_LAMBDA {
    template <std::decomposable<2> Type>
```

```
    auto operator()(const Type instance) const {
        auto& [key, value] = instance;
        ...
    }
};
```

(Note that the above is just for illustration purposes. It has some differences with the way such a lambda would translate into real code - for example, returning a value from the structured bindings in the initialization would not be a candidate for NRVO because of the explicit disallowance in `[class.copy.elision]`, but here it might be)

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

```
template <typename... Types> struct Overload { using Types::operator()...; };
template <typename... Types> Overload(Types...) -> Overload<Types...>;

namespace {
    auto one = [](int) {};
    auto two = [](auto [key, value]) {};
    auto overloaded = 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 = 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

```
auto f_ptr = static_cast<void (*) (std::tuple<int, int>)>([](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

Change in 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-` 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` 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 (not from a structured binding `parameter-declaration`) `template-parameter` is a parameter pack if the corresponding `parameter-declaration` declares a function parameter pack (11.3.5). 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`.